Pergamon

Information Systems Vol. 22, No. 5, pp. 249–290, 1997
© 1997 Elsevier Science Ltd
All rights reserved. Printed in Great Britain
0306-4379/97 $17.00 + 0.00

PII: S0306-4379(97)00017-3

SCHEMA VERSIONING FOR MULTITEMPORAL RELATIONAL DATABASES†

CRISTINA DE CASTRO[1], FABIO GRANDI[2] and MARIA RITA SCALAS[1,2]

[1]Centro di Studio per l'Informatica e i Sistemi di Telecomunicazione, Consiglio Nazionale delle Ricerche
[2]Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna Viale Risorgimento 2, I-40136 Bologna, Italy

(Received 26 July 1994; in final revised form 13 May 1997)

Abstract — In order to follow the evolution of application needs, a database management system is easily expected to undergo changes involving database structure after implementation. Schema evolution concerns the ability of maintaining extant data in response to changes in database structure. Schema versioning enables the use of extensional data through multiple schema interfaces as created by a history of schema changes. However, schema versioning has been considered only to a limited extent in current literature. Also in the field of temporal databases, whereas a great deal of work has been done concerning temporal versioning of extensional data, a thorough investigation of schema versioning potentialities has not yet been made. In this paper we consider schema versioning in a broader perspective and introduce new design options whose distinct semantic properties and functionalities will be discussed. First of all, we consider solutions for schema versioning along *transaction time* but also along *valid time*. Moreover, the support of schema versioning implies operations both at intensional and extensional level. Two distinct design solutions (namely *single-* and *multi-pool*) are presented for the management of extensional data in a system supporting schema versioning. Finally, a further distinction is introduced to define *synchronous* and *asynchronous* management of versioned data and schemata. The proposed solutions differ in their semantics and in the possible operations they support. The mechanisms for the selection of data through a schema version are in many cases strictly related to the particular schema versioning solution adopted, that also affects the data definition and manipulation language at user-interface level. In particular, we show how the temporal language TSQL2, originally designed to support basic functionalities of transaction-time schema versioning, can accordingly be extended. ©1997 Elsevier Science Ltd

Key words: Transaction-Time, Valid-Time, Temporal Database, Multitemporal Database, Schema Versioning

1. INTRODUCTION

An ever growing interest for data versioning capabilities continues to involve database researchers, manufacturers and practitioners. Raw data, database structures and applications are evolving entities which require adequate support of past, present and even future versions. In particular, the need for supporting evolving database schemata (intensional data versioning) gave rise to schema evolution and schema versioning issues. A large bibliography is now available on these topics [10, 12]. In [13], Roddick provides an excellent survey on the state of the art. Two application areas are especially concerned with intensional versioning: CAD/CAM applications and temporal databases.

Temporal databases contain time-varying data [6, 8, 11, 19, 21, 22] and the interest on intensional data versioning arose as a logical extension of the work formerly done on extensions. Two time dimensions are usually considered in the framework of temporal databases: *transaction time*, which tells when facts are logically present and events occur in the database, and *valid time*, which tells when facts are true and events occur in reality [6]. Transaction time can be used to *roll back* the database state to a past point in time.

CAD/CAM, CIM and other engineering applications first put forward the requirement of managing multiple design versions. Due to application requirements, object-oriented database systems are also mostly concerned in this field (see [11, 13] for references to the wide bibliography). Since the temporal aspect of the versioning is charged to the system, the adoption of transaction time

†Recommended by Peri Loucopoulos

is easily understood. However, successive, but also parallel and even merging versions of the same objects are considered, giving rise to a notion of "branching" transaction time in opposition to the "linear" model used in temporal databases.

According to the temporal dimensions they support, temporal databases can be classified as *monotemporal* (*transaction-* or *valid-time*), *bitemporal* or *snapshot* [6]. Transaction-time databases record all the versions of data inserted, deleted or updated in successive transactions (current and non current versions). Valid-time databases maintain the most recently created versions of data, each relative to a distinct valid-time interval (current versions, forming the present historical state). Bitemporal databases support both transaction and valid times and thus maintain all the valid-time versions recorded by successive transactions (present and past historical states). Snapshot DBs do not support time: they maintain only the most recently inserted (current) version. A DB in which relations of different temporal formats (e.g. snapshot, valid-time and bitemporal) coexist can be called *multitemporal* [2].

Dozens of temporal data models and languages have been proposed in the last decade. Given this rich crop of proposals, it is worth remarking on the TSQL2 initiative: a committee formed by academic and industrial specialists gathered in 1993 to prepare a consensual temporal extension of the SQL standard [5]. The complete language design, also based on a consensual temporal relational model, was released in 1994 [17, 23]. TSQL2 was also intended as a synthesis to put into practice most of the best ideas proposed in TDB literature. Therefore, also schema versioning support [15] was included in the design. Only schema versioning along transaction time has so far been considered necessary in temporal databases and consequently embedded in TSQL2.

At query-language level, TSQL2 is basically augmented with a new stand-alone statement for schema version selection:

```
SET SCHEMA < datetime value expression >
```

For instance, after the execution of the statement:

```
SET SCHEMA DATE '1994-06-30'
```

only schema versions current as of the end of June 1994 are used to solve the queries which follow, until a new SET SCHEMA statement is executed.

In general, according to consensual definitions used by the temporal database community [6], three levels of schema support can be considered:

Schema change is provided by a system which simply allows modifications of the database schema: neither previous intensional nor extensional data are maintained;

Schema evolution is provided by a system which allows recovery of previous extensional data after each schema change, but does not support previous schema versions;

Schema versioning is provided by a system which allows both maintenance of extensional data and management of previous schemata after schema changes.

Example 1 Let us consider an example, also valid in a traditional (i.e. snapshot) database. Suppose an initial state as produced by the following SQL (or TSQL2) statements:

```
CREATE TABLE EMPLOYEE                                        S1
   ( NAME CHAR(10) NOT NULL PRIMARY KEY,
     ADDRESS CHAR(20), CITY CHAR(10) ) ;
INSERT INTO EMPLOYEE                                         S2
   VALUES('Brown','King's Road 15','London') ;
INSERT INTO EMPLOYEE                                         S3
   VALUES('Rossi','Via Veneto 7','Rome')
```

Statement S_1 creates a new (snapshot) relation schema SV1 by inserting the corresponding information into the catalogues, and a new table EMPLOYEE initially empty. Statements S_2 and S_3 insert two rows into the new table. Thus, the initial state can be depicted as follows:

SV1

`EMPLOYEE(NAME,ADDRESS,CITY)`

NAME	ADDRESS	CITY
Brown	King's Road 15	London
Rossi	Via Veneto 7	Rome

Then let us consider the following schema change:

```
ALTER TABLE EMPLOYEE                                      S4
   DROP COLUMN ADDRESS ;
ALTER TABLE EMPLOYEE                                      S5
   ADD COLUMN PHONE CHAR(8)
```

and examine what would happen with the three levels of schema support.

Schema Change

In a system only supporting schema changes, the effects of S_4 can be depicted as follows:

SV2

`EMPLOYEE(NAME,CITY,PHONE)`

NAME	CITY	PHONE
(empty)		

The old information must be restored by the user, thus the information concerning employees Brown and Rossi must be inserted (e.g. within the same transaction containing S_4 and S_5) as new complete rows as follows:

```
INSERT INTO EMPLOYEE                                      S6
   VALUES('Brown','London','224466') ;
INSERT INTO EMPLOYEE                                      S7
   VALUES('Rossi','Rome','775533')
```

The final state can be represented as:

SV2

`EMPLOYEE(NAME,CITY,PHONE)`

NAME	CITY	PHONE
Brown	London	224466
Rossi	Rome	775533

Schema Evolution

If our system supports schema evolution, the effects of schema change S_4 can be represented as follows:

SV2

`EMPLOYEE(NAME,CITY,PHONE)`

NAME	CITY	PHONE
Brown	London	*Null*
Rossi	Rome	*Null*

In this case, the contents of old columns NAME and CITY is automatically retained after the schema change. The new phone information can be stored with the following statements:

```
UPDATE EMPLOYEE                                           S8
   SET PHONE='224466'
     WHERE Name='Brown' ;
UPDATE EMPLOYEE                                           S9
   SET PHONE='775522'
     WHERE Name='Rossi'
```

The final state can be represented as before.

Schema Versioning

Finally, if the system supports schema versioning, we assume the table EMPLOYEE to have been created on 1/1/1990 and the schema change S_4 to be effected on 7/1/1994. Hence, we can represent as follows the effects of schema change (and updates S_8 and S_9), that is the two resulting schema versions with their associated view on EMPLOYEE data:

SV1 [1/1/90 - 6/30/94]

`EMPLOYEE(NAME,ADDRESS,CITY)`

NAME	ADDRESS	CITY
Brown	King's Road 15	London
Rossi	Via Veneto 7	Rome

SV2 [7/1/94 - ∞]

`EMPLOYEE(NAME,CITY,PHONE)`

NAME	CITY	PHONE
Brown	London	224466
Rossi	Rome	775533

The reader should note that we are still in the presence of snapshot data. We used snapshot data to show that schema versioning can be defined independently and also in absence of data versioning.

In a temporal framework, the two schema versions can be selected by means of time (e.g. before or after the schema change time, 7/1/1994). Therefore, an access to table EMPLOYEE through the TSQL2 fragment:

```
SET SCHEMA DATE '1993-01-01' ;                                    S10
SELECT * FROM EMPLOYEE                                            S11
```

is effected through the schema version SV1 and retrieves the same table as before the schema change.
An access by means of the TSQL2 statements:

```
SET SCHEMA DATE '1995-01-01' ;                                    S12
SELECT * FROM EMPLOYEE                                            S13
```

is effected through the schema version SV2 and produces the same table as in the final state of the schema evolution case.

Moreover, old applications written before the schema change, requiring the column ADDRESS (and not expecting to find the column PHONE), may still work on legacy data by means of schema version SV1. This is one of the main motivations for schema versioning. On the contrary, in a system only supporting schema evolution, all the old applications must be rewritten even in order to work on old data. □

Notice that, until now, we did not make any assumption or show any particular solution for the implementation of schema versions and, in particular, for the management of extensional data.

The TSQL2 schema versioning mechanism is based on the concept of *completed schema*, that is tables are defined over the union of all attributes ever defined for them, including deleted ones. Dropped columns are actually deactivated rather then physically removed. The full contents of a table associated with its completed schema can be retrieved with the "double asterisk" option in the SELECT statement as in the following one:

```
SELECT ** FROM EMPLOYEE
```

Following our example, the results of such a query would appear as:

NAME	ADDRESS	CITY	PHONE
Brown	King's Road 15	London	224466
Rossi	Via Veneto 7	Rome	775533

In this way, no modifications need to be made to the data model [17, Section 22.3]. As a matter of fact, the schema versioning support does not introduce any new level of data versioning and can be based on a "classical" view mechanism. In the example above, it is untrue that there are two versions of Brown's (or Rossi's) data, the former associated to SV1 (and its timestamps) and the latter to SV2: the value of the attribute CITY, for instance, cannot be different as it corresponds to a single stored value. For each piece of information, stored in a table according to the *completed* schema, there is a single line of evolution due to update activity. Different schema versions can be implemented in TSQL2 through view functions, that is syntactic devices acting as complex cast operators.

In this work we investigate new design options for extended schema versioning support and discuss the improved functionalities they may provide. The rest of the paper is organized as follows. A list of the options and basic motivations for the new design solutions are the subject of the next Section. Section 3 is devoted to the techniques for the management of all the proposed schema versioning solutions, including algorithms and a comprehensive example. Section 4 illustrates syntactic and semantic query language extensions in the environment considered. In Section 5, a comparative discussion of the different schema versioning solutions and conclusions can finally be found.

2. BACKGROUND, NEW SOLUTIONS, AND MOTIVATIONS

The temporal relational environment considered in this paper is characterized by the following features:

- Extensional data are *multitemporal* [2], that is we consider a database in which relations of different temporal format (snapshot, transaction-time, valid-time, bitemporal) coexist.
- All the principal "traditional" relational schema changes are considered, such as addition, elimination, renaming of an attribute or of a relation, change of the domain of an attribute.
- Since the purpose is to explain the semantics of the logical solutions for the management of schema versions, we limit the discussion to changes applied to a single relation[†].
- The modification of the temporal format of extensional data is also included as a possible schema change [18], as it is typical of a multitemporal environment. The management of this change is based on operations for the translation of a temporal relation from one temporal format to another [2].
- For the support of schema versioning at intensional level, in the sprit of the relational model, the catalogues are defined and managed as temporal relations. Therefore they are augmented with the required temporal attributes and their tuples are timestamped.
- For the sake of simplicity, among all the traditional relational system catalogues (on relations, attributes, indexes, views, etc.) we consider only the temporal extension of the two catalogues describing the database relations and their attributes, since they are those mostly involved in schema versioning.

2.1. Notation

We assume to deal with the same temporal data model adopted for the definition of TSQL2. Time is assumed to be discrete and is represented by means of *chronons* [6]. According to the BCDM (Bitemporal Conceptual Data Model) [7], tuples in temporal relations (data tables and catalogues) are timestamped with *temporal elements*, that is sets of chronons. Contiguous sets of bitemporal (monotemporal) chronons can efficiently be represented as rectangles (intervals) within elements.

[†]We do not deal with the consequences of schema changes on the relationships and on inheritances among entities. The management of all the possible schema changes, in particular those involving more than one relation at a time, is beyond the scope of this paper and of the relational model itself. A richer taxonomy and study on the subject can be found in [14].

The symbol "0" is used to denote the special values INITIATION in transaction time (i.e. the time the system was started), and BEGINNING in valid time (i.e. the minimum value of valid time). The symbol "∞" is used to denote the special values UNTIL_CHANGED in transaction time (which is used to timestamp a still current fact, namely data not updated yet), and FOREVER in valid time (i.e. the maximum value of valid time). The symbol "now" denotes the current transaction time and the present valid time during the same transaction. The full temporal domains (universes) of transaction time and valid time and the bitemporal one are thus:

$$\mathcal{U}_t = \{0 \mathinner{..} \infty\}_t \qquad \mathcal{U}_v = \{0 \mathinner{..} \infty\}_v \qquad \mathcal{U}_b = \{0 \mathinner{..} \infty\}_t \times \{0 \mathinner{..} \infty\}_v$$

If r is a tuple in a temporal relation (data table or catalogue), then $\mathcal{T}(r)$ denotes its timestamp. To explicitly show the temporal format of the timestamp, a subscript may be added; for instance $\mathcal{T}_b(r)$ denotes the timestamp of the bitemporal tuple r.

- A transaction-time tuple is *current* if now $\in \mathcal{T}_t(r)$.
- A bitemporal tuple is *current* if $\exists t \in \mathcal{U}_v \ : \ (\text{now}, t) \in \mathcal{T}_b(r)$.
- The symbol $M_{F_1F_2}$ indicates the temporal conversion function which translates a table from the temporal format F_1 into the temporal format F_2 [2].

2.2. Design Options

The design options for schema versioning presented in this paper can be classified according to the following taxonomy:

1. Temporal Dimensions Adopted
   (a) Transaction-time schema versioning
   (b) Valid-time schema versioning
   (c) Bitemporal schema versioning
2. Management of Extensional Data
   (a) Single-pool solution
   (b) Multi-pool solution
3. Interaction between Intensional and Extensional Versioning
   (a) Synchronous management
   (b) Asynchronous management

Such options are conceptually orthogonal even if their combinations have to be constrained in some cases, as diffusely discussed in the rest of the paper.

The motivations for the design options are introduced in the following.

2.3. Temporal Dimensions Adopted

All the schema versioning proposals in current literature concern the maintainance of schema versions along ***transaction time*** [11, 12, 13]. Transaction-time schema versioning is sufficient to correctly deal with any ***on-time*** schema change, that is schema changes effective when applied, or with applications for which the exact time of execution of a schema change is not crucial (this seems to be the case of most CAD/CAM applications), and it is not required in ***schema evolution***.

On the other hand, ***valid-time*** schema versioning is made necessary by database applications requiring retro- or pro-active schema changes. Retroactive changes are quite common in databases, concerning both extensional and intensional data. As valid-time databases have been introduced in order to accommodate retroactive changes of extensional data, valid-time schema versioning is necessary to permit also retroactive changes of intensional data. The idea of maintaining schema and data versions along more than a single time line was first introduced, for the same reasons, by Ariav in [1]. For instance, retroactive changes of intensional data can be enforced by changes in laws with retroactive effects (e.g. it could have been stated to add social security numbers to employee records on 3/1/1996 but effective from 1/1/1996) or, even more likely, they can be a consequence of *deferred updates* (e.g. the addition of social security numbers can be stated and effective from now, but the corresponding schema change will be applied to the database only next month, and thus it must be enforced retroactively). Let us consider in detail the first case (retroactive change) with an example.

Example 2 We assume to start from the state:

SVa [1/1/90 - ∞]
`EMPLOYEE(NAME,CITY)`

NAME	CITY
Brown	London
Rossi	Rome

The following schema and data changes:

```
ALTER TABLE EMPLOYEE                                         S14
   ADD COLUMN SSN CHAR(12) ;
UPDATE EMPLOYEE                                              S15
   SET SSN='123-Y44-579A'
     WHERE Name='Brown' ;
UPDATE EMPLOYEE                                              S16
   SET SSN='123-X55-468B'
     WHERE Name='Rossi'
```

are applied on 3/1/1996.

If transaction-time schema versioning is adopted, the above transaction, executed on 3/1/1996, can only be effective from the moment it is executed (on-time transaction), yielding a final state which can be represented as follows:

SVa [1/1/90 - 2/29/96]
`EMPLOYEE(NAME,CITY)`

NAME	CITY
Brown	London
Rossi	Rome

SVb [3/1/96 - ∞]
`EMPLOYEE(NAME,CITY,SSN)`

NAME	CITY	SSN
Brown	London	123-Y44-579A
Rossi	Rome	123-X55-468B

In this case, a query using the schema as of December 1995 (i.e. SVa) returns the same data as before the schema change, that is without the attribute SSN. However, also a query using the schema as of January or February 1996 returns the same data, as it still uses SVa. Only a query using the schema as of March 1996 or later (i.e. SVb) returns employee data with the SSN, conforming to the new rules for employee records. Therefore, new applications requiring the SSN cannot work on the database as of January or February 1996, even if they would be required to do so.

On the contrary, if valid-time schema versioning is adopted, the above transaction could be explicitly made *retroactive*, in particular effective from 1/1/1996 even if effected on 3/1/1996. The final state can be represented as follows:

SVa [1/1/90 - 12/31/95]

`EMPLOYEE(NAME,CITY)`

NAME	CITY
Brown	London
Rossi	Rome

SVb [1/1/96 - ∞]

`EMPLOYEE(NAME,CITY,SSN)`

NAME	CITY	SSN
Brown	London	123-Y44-579A
Rossi	Rome	123-X55-468B

In this new case, a query using the schema as of December 1995 (i.e. SVa) returns the same data as before the schema change, but a query using the schema as of January 1996 or later (i.e. SVb) correctly returns employee data with the SSN. Therefore, new applications requiring the SSN column can work on the database also as of January or February 1996. □

As far as proactive changes are concerned, at an early stage of research in temporal databases, also the necessity of future versions of stored extensional data was debated at length. In the end, there was agreement on its necessity in order to allow the use of a temporal database for planning activities. However, when the planning activity involves not only the reality represented in the database (extensional data) but also the database structure and its representational ability (intensional data), future schema versions in addition to future data versions are needed. Future schema versions, produced by proactive schema changes, are only possible with valid-time schema versioning. One example of a planning activity involving intensional data is proactive database design in *what-if* analysis, during development or maintenance of database applications. While solutions based on non-temporal concurrent views (e.g. ***hypothetical databases*** [20]) may provide partial support for that, an explicit management of time and of versions in time, which is often crucial for planning, is only possible with valid-time schema versioning.

Moreover, in transaction-time schema versioning, only the last inserted version is subject to changes: all the other schema versions and the corresponding data are archived and thus no longer updatable. Valid-time schema versioning allows the correction of design errors in the past and the reuse of past schema versions, since any stored schema version can be updated and assigned a new validity, leading to an improved flexibility and economy in the process of database maintenance.

Advanced database applications, with enhanced auditing requirements, may also need bitemporal schema versioning, when not only retro- and pro-active schema changes must be managed, but it is also necessary to keep track of them in the database. As happens with extensional updates in bitemporal databases, this is only possible by means of schema versioning along both time dimensions. Although valid-time and bitemporal schema versioning were not included in TSQL2 design, their motivations have been acknowledged in [15]. In the example of valid-time schema versioning above, if an application producing a report on employees is run in February 1996 (before the schema change is actually effected), it obviously produces a report without employee social security numbers. If the personnel department director wondered, for instance in April 1996, why the February report does not contain SSNs, while the database valid in February does contain them, there would be no answer derivable from the database itself (there is no way of memorizing that the schema change was retroactive). No responsibility for SSN data missing from the report can be ascribed. On the other hand, if bitemporal schema versioning is adopted, the same question can easily be answered. In April 1996, employee schema and data valid in February 1996 can be retrieved from the current database or from the database rolled back (along transaction time!) to February 1996. In the former case the SSN column would be missing and in the latter it would be present in employee data. Therefore, the responsibility for the incomplete report printed in February can now easily be ascribed to the retroactive schema change. Also catalogues can be queried (intensional query) to exactly know when the retroactive change was effected. Such information could be critical, for instance, for auditing companies involved in insurance and legal disputes.

In this paper we consider the distinct functionalities of schema versioning along valid time, transaction time and along both time dimensions. The operations to be performed at the intensional and extensional level when a schema change occurs will be described in detail in Section 3.

2.4. Management of Extensional Data

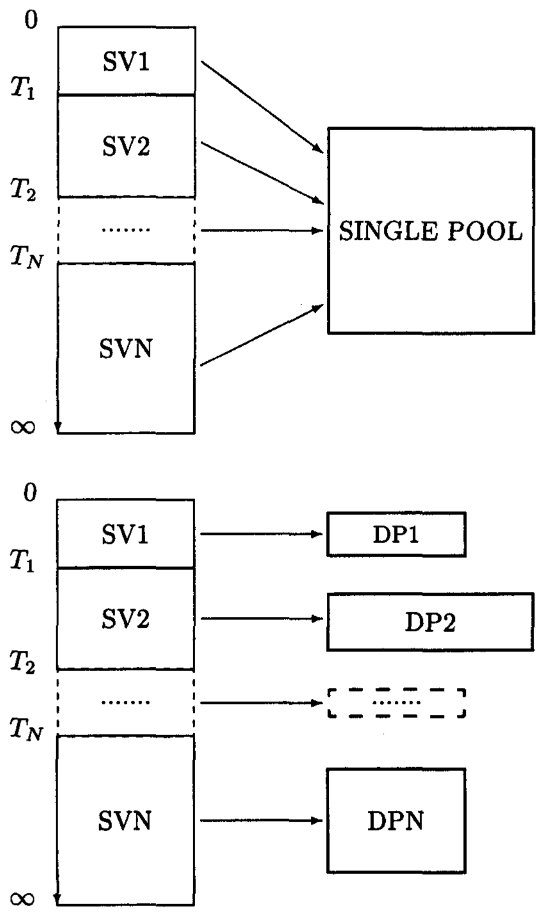


Fig. 1: Single-Pool and Multi-Pool Schema Versioning.

As far as the extensional aspects of schema versioning are concerned, we consider two distinct solutions and show how each of them works in response to schema changes, queries or updates. In the following, by the term ***data pool*** we denote a repository for extensional data. The solutions for organization and management of data pools are defined and discussed at a logical level, without going into physical design details. The two approaches we consider in this paper are (see Figure 1):

- **Single-Pool Solution** The data corresponding to all schema versions of a relation are clustered together in a single data pool.
- **Multi-Pool Solution** Distinct data pools are maintained for distinct schema versions of a relation [3, 16].

The single-pool solution is consistent with the proposals sketched in [15] in [23] for schema evolution and schema versioning along transaction time, and is based on a straightforward implementation of TSQL2 tables based on the ***completed schema***. As a matter of fact, the ***completed schema*** exactly corresponds to the structure of the single pool, so that data connected to any schema versions can be stored together. In the multi-pool solution, distinct data pools are structured according to the different corresponding schema versions.

Although the single-pool solution (and the TSQL2 *completed schema*) is considerable for its simplicity and apparent economy, it presents unavoidable drawbacks and inconsistencies due, for instance, to concurrent updates effected through different schema versions, as illustrated by the example that follows.

Example 3 Consider the relation EMPLOYEE with the two schema versions SV1 and SV2 resulting from the example Example 1 in Section 1, and suppose the two following transactions are concurrently run:

```
--- Transaction T1
SET SCHEMA DATE '1993-01-01' ;
UPDATE EMPLOYEE
   SET CITY='Liverpool'
     WHERE NAME='Brown'
```

and:

```
--- Transaction T2
SET SCHEMA DATE '1995-01-01' ;
UPDATE EMPLOYEE
   SET CITY='Edinburgh'
     WHERE NAME='Brown'
```

The former transaction changes Brown's city to Liverpool through the schema version SV1, the latter to Edinburgh through SV2. In the multi-pool solution, as expected, a subsequent access to Brown's city through SV1 will return Liverpool and through SV2 Edinburgh. In the single-pool solution, since both transactions affect a single copy of Brown's data stored in the only pool, the value stored after both transactions commit, depends on which transaction ends last. If it is T1 the value will be Liverpool, Edinburgh if T2. Such a unique value will be returned by subsequent retrievals either effected through SV1 or through SV2 (or through any other schema version). Therefore, the adoption of the single-pool violates to some extent (viz. with naïve concurrency control policies) the property of serializability of transactions. In the multi-pool indeed, the results of transactions T1 and T2 do not depend on their commit order, because independent pools are affected. □

Notice that the adoption of the multi-pool solution has some impact on the data model. Distinct data pools allow separate lines of evolutions of the same data due to update activity through different schema versions. As a matter of fact, in the example above, we have seemingly two versions of Brown's data after the executions of T1 and T2: the former with CITY equal to Liverpool associated to SV1 (and its timestamps), the latter with Edinburgh associated to SV2. However, these are not "normal" data versions —as schema version timestamps do not timestamp the connected extensional data too— but *metaversions*, in the same sense that a database schema is a metarepresentation of real world entities. Schema versions are descriptions of the real world representation adopted in the database; they can be past, present or future, real, hypothetical or even virtual, regardless of the temporal nature of extant data. In the example, extant data still preserve their snapshot status, although there are two versions of Brown's data which can be retrieved through schema versions relevant at different dates! Therefore, real world entities can be versioned -at extensional level- via tuple timestamping but also *metaversioned* -at intensional level- via schema version timestamping. In any case, the two levels should not be confused, particularly in the most complex temporal framework (multi-pool schema versioning).

Further semantic differences between the two solutions will be discussed later in the paper.

2.5. Interaction between Intensional and Extensional Versioning

In all the examples we have mentioned so far, plain *snapshot* data were considered, as it was sufficient to introduce the previous schema versioning solutions. However, specific problems and new options arise when we consider schema versioning in the context of a database storing temporal data, since we have to deal with the interaction between intensional and extensional versioning.

This interaction leads us to the last degree of freedom in data management we propose in this work as design option for database schema versioning. It takes place when extensional and intensional data are versioned along the same temporal dimension(s), and can be formalized in the following distinct policies:

- **Synchronous Management.** Temporal data are always stored, retrieved and updated through the schema version having the same temporal pertinence[†] of data, along the common temporal dimension(s).
- **Asynchronous Management.** Temporal data can be retrieved and updated through any schema version, whose pertinence is independent of the pertinence of data, also along common temporal dimension(s).

Let us consider in general the possible relationships between the time dimension(s) chosen for schema versioning and for data versioning at design time. Even in a "classical" environment (e.g. TSQL2), we may adopt transaction-time schema versioning in the presence of snapshot, valid-, transaction-time or bitemporal tables. In our approach, since also valid-time and bitemporal schema versioning can be used, the range of design choices is extended to every combination of possible extensional/intensional versionings (no versioning, along valid- or transaction-time, bitemporal versioning). For instance, in a database where it is important to allow and keep track of retroactive schema changes but only on-time data changes are considered, bitemporal schema versioning and transaction-time data versioning can be adopted. In a database where retro- or pro-active schema changes are not so important but retroactive data changes are (and there is no need for keeping track of them), the right choice is transaction-time schema versioning and valid-time data versioning, etc. Moreover, in a multitemporal system, the choice of the most suitable schema versioning option is unique for a database and affects all its tables, but the individual temporal types of the tables may differ to fit individual requirements on extensional data changes. Thus, for example, in a database supporting valid-time schema versioning we can create snapshot tables as well as bitemporal tables.

Due to the orthogonality of temporal dimensions, there cannot be any kind of semantic interaction between intensional and extensional data versioning along different temporal dimensions, that is transaction-time schema versioning does not affect valid-time data versioning and valid-time schema versioning does not affect transaction-time data versioning. However, the adoption of common temporal dimensions for intensional and extensional versioning gives rise to some semantic interactions. Consider for instance the following case, which is based on a perfectly "legal" use of TSQL2. Let us assume that our TSQL2 system supports (transaction-time) schema versioning and that `EMPLOYEE` is a transaction-time relation. Hence, we may write the following code:

```
SET SCHEMA DATE '1995-01-01' ;
SELECT * FROM EMPLOYEE
   WHERE TRANSACTION(EMPLOYEE) CONTAINS DATE '1996-01-01'
```

The `SET SCHEMA` statement above is aimed to select the schema version as of the beginning of 1995, thus, it rolls back the intensional database to 1/1/1995. The `WHERE` clause of the query is aimed to select the data version as of the beginning of 1996, thus, it rolls back the extensional database to 1/1/1996. Therefore, the result of the execution of the two statements together is to bring back the database to an *inconsistent* state, part of it dating to 1995 and part to 1996. The correctness of the above code would strain the semantics of transaction time that is the temporal dimension along which a database can be rolled back to a past state of its life. Clearly, such a state must be consistent, and the situation above-described should be avoided.

In conclusion, for consistency reasons, transaction-time schema and data versions must be managed in a *synchronous* way, that is intensional and extensional data must be stored and accessed for synchronized values of their timestamps. However, a similar restriction does not hold in any way for valid time, owing to its different semantics. Therefore, the management of intensional and extensional data:

[†]The term "temporal pertinence" is used here as a synonym of *validity*, which can be referred to any temporal dimension. It avoids the use of awkward expressions like valid-time validity, transaction-time validity or bitemporal validity.

- is always *asynchronous* along *orthogonal* time dimensions;
- is always *synchronous* along *transaction time*;
- can be *synchronous* or *asynchronous* along *valid time*.

The requirements and design choices also affect subsequent database administration and usage. For instance, if our application has also strong auditing requirements (every retro- and pro-active change must be logged), bitemporal versioning must be used both for schema and data versioning. Then, the management along transaction time is synchronous; this is, besides being correct, also desirable: the whole database can be rolled back for auditing purposes. On the other hand, application requirements on asynchronous management can be satisfied by the presence of valid time completely.

Furthermore, it can be noticed that synchronous management implies *synchronous versioning*, where the temporal pertinence of a schema version must include the temporal pertinence of the corresponding data along the common temporal dimension(s). On the contrary, asynchronous management gives rise to *asynchronous versioning*, that is the temporal pertinence of a schema version and the temporal pertinence of the corresponding data are completely independent.

Example 4 As an example of the importance of asynchronous management, consider the following case. We start from the database state:

SVα [1/1/90 - ∞]

`EMPLOYEE(NAME,SALARY|T)`

NAME	SALARY	T
Brown	1000$	{1/1/92..12/31/94}
Brown	1500$	{1/1/95..∞}

composed of a valid-time relation `EMPLOYEE` ("T" is the valid-time attribute) with exactly one schema version SVα (in valid-time schema versioning) and two tuples representing Brown's salary history. We assume then a schema change adding the column `DEPT` to `EMPLOYEE`, valid from 1/1/1994 (the exact syntax for such schema change cannot be given in TSQL2, which must be extended as shown in Section 3). We also assume that the same transaction, by means of the newly created schema version, stores the fact that employee Brown has always worked for the Sales department. If synchronous management is adopted, then the effect of the change can be represented as:

SVα [1/1/90 - 12/31/93]

`EMPLOYEE(NAME,SALARY|T)`

NAME	SALARY	T
Brown	1000$	{1/1/92..12/31/93}

SVβ [1/1/94 - ∞]

`EMPLOYEE(NAME,SALARY,DEPT|T)`

NAME	SALARY	DEPT	T
Brown	1000$	Sales	{1/1/94..12/31/94}
Brown	1500$	Sales	{1/1/95..∞}

We can observe that it has not been possible to store the department information for the period 1992-1993, since Brown's data concerning that period are connected to the old schema version without the `DEPT` attribute. Furthermore, special applications using both schema versions must be written anew even to reconstruct the complete history, from 1990 to now, of old data (without the `DEPT` attribute), because it is "split" in synchronicity with the two schema versions.

If asynchronous management is in fact adopted, the effect of the change can be represented as:

SVα [1/1/90 - 12/31/93]

`EMPLOYEE(NAME,SALARY|T)`

NAME	SALARY	T
Brown	1000$	{1/1/92..12/31/94}
Brown	1500$	{1/1/95..∞}

SVβ [1/1/94 - ∞]

`EMPLOYEE(NAME,SALARY,DEPT|T)`

NAME	SALARY	DEPT	T
Brown	1000$	Sales	{1/1/92..12/31/94}
Brown	1500$	Sales	{1/1/95..∞}

In this case, the complete history of employee data is maintained for each schema version. Old applications may still work on old data producing the same results as before, and new applications can be written to work on new data including the DEPT information. □

However, the exact results of schema and data changes in the presence of (a)synchronous management is also affected by the implementation of tables via single- or multi-pools. Such issue will be further illustrated in Section 3 and discussed in Section 5.

Finally, it can be noticed that the concept of asynchronous management extends the functionalities of previous proposals. In the schema versioning approaches proposed for object-oriented systems, data objects are normally only visible (and updatable) through the schema version they belong to, that is the one in which they were created [9, 11]. More flexibility has then been added by proposals in the temporal database field, starting from [1]. In [13], Roddick recently introduced a distinction between *partial* and *full* schema versioning. In the former approach, data can be accessed via any schema versions but only updated through one designated (normally the current) schema version. In the latter, any schema version can be used for both data access and modification. Our approach reconsiders such concepts in the perspective of (a)synchronous management, constraining with consistency the full schema versioning in transaction time and leaving other choices as design options.

3. MANAGEMENT OF SCHEMA VERSIONING

In this section we consider transaction-time, valid-time and bitemporal schema versioning, and describe, at a logical level, mechanisms for their support and management.

System catalogues are assumed to be in a slightly simplified format with respect to the SQL-92 standard [5]. We also assume system-defined identifiers to be used for time-invariant identification of database objects (namely tables, columns, domains, etc.).

The TABLES catalogue stores information on relations in a database. The COLUMNS catalogue stores detailed information on relation columns. The TABLES and the COLUMNS catalogues together define the relation schemata:

```
TABLES (TABLE_ID, TABLE_NAME, TABLE_TYPE)
```

```
COLUMNS (TABLE_ID, COLUMN_NAME, DOMAIN_ID ...)
```

The DOMAIN_ID can be seen as a pointer to another catalogue (e.g. DOMAINS), where detailed information on column domains can be found. Further attributes (e.g. ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE) may usually be present. For instance, the schema SV1 of the relation EMPLOYEE in Example 1 can be stored in SQL-92 catalogues as:

TABLE_ID	TABLE_NAME	TABLE_TYPE
T1	EMPLOYEE	BASE TABLE

TABLE_ID	COLUMN_NAME	DOMAIN_ID
T1	NAME	D1
T1	ADDRESS	D2
T1	CITY	D3

In TSQL2, such catalogues are enlarged as follows:

```
TABLES (TABLE_ID, TABLE_NAME, TABLE_TYPE, VALID_TIME, TRANSACTION_TIME | T)
```

```
COLUMNS (TABLE_ID, COLUMN_NAME, DOMAIN_ID ... | T)
```

The attributes VALID_TIME (whose possible values are STATE, EVENT or NONE) and TRANSACTION_TIME (whose possible values are STATE or NONE) have been added to define the temporal type of a table. Furthermore, a transaction-time attribute (T) has been added to both catalogues to define and manage schema versions (they actually become transaction-time relations). The pair formed by a TABLE_ID and a timestamp uniquely identifies the information concerning a schema version of the corresponding table in the catalogues.

For instance, the schema versions SVα and SVβ of relation EMPLOYEE used in Example 4 can be stored in TSQL2 catalogues as shown in the following:

TABLE_ID	TABLE_NAME	TABLE_TYPE	VALID_TIME	TRANSACTION_TIME	T
T2	EMPLOYEE	BASE TABLE	STATE	NONE	{1/1/90..∞}

TABLE_ID	COLUMN_NAME	DOMAIN_ID	T
T2	NAME	D1	{1/1/90..∞}
T2	SALARY	D4	{1/1/90..∞}
T2	DEPT	D5	{1/1/94..∞}

Notice that schema versions are "distributed" among several catalogue tuples, and catalogue tuples may in turn represent pieces of different schema versions. For instance, the first two tuples in catalogue COLUMNS represent pieces of both schema versions, while the last catalogue tuple only belongs to SVβ.

In our proposal, catalogues may be time-stamped either with transaction-time, valid-time, or bitemporal elements, according to the temporal dimensions chosen for schema versioning. For the sake of simplicity, unlike TSQL2, we do not consider (valid-time) *event tables* [6]. Therefore, the two catalogue attributes VALID_TIME and TRANSACTION_TIME can be replaced by a single attribute TEMPORAL_TYPE, whose possible values are *transaction*, *valid* or *bitemporal*. However, event tables and corresponding schema changes (e.g. turning an event table into a state table and *vice versa*) could easily be accommodated in our model via suitable temporal conversion functions. Moreover, the definition of catalogues is different if extensional data management is based on the single- or on the multi-pool, as detailed in the following.

Multi-Pool Catalogues

In this case, the catalogues can be defined as follows:

```
TABLES (TABLE_ID, POOL_ID, TABLE_NAME, TABLE_TYPE, TEMPORAL_TYPE | T)
```

```
COLUMNS (TABLE_ID, COLUMN_NAME, DOMAIN_ID ... | T)
```

Each schema version (identified by a TABLE_ID and a timestamp T) is associated to a specific data pool (identified by a POOL_ID) in the catalogue TABLES. The detailed structure of individual schema versions is then defined by the catalogue COLUMNS.

Single-Pool Catalogues

In this case, two pairs of catalogues can be used. The first pair:

```
TABLES (TABLE_ID, TABLE_NAME, TABLE_TYPE, TEMPORAL_TYPE | T)
```

```
COLUMNS (TABLE_ID, COLUMN_NAME, DOMAIN_ID ... | T)
```

defines the structure of individual schema versions of the relations. The second pair:

```
POOLS (TABLE_ID, POOL_ID, TEMPORAL_TYPE)
```

```
POOL_COLUMNS (TABLE_ID, COLUMN_NAME, DOMAIN_ID ...)
```

defines the current structure of each single-pool (*completed schema*). While the first pair is made up of temporal catalogues, the second pair is made up of *snapshot* catalogues.

The POOLS and POOL_COLUMNS catalogues define the storage structure of data in the single-pool associated to a given relation, whereas the TABLES and COLUMNS catalogues define the views which map the single-pool data into each schema version of the relation. The TEMPORAL_TYPE attribute in catalogue POOLS represents the union of the temporal dimensions associated to every schema version of the relation.

In the following three subsections on transaction-time, valid-time and bitemporal schema versioning, we present the schema and data management techniques necessary to implement our schema versioning solutions with all the design options described in Section 2. Their actions on intensional and extensional data in response to a schema change are illustrated by means of symbolic examples (dealing with schema versions and pools) and formalized by means of low-level algorithms (dealing with tuples in catalogues and data pools). In all the examples, we also consider the same sequence of schema changes in order to show their different effects in the different cases.

Example 5 The sample sequence consists of the following transactions:

- TR_1 - schema change - on-time transaction - committed on 1983/1/1 - effective from 1983/1/1 to 1985/12/31: creation of the *transaction-time* table EMPLOYEE with columns NAME, ADDRESS, CITY;
- TR_2 - data change - on-time transaction - committed on 1984/1/1: insertion of the tuple "(Blanc, BD St Michel 87, Paris)" into table EMPLOYEE;
- TR_3 - schema change - retroactive transaction - committed on 1985/1/1 - effective from 1983/1/1 to 1985/12/31: elimination of column ADDRESS from the table EMPLOYEE (the transaction modifies the previous schema version without altering its validity);
- TR_4 - data change - on-time transaction - committed on 1986/1/1: update of the CITY attribute (set to Bruxelles) in Blanc's data stored in the table EMPLOYEE;
- TR_5 - schema change - proactive transaction - committed on 1987/1/1 - effective from 1988/1/1: addition of column PHONE to the table EMPLOYEE.

Notice that extensional changes, namely transactions TR_2 and TR_4, are obviously effected through (mandatory) on-time transactions, as EMPLOYEE is a transaction-time table. Intensional changes that are not defined on-time in the above list, namely transactions TR_3 and TR_5, can actually be effected as retro-/pro-active schema changes only if valid time is a time dimension included in schema versioning. If transaction-time schema versioning is adopted, they can only be executed as on-time transactions (effective from their commit time), thus violating the correct semantics of their specifications in the description above. For instance, in transaction-time schema versioning, transaction TR_3 *should* be effective from 1983 to 1986 according to the specification, but it *will* actually be effective from 1985 on, due to the limited capabilities of the system. □

Example 5 will be continued in the subsections which follow with the alternatives from Example 5.1 to Example 5.8.

3.1. Transaction-Time Schema Versioning

In transaction-time schema versioning, a schema change always concerns the *current* schema version, owing to the definition of transaction time. Therefore, no SET SCHEMA statement can be used before a schema change to select the schema version to be modified. The default transaction-time value set for schema selection by the last executed `SET SCHEMA` statement is ignored. Moreover, no time pertinence for the new schema version can be supplied by the user: the implicit transaction-time pertinence of a schema change is always `[NOW - UNTIL_CHANGED]`. The TSQL2 `CREATE` and `ALTER` statements can thus be used without modifications. Moreover, if extensional data also contain transaction time, the management can only be synchronous.

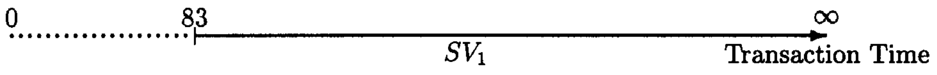
3.1.1. *Intensional Management*

Transaction time schema versioning requires the simplest management of intensional data in response to schema changes. Only current schema versions can be affected by a schema change, need to be archived and replaced by a new current schema version, obtained by the old one by applying the required changes.

Example 5.1 Consider the sequence of schema changes (Example 5) introduced in Section 5. Obviously, with transaction-time schema versioning we cannot effect retro- or pro-active changes. Therefore, transactions TR_1, TR_3 and TR_5 will be effective from the time they are committed. Transaction TR_1 can be expressed in TSQL2 as follows:

```
CREATE TABLE EMPLOYEE
   ( NAME CHAR(20),
     ADDRESS CHAR(30),
     CITY CHAR(20) )
        AS TRANSACTION
```
TR_1

Its effects on system catalogues can be represented as follows:



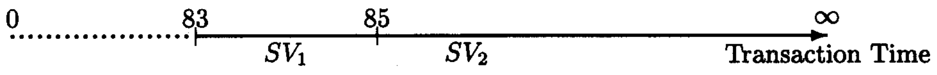
SV_1 [1/1/83 - ∞]

EMPLOYEE(NAME,ADDRESS,CITY)

The figure displays the positioning on the transaction-time axis and the symbolic content of the newly created schema version SV_1. Transaction TR_3 can be written as:

```
ALTER TABLE EMPLOYEE
   DROP COLUMN ADDRESS
```
TR_3

It affects the current schema version SV_1, archives it by limiting its time pertinence to 12/31/84 (cf. TR_3 is executed on 1/1/85, and we assume for simplicity a time granularity of a day), and replaces it by a new schema version SV_2 without the `ADDRESS` attribute. The result can be displayed as:



SV_1 [1/1/83 - 12/31/84]

EMPLOYEE(NAME,ADDRESS,CITY)

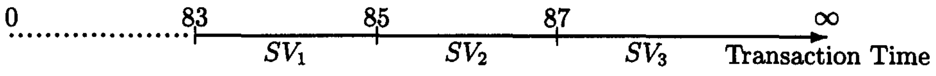
SV_2 [1/1/85 - ∞]

EMPLOYEE(NAME,CITY)

In a similar way, the last transaction (TR_5), executed on 1/1/87 and expressed as:

```
ALTER TABLE EMPLOYEE
   ADD COLUMN PHONE CHAR(20)
```
TR_5

produces the final state:



SV_1 [1/1/83 - 12/31/84]

`EMPLOYEE(NAME,ADDRESS,CITY)`

SV_2 [1/1/85 - 12/31/86]

`EMPLOYEE(NAME,CITY)`

SV_3 [1/1/87 - ∞]

`EMPLOYEE(NAME,CITY,PHONE)`

□

We introduce now the algorithm required for catalogue management. When, after a schema change, a new (current) schema version is produced and the previous one is archived, this is accomplished through an update of the catalogues TABLES and COLUMNS involving the current tuples concerning the table modified. In the case of single-pool, also the catalogues POOLS and POOL_COLUMNS. need to be modified, if the schema change enlarges the *completed schema*. Their management is effected as follows:

Algorithm 1

```
• Let $SV_n$ be the current schema version to be modified; $SV_n$ is determined
  as the set of tuples $r$ concerning the modified relation such that now $\in \mathcal{T}_t(r)$
  ($SV_n$ is empty if the schema change is a CREATE TABLE)

• Let $SV_{n+1}$ be the new schema version. $SV_{n+1}$ is obtained from $SV_n$ and from
  the schema change statement, by applying the required modifications
  ($SV_{n+1}$ is empty if the schema change is a DROP TABLE)

    – Creation of the new schema version
      for each tuple $r \in SV_{n+1}$, set
      $\mathcal{T}_t(r) := \{\text{now..}\infty\}_t$ and insert $r$ in the catalogues
    – Archiving of the old schema version
      for each tuple $r \in SV_n$, update $r$ in the catalogues by setting
      $\mathcal{T}_t(r) := \mathcal{T}_t(r) \setminus \{\text{now..}\infty\}_t$
```

If Single-Pool:

```
• Let $SV_n^*$ be the current completed schema before the change; $SV_n^*$ is determined
  as the set of tuples $r$ concerning the modified relation in the pool catalogues

• Let $SV_{n+1}^*$ be the new completed schema; $SV_{n+1}^*$ is obtained from $SV_n^*$ and from the
  schema change only considering the modifications which enlarge the schema

    – Insertion of new pool information
      for each tuple $r \in SV_{n+1}^* \setminus SV_n^*$, store it in the pool catalogues
```

Notice that SV_n and SV_{n+1} are always subsets of tuples from the catalogues TABLES and COLUMNS, whereas, in the single-pool, SV_n^* and SV_{n+1}^* are subsets of tuples from the catalogues POOLS and POOL_COLUMNS.

As far as extensional management is concerned, let us now describe the (synchronous) single- and multi-pool.

3.1.2. Extensional Management: Single-Pool Solution (Synchronous)

A single-pool consists of a repository where all the extensional data, concerning every schema version, of a given relation are stored together. The single-pool is formatted according to the *completed schema* of the relation. The *completed schema* includes all the attributes incrementally defined by successive schema changes. If a schema change is destructive (namely the drop of an attribute, the drop of a temporal dimension or the restriction of a domain) the change does not affect the completed schema and, thus, the single-pool. It only affects intensional data.

If the change of a domain produces a new domain incompatible with the old one (e.g. when changing an attribute CODE from numeric to alphabetic), two attributes must be maintained in the single-pool, with the same name as seen by the user, but corresponding to different domains and belonging to different schema versions as recorded in the catalogues. Since the change of temporal format is also allowed, if a schema change adds new temporal dimensions, the whole data pool must be converted to the enlarged temporal format, using the temporal conversion maps defined in [2], as already shown in [3]. The conversion of the single-pool to the current structure of the *completed schema* requires the introduction of null values, unless a default value is specified. For instance, if an attribute is added, the data of every schema version acquire that attribute which is initialized with null values[†]. However, null values introduced in current tuples are expected to be replaced by significant values by upcoming updates, whereas null values introduced to fill added columns in archived tuples cannot be changed and, thus, represent storage space waste.

From a practical viewpoint, with this schema versioning solution, if the schema change enlarges the overall single-pool format, all the tuples of the current data pool must be converted to the enlarged format. Moreover, if the change adds a missing temporal dimension to the pool, a suitable temporal conversion function must also be applied. In general, if F_n and F_{n+1} denote the temporal format of the single-pool before and after the schema change, respectively, then F_{n+1} can be obtained from F_n and from the schema change according to Table 1. The temporal conversion function $M_{F_n F_{n+1}}$ can then be applied to convert the temporal format of the data pool.

	Added dimension		Dropped dimension	
Previous format	v	t	v	t
s	v	t	N/A	N/A
v	N/A	b	v	N/A
t	b	N/A	N/A	t
b	N/A	N/A	b	b

Table 1: The New Temporal Format F_{n+1} in the Single-Pool.

In Table 1, the symbols s, v, t and b stand for snapshot, valid-, transaction-time and bitemporal, respectively. The "N/A" (*not available*) entry reminds the reader that it is not possible to drop a temporal dimension not already present in a table, nor to add one already present. In those cases, the **ALTER TABLE** statement execution raises an error exception ending up the transaction abort.

Example 5.2 Considering Example 5 of Section 5, the effects of the transactions from TR_1 to TR_5 can be represented as follows. Initially, TR_1 creates an empty data pool with the structure:

DP_1

NAME	ADDRESS	CITY	T
(empty)			

Transaction TR_2, executed on 1/1/1984 can be expressed as:

```
INSERT INTO EMPLOYEE                                          TR2
   VALUES('Blanc','BD St Michel 87','Paris')
```

[†]The same use of null values is expected in the plain SQL-92 schema change mechanism and, thus, no special semantics actually needs to be defined. However, a more sophisticated discussion on the meaning of null values in the presence of schema versioning can be found in [13].

and starts to populate DP_1 as follows:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..∞}

The schema change TR_3, which is a column drop, does not affect the single-pool structure and contents. The data change TR_4, executed on 1/1/86, can be expressed as:

```
UPDATE EMPLOYEE                                                    TR4
   SET City='Bruxelles'
     WHERE Name='Blanc'
```

and has the following effects on the data pool:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/85}
Blanc	*Null*	Bruxelles	{1/1/86..∞}

In the new tuple, the address value is null, because the statement TR_4 modifies the version of Blanc's data current as of 1/1/86, which do not have the ADDRESS column (the tuple to be modified is retrieved from the data pool through schema version SV_2).

Transaction TR_5 actually modifies the single pool, since it enlarges the *completed schema*, as follows:

DP_2

NAME	ADDRESS	CITY	PHONE	T
Blanc	BD St Michel 87	Paris	*Null*	{1/1/84..12/31/85}
Blanc	*Null*	Bruxelles	*Null*	{1/1/86..∞}

□

However, it should be noted that the single-pool modification does not affect past data and thus the semantics of transaction time is preserved. For example, the first tuple stored in DP_2 in a format formally containing attribute PHONE can only be retrieved through SV_1, which "hides" PHONE.

We formalize now the algorithms to be used for extensional data management. Let DP_n be the current single-pool, associated with the current *completed schema* version SV_n^*, before the schema change. The conversion of the single-pool can be described as the creation of a new single-pool DP_{n+1}, in which extensional data from DP_n are stored after conversion, followed by the deletion of DP_n. Obviously, if $SV_{n+1}^* = SV_n^*$ (e.g. when there are only column drops), there is no need to create a new pool identical to the old one. In this case the old pool DP_n is left untouched. Otherwise, if $SV_{n+1}^* \neq SV_n^*$, a new pool DP_{n+1} is created and the algorithm for the conversion consists of the following steps:

Algorithm 2

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}^*$
• For each tuple $r \in DP_n$, consider the conversion from $SV_n^*$ to $SV_{n+1}^*$
    – for added columns: add new attributes padded with nulls
    – for enlarged domains: convert attribute values to the enlarged format
    – for added time dimensions: apply the conversion map $M_{F_n F_{n+1}}$
    – store $r$ in $DP_{n+1}$
• Delete $DP_n$
• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}^*$)
```

3.1.3. Extensional Management: Multi-Pool Solution (Synchronous)

In this case, one separate data pool, even if archived, is maintained for each transaction-time schema version.

If the schema change involves the addition or elimination of a temporal dimension, the appropriate temporal conversion function must also be applied. If F_n denotes the temporal format of the old current data pool, the format of the new data pool, F_{n+1}, can be obtained from F_n and from the schema change according to Table 2. Notice that Table 2 differs from Table 1 only in the

	Added dimension		Dropped dimension	
Previous format	v	t	v	t
s	v	t	N/A	N/A
v	N/A	b	s	N/A
t	b	N/A	N/A	s
b	N/A	N/A	t	v

Table 2: The New Temporal Format F_{n+1} in the Multi-Pool

case of dropped dimensions. The temporal conversion function $M_{F_n F_{n+1}}$ can be applied to convert the temporal format of the new data pool.

Example 5.3 Considering Example 5 of Section 5, the effects of the transactions from TR_1 to TR_5 can be represented as follows. As in the single-pool case, TR_1 creates an empty data pool DP_1 which is initially populated by the insertion in transaction TR_2:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..∞}

The schema change TR_3 creates a new data pool DP_2 as follows:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/84}

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/85..∞}

Notice that extensional data are "split" along the time dimension into the two pools, due to the mandatory synchronous management (both data and schemata are versioned along transaction time). Transaction TR_4 only affects pool DP_2 (which is the current one) as follows:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/84}

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/85..12/31/85}
Blanc	Bruxelles	{1/1/86..∞}

Transaction TR_5 creates a third data pool:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/84}

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/85..12/31/85}
Blanc	Bruxelles	{1/1/86..12/31/86}

DP_3

NAME	CITY	PHONE	T
Blanc	Bruxelles	*Null*	{1/1/87..∞}

□

We introduce now the management algorithm. Let DP_n be the data pool associated with the current schema verion SV_n. The schema versioning requires the creation of a new data pool, DP_{n+1}, to be associated with the new schema version SV_{n+1}. Extensional data are copied from DP_n to DP_{n+1} after the required conversion.

Unlike the single-pool solution, DP_n must be retained after the schema change. However, since the management must also be synchronous, if data in the old and/or in the new data pool contain transaction time, their timestamps must accordingly be partitioned.

The pool management algorithm can be briefly described as follows.

Algorithm 3

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}$

• For each tuple $r \in DP_n$, consider the conversion from $SV_n$ to $SV_{n+1}$

    – for added columns: add new attributes padded with nulls
    – for enlarged domains: convert attribute values to the enlarged format
    – for dropped columns: eliminate the old attributes
    – for restricted domains: convert the attribute values to the restricted
      format
    – for changed time dimensions: apply the conversion map $M_{F_n F_{n+1}}$
    – store $r$ in $DP_{n+1}$

• For each tuple $r \in DP_n$

    – if $F_n = t$: set $\mathcal{T}_t(r) := \mathcal{T}_t(r) \setminus \{\text{now}, \infty\}_t$
    – if $F_n = b$: set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \setminus \{\text{now}, \infty\}_t \times \mathcal{U}_v$

• For each tuple $r \in DP_{n+1}$

    – if $F_{n+1} = t$: set $\mathcal{T}_t(r) := \mathcal{T}_t(r) \cap \{\text{now}, \infty\}_t$
      (if $\mathcal{T}_t(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_{n+1} = b$: set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \cap \{\text{now}, \infty\}_t \times \mathcal{U}_v$
      (if $\mathcal{T}_b(r) = \emptyset$ then delete $r$ from the data pool)

• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}$)
```

3.2. Valid-Time Schema Versioning

In valid-time schema versioning, a new VALID clause is introduced in the DDL statements in order to allow the user to specify the validity of the schema change, enabling retro- and pro-active changes. Moreover, we are not limited to modify the last created schema version as with transaction time. By means of a SET SCHEMA statement preceding the schema change, we may select any schema version for modification.

The solution proposed for valid-time schema versioning is described in the following. In Section 3.2.1 we describe the management of intensional data. In Section 3.2.2 and Section 3.2.3 we present the single- and the multi-pool solutions for the management of extensional data.

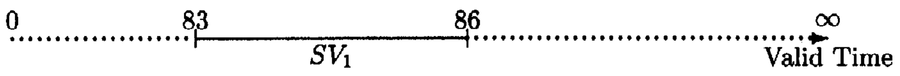
3.2.1. Intensional Management

With valid-time schema versioning, a schema change creates a new schema version by applying the desired changes to the schema version selected for modification via the SET SCHEMA statement. The new version is assigned the validity of the schema change. Previous schema versions completely overlapped by the change validity are deleted and previous schema versions which are only partially overlapped in validity by the schema change have their validity accordingly restricted. The semantics adopted for the modification of valid-time schema versions is consistent at intensional level with the semantics of the modification of extensional data versions defined for TSQL2 (UPDATE statement).

Example 5.4 Reconsider now the example sequence of schema changes (Example 5) introduced in Section 5. With valid-time schema versioning, retro- and pro-active changes can actually be effected. Transaction TR_1 can be expressed in TSQL2 as follows:

```
CREATE TABLE EMPLOYEE                                                    TR1
   ( NAME CHAR(20) NOT NULL PRIMARY KEY, ADDRESS CHAR(30), CITY CHAR(20) )
     AS TRANSACTION
        VALID PERIOD '[1983-01-01 - 1985-12-31]'
```

Its effects on system catalogues can be represented as follows:



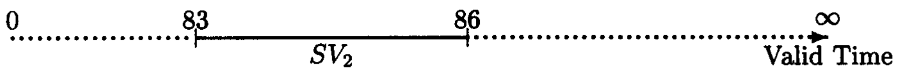
SV_1 [1/1/83 - 12/31/85]

EMPLOYEE(NAME,ADDRESS,CITY)

The schema change in TR_3 can then be expressed as:

```
SET SCHEMA DATE '1984-01-01' ;                                           TR3
ALTER TABLE EMPLOYEE
   DROP COLUMN ADDRESS
```

The SET SCHEMA statement ensures that the schema version to be selected for modification is the one valid on 1/1/84, that is SV_1. It can be noticed that the ALTER statement above does not contain a VALID clause, since in the example it does not alter the validity of the modified schema version (from the beginning of 1983 to the end of 1985).
The result is:



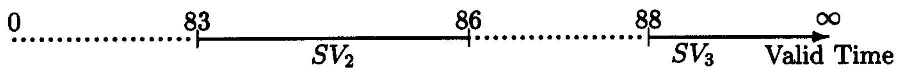
SV_2 [1/1/83 - 12/31/85]

EMPLOYEE(NAME,CITY)

Notice that the new schema version SV_2 completely substitutes the old one SV_1, since it has the same validity. Transaction TR_5 can be expressed as:

```
SET SCHEMA DATE '1984-01-01' ;                                           TR5
ALTER TABLE EMPLOYEE
   ADD COLUMN PHONE CHAR(20)
     VALID PERIOD '[1988-01-01 - forever]'
```

and produces the result:



SV_2 [1/1/83 - 12/31/85]

EMPLOYEE(NAME,CITY)

SV_3 [1/1/88 - ∞]

EMPLOYEE(NAME,CITY,PHONE)

□

When a schema change is applied, a new schema version is produced and several old schema versions may be deleted or have their timestamps restricted, in order not to overlap the validity of the new schema version. This is accomplished through an update of the catalogues containing the tuples concerning the table modified. Their management can be effected as described in the following algorithm. Notice that with valid-time schema versioning, in the case of single pool, the *completed schema* of a relation can also *shrink*. This happens after a schema change that drops a column or restricts a domain in all the schema versions currently defined for the relation.

Algorithm 4

```
• Let $SV_k$ be the schema version to be modified.  It is determined as the
  schema version currently in use for the relation to be modified
  (empty if the schema change is a CREATE TABLE)

• Let $\mathcal{T}_v(SC)$ be the validity of the schema change.
  If there is no explicit VALID clause in the schema change,
  then it equals the validity of the schema version to be modified,
  computed as $\mathcal{T}_v(SC) = \cap_{r \in SV_k} \mathcal{T}_v(r)$

• Let $SV_{n+1}$ be the new schema version to be inserted.
  $SV_{n+1}$ is obtained from $SV_k$ and from the schema change statement,
  by applying the required modifications
  (empty if the schema change is a DROP TABLE)

• Creation of the New Schema Version
  for each tuple $r \in SV_{n+1}$ set $\mathcal{T}_v(r) := \mathcal{T}_v(SC)$ and insert $r$ in the catalogues

• Affected Schema Versions
  for each schema version $SV_i$ concerning the modified relation such that
  $\mathcal{T}_v(SV_i) \cap \mathcal{T}_v(SC) \neq \emptyset$, then for each catalogue tuple $r \in SV_i$

    – update $r$ in the catalogues by setting $\mathcal{T}_v(r) := \mathcal{T}_v(r) \setminus \mathcal{T}_v(SC)$
      (if $\mathcal{T}_v(r) = \emptyset$ then delete $r$ from the catalogues)

If Single-Pool:

• Let $SV_n^*$ be the current completed schema before the change;
  $SV_n^*$ is determined as the set of tuples $r$ concerning the modified relation in
  the pool catalogues before the change.

• Let $SV_{n+1}^*$ be the new completed schema after the change;
  $SV_{n+1}^*$ is obtained from $SV_n^*$ and from the schema change,
  taking into account the modifications which enlarge the schema,
  or the modifications which restrict it if no resulting schema version still
  contains the dropped attribute or time dimension after the change

    – Elimination of Obsolete Pool Information
      for each tuple $r \in SV_n^* \setminus SV_{n+1}^*$, delete it from the pool catalogues
    – Insertion of New Pool Information
      for each tuple $r \in SV_{n+1}^* \setminus SV_n^*$, store it in the pool catalogues
```

As for transaction-time schema versioning, notice that SV_k and SV_{n+1} are always subsets of tuples from the catalogues TABLES and COLUMNS, whereas, in the single-pool, SV_n^* and SV_{n+1}^* are subsets of tuples from the catalogues POOLS and POOL_COLUMNS.

3.2.2. *Extensional Management: Single-Pool Solution (Synchronous and Asynchronous)*

As far as synchronous or asynchronous management of the single-pool is concerned, the distinction between them only appears in data manipulation, when individual schema versions are applied as different views of the stored data. Since the data corresponding to different schema versions are stored together in the single-pool (according to a common format), no additional operation is required for synchronous management with respect to asynchronous management in response to schema changes (tuples whose timestamps were cut according to schema version validities would merge again when stored in the same pool). Therefore, the algorithm for the single-pool in the asynchronous and synchronous case is the same.

However, the single-pool management in valid-time schema versioning has an important difference from its analogue in transaction-time schema versioning. The main difference is that (according to the shrinking of the *completed schema*) the structure of the single-pool may also shrink, when appropriate, to minimize the overall amount of stored data. With transaction time, no kind of logical deletion is physically destructive, thus, also schema "restrictions" (i.e. drop of an attribute or of a time dimension, restriction of a domain) do not affect the *completed schema*, which can only grow as a result. On the contrary, with valid time, schema restictions can effectively be destructive, if the restriction applies to *all* the resulting schema versions. For instance, suppose that an attribute is dropped and the new schema version (without the attribute) overlaps in validity all the old schema versions that contained the dropped attribute. Since there is no resulting schema version addressing the dropped attribute after the schema change, there is no point in mantaining the corresponding column in the single-pool, because no transaction could ever access such data.

Therefore, if the schema change involves the addition or elimination of a temporal dimension, the new temporal format of the single-pool is usually computed according to Table 1 but, if the change involves the final elimination of a temporal dimension from the *completed schema*, the new temporal format of the single-pool must be derived from Table 2.

Example 5.5 Coming back to Example 5 in Section 5, the effects of the transactions from TR_1 to TR_5 are described in the following. The effects of TR_1 and TR_2 can still be described by the single-pool state:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..∞}

The schema change TR_3, which is a column drop and completely overlaps in validity (i.e. deletes) the only previous schema version (SV_1) with that column, affects the single-pool structure and contents. As a matter of fact, column ADDRESS can be discarded, since there is no reference to it left in the catalogues. In other words, the ADDRESS attribute no longer belongs to the *completed schema* and no query could ever reference ADDRESS data in the future. Hence, the effects of TR_3 on the single pool are the following:

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/84..∞}

The data change TR_4 then has the following effects on the data pool:

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/84..12/31/85}
Blanc	Bruxelles	{1/1/86..∞}

Transaction TR_5 modifies the single pool, since it enlarges the *completed schema*, as follows:

DP_3

NAME	CITY	PHONE	T
Blanc	Paris	*Null*	{1/1/84..12/31/85}
Blanc	Bruxelles	*Null*	{1/1/86..∞}

□

The management algorithm can be formalized as follows. Let DP_n be the current single-pool before the schema change. The conversion of the single-pool can be described as the creation of a new single-pool DP_{n+1}, in which extensional data from DP_n are stored after conversion, followed by the deletion of DP_n. A brief description of the algorithm used for the conversion follows. The symbols SV_n^* and SV_{n+1}^* denote the old and the new *completed schema*, respectively.

Algorithm 5

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}^*$
• For each tuple $r \in DP_n$, consider the conversion from $SV_n^*$ to $SV_{n+1}^*$
    – for added columns: add new attributes padded with nulls
    – for enlarged domains: convert attribute values to the enlarged format
    – for dropped columns: eliminate the old attribute
    – for restricted domains: convert the attribute values to the restricted
      format
    – for added or dropped time dimensions: apply the conversion map $M_{F_n F_{n+1}}$
    – Store $r$ in $DP_{n+1}$
• Delete $DP_n$
• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}^*$)
```

3.2.3. Extensional Management: Multi-Pool Solution

The multi-pool solution requires the creation of as many data pools as the number of schema versions. Each schema version references its own data pool.

After a schema change, a new data pool is started according to the new schema version, and populated with the data coming from the pool corresponding to the modified schema version and updated according to the schema change. The data pools connected to completely overlapped (deleted) schema versions are also deleted. The data pools connected to only partially overlapped schema versions are left untouched in the case of asynchronous management, or have their timestamps restricted in the case of synchronous management.

Asynchronous Multi-Pool Solution

In this case, each data pool is formatted according to the corresponding schema version. Therefore, when a new data pool is started, the tuples are copied from the initial affected pool, according to the change applied to the previous schema (add/drop of an attribute, change of the temporal format of the table, etc.).

Example 5.6 Considering Example 5 of Section 5, the effects of the transactions from TR_1 to TR_5 can be represented as follows. As in the single-pool case, TR_1 creates an empty data pool DP_1 which is initially populated by the insertion of transaction TR_2:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..∞}

The schema change TR_3 creates a new data pool DP_2; since the new schema version SV_2 completely overlaps the modified one SV_1, which is deleted by the transaction, the data pool DP_1 associated to SV_1 can also be eliminated (it could not be accessed any longer). The result is:

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/84..∞}

Transaction TR_4 affects pool DP_2 as follows:

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/84..12/31/85}
Blanc	Bruxelles	{1/1/86..∞}

Transaction TR_5 creates a new data pool (DP_3):

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/84..12/31/85}
Blanc	Bruxelles	{1/1/86..∞}

DP_3

NAME	CITY	PHONE	T
Blanc	Paris	*Null*	{1/1/84..12/31/85}
Blanc	Bruxelles	*Null*	{1/1/86..∞}

□

A formal description of the algorithm for the support of the valid-time asynchronous multi-pool follows.

Let DP_k be the data pool associated with the modified schema version SV_k. The schema versioning requires the creation of a new data pool, DP_{n+1}, to be associated with the new schema version SV_{n+1}. Extensional data are copied from DP_k to DP_{n+1} Unlike the single-pool solution, also the old pool DP_k is retained after the schema change, unless SV_k has been deleted.

The pool management algorithm can briefly be described as follows.

Algorithm 6

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}$
• For each tuple $r \in DP_k$, consider the conversion from $SV_k$ to $SV_{n+1}$
    – for added columns: add new attributes padded with nulls
    – for enlarged domains: convert attribute values to the enlarged format
    – for dropped columns: eliminate the old attributes
    – for restricted domains: convert the attribute values to the restricted
      format
    – for changed time dimensions: apply the conversion map $M_{F_k F_{n+1}}$
    – Store $r$ in $DP_{n+1}$
• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}$)
• For each completely overlapped data pool $DP_i$ (associated to the completely
  overlapped schema version $SV_i$, such that $\mathcal{T}_v(SV_i) \subseteq \mathcal{T}_v(SC)$), delete $DP_i$
```

Synchronous Multi-Pool Solution

The only difference with respect to the previous case concerns the management of data pools connected to schema versions which are partially overlapped by the schema change, if their data contain valid time. In that case, the valid timestamp of their data must be restricted to be included in the corresponding schema version validity.

Since we assumed that table EMPLOYEE managed by transactions from TR_1 to TR_5 in Section 5 is a transaction-time table, our example is not applicable to the case described here to show differences with respect to the previous case. However, the behaviour of a synchronous management has been illustrated through transaction-time versioning and no substantial differences would be shown by a new example concerning valid time.

Let DP_k be the data pool associated with the current schema verion SV_k. The schema versioning requires the creation of a new data pool, DP_{n+1}, to be associated with the new schema version SV_{n+1}. Extensional data are copied from DP_k to DP_{n+1}, formatted according to the new schema version SV_{n+1}.

As in the previous case, DP_k must be retained after the schema change, unless completely overlapped by the change validity. However, since the management is synchronous, if data in the old and/or in the new data pool contain valid time, their timestamps must accordingly be restricted.

The pool management algorithm can briefly described as follows.

Algorithm 7

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}$
• For each tuple $r \in DP_k$, consider the conversion from $SV_k$ to $SV_{n+1}$
    – for added columns: add new attributes padded with nulls
    – for enlarged domains: convert attribute values to the enlarged format
    – for dropped columns: eliminate the old attributes
    – for restricted domains: convert the attribute values to the restricted
      format
    – for changed time dimensions: apply the conversion map $M_{F_k F_{n+1}}$
    – Store $r$ in $DP_{n+1}$
• For each completely overlapped data pool $DP_i$ (associated to the completely
  overlapped schema version $SV_i$, such that $\mathcal{T}_v(SV_i) \subseteq \mathcal{T}_v(SC)$), delete $DP_i$
• For each affected data pool $DP_i$ (associated to the affected schema version
  $SV_i$, such that $\mathcal{T}_v(SV_i) \cap \mathcal{T}_v(SC) \neq \emptyset$), then for each tuple $r \in DP_i$
    – if $F_i = v$: set $\mathcal{T}_v(r) := \mathcal{T}_v(r) \setminus \mathcal{T}_v(SC)$
      (if $\mathcal{T}_v(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_i = b$: set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \setminus \mathcal{U}_t \times \mathcal{T}_v(SC)$
      (if $\mathcal{T}_b(r) = \emptyset$ then delete $r$ from the data pool)
• For each tuple $r \in DP_{n+1}$
    – if $F_{n+1} = v$: set $\mathcal{T}_v(r) := \mathcal{T}_v(r) \cap \mathcal{T}_v(SC)$
      (if $\mathcal{T}_v(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_{n+1} = b$: set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \cap \mathcal{U}_t \times \mathcal{T}_v(SC)$
      (if $\mathcal{T}_b(r) = \emptyset$ then delete $r$ from the data pool)
• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}$)
```

3.3. Bitemporal Schema Versioning

Bitemporal schema versioning allows the maintenance along transaction time of all the valid-time schema versions created by successive schema changes. It allows retro- and pro-active schema changes and keeps track of them in the database. With bitemporal versioning, a schema change concerns current schema versions overlapping the schema change validity $\mathcal{T}_v(SC)$. Catalogues are defined and manipulated as bitemporal relations.

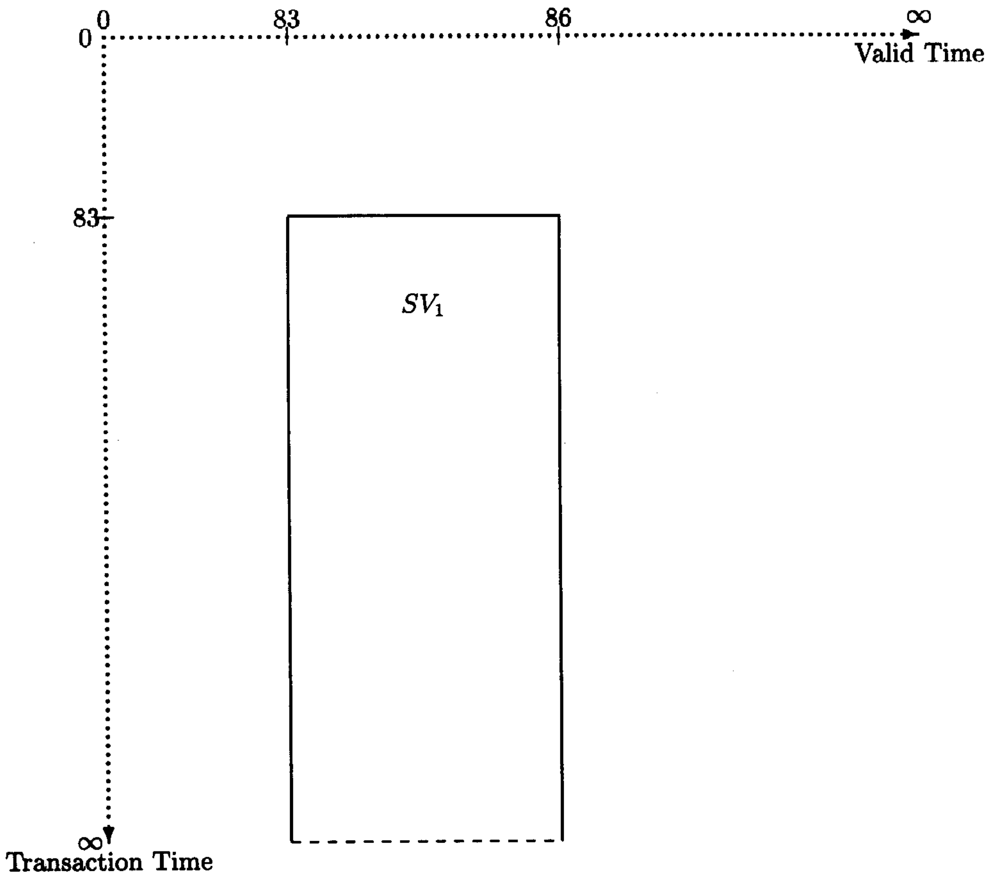
The syntax of the DDL is the same seen in valid-time schema versioning, with `CREATE` and `ALTER` statements augmented with the `VALID` clause to specify the schema change validity (obviously the transaction-time pertinence `[NOW - UNTIL_CHANGED]` is still understood for every schema change and cannot be managed by the user in any case). Therefore, exactly the same transactions can be effected in a valid-time or a bitemporal database. Moreover, their effects are exactly the same in the valid-time database and in the current historical state of the bitemporal database. However, the bitemporal database also maintains past historical states which enable the rollback functionality for auditing purposes both on intensional and extensional data. Moreover, as in valid-time schema versioning, the SET SCHEMA statement can be used to select schema versions to be modifed in schema changes by means of their validity. As in transaction-time, modified schema versions are always current with respect to transaction time (a default value of transaction time possibly set by the SET SCHEMA is still ignored).

In the following, we present the solution proposed for bitemporal schema versioning. In Section 3.3.1 we describe the management of intensional data. In Section 3.3.2 and Section 3.3.3 we present the single- and the multi-pool solutions for the management of extensional data.

3.3.1. Intensional Management

In bitemporal schema versioning, a schema change creates a new schema version by applying the desired changes to the schema version which is current and satisfy the validity condition expressed via the SET SCHEMA statement. The modified schema version is archived during the schema change management and is substituted by a new current version with the validity of the schema change. Notice that previous schema versions completely overlapped by the change validity are just archived, rather than being deleted as happens in valid-time schema versioning.

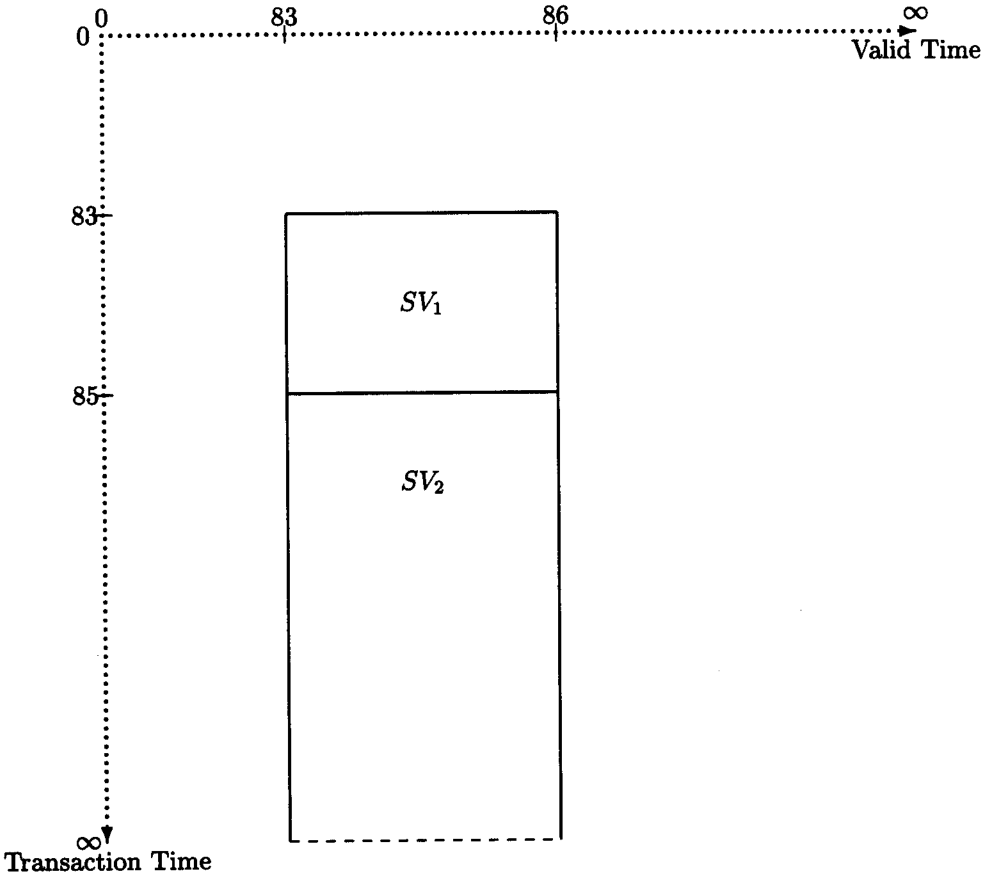
Example 5.7 Considering the three transactions in Example 5, they can be expressed as shown in the previous section concerning valid-time schema versioning. The effect of TR_1 this time can be represented as follows:



SV_1 $[1/1/83 - \infty]_t \times [1/1/83 - 12/31/85]_v$

EMPLOYEE(NAME,ADDRESS,CITY)

The resulting schema version SV_1 is current, was created in the database on 1/1/1983, and is valid from the beginning of 1983 to the end of 1985. TR_3 leads to the following state:



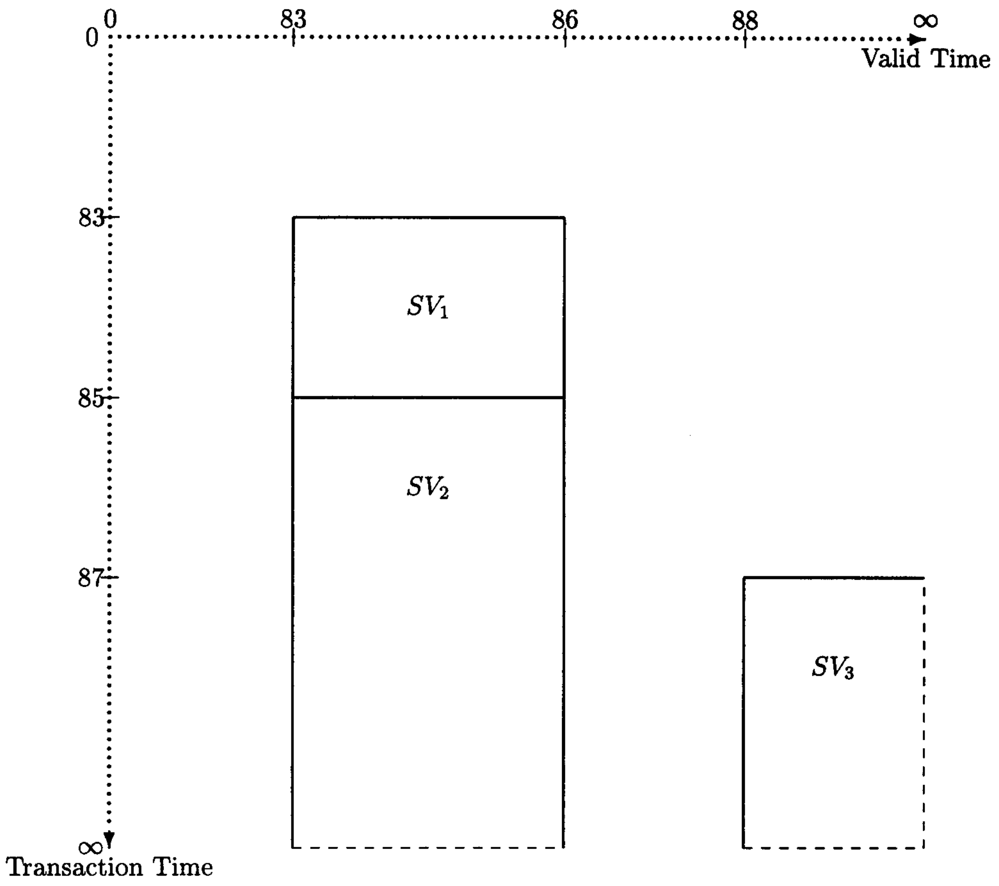
SV_1 $[1/1/83 - 12/31/84]_t \times [1/1/83 - 12/31/85]_v$

EMPLOYEE(NAME,ADDRESS,CITY)

SV_2 $[1/1/85 - \infty]_t \times [1/1/83 - 12/31/85]_v$

EMPLOYEE(NAME,CITY)

Notice how also SV_1, archived on 1985/1/1, is maintained in this case in the catalogues, and can later be used to roll back the schema of table EMPLOYEE. However, it is replaced in the current historical state by the new schema version SV_2, with the same validity. Eventually, the final state produced by TR_5 is:



SV_1 $[1/1/83 - 12/31/84]_t \times [1/1/83 - 12/31/85]_v$

```
EMPLOYEE(NAME,ADDRESS,CITY)
```

SV_2 $[1/1/85 - \infty]_t \times [1/1/83 - 12/31/85]_v$

```
EMPLOYEE(NAME,CITY)
```

SV_3 $[1/1/87 - \infty]_t \times [1/1/88 - \infty]_v$

```
EMPLOYEE(NAME,CITY,PHONE)
```

No schema version is archived in this case, because the validity assigned to the new schema version does not overlap the validity of any previous schema version, including the modified one (SV_2). □

The management of bitemporal schema versions at intensional level combines the features of transaction-time and valid-time schema versioning. It can be formalized in the following algorithm:

Algorithm 8

```
• Let $SV_k$ be the schema version to be modified.
  It is determined as the schema version in use for the relation to be
  modified along valid time and current along transaction time
  (empty if the schema change is a CREATE TABLE)

• Let $\mathcal{T}_v(SC)$ be the validity of the schema change.
  If there is no explicit VALID clause in the schema change, then it equals
  the validity of the schema version to be modified, computed as
  $\mathcal{T}_v(SC) = \cap_{r \in SV_k} \mathcal{T}_v(r)$, where $\mathcal{T}_v(r) = \{t_v | (now, t_v) \in \mathcal{T}_b(r)\}$
```

- Let SV_{n+1} be the new schema version to be inserted.
  SV_{n+1} is obtained from SV_k and from the schema change statement,
  by applying the required modifications
  (empty if the schema change is a DROP TABLE)

- **Creation of the New Schema Version**
  for each tuple $r \in SV_{n+1}$ set $\mathcal{T}_b(r) := \{\text{now}..\infty\}_t \times \mathcal{T}_v(SC)$ and insert r in the catalogues

- **Affected Schema Versions**
  for each schema version SV_i concerning the modified relation such that $\mathcal{T}_b(SV_i) \cap \{\text{now}..\infty\}_t \times \mathcal{T}_v(SC) \neq \emptyset$, then for each tuple $r \in SV_i$

  - update r in the catalogues by setting $\mathcal{T}_b(r) := \mathcal{T}_b(r) \setminus \{\text{now}..\infty\}_t \times \mathcal{T}_v(SC)$

If Single-Pool:

- *Do the same as for transaction-time schema versioning*

3.3.2. Extensional Management

As far as extensional management is concerned, bitemporal schema versioning can either be synchronous or asynchronous. Obviously, the distinction only applies to valid time, since synchronous management is always mandatory along transaction time. Therefore, in asynchronous bitemporal schema versioning we have synchronous management along transaction time and asynchronous management along valid time, whereas in synchronous bitemporal versioning we have synchronous management along both time dimensions.

With the single-pool solution, when a schema change is applied the whole data pool is always converted to the completed schema format, which can only grow. Actually, unlike valid-time schema versioning, previous schema versions can never disappear. Even schema versions completely overlapped by new ones are not physically deleted but only archived along transaction time. With the multi-pool solution there are in fact no substantial differences with respect to the previous cases.

Asynchronous and Synchronous Single-Pool Solution

Since it is a single-pool solution, there is no difference between asynchronous and synchronous management. Moreover, the pool management algorithm is exactly the same as the one for the single-pool solution of transaction-time schema versioning. It differs from the single-pool solution of valid-time schema versioning since the *completed schema* never shrinks. Previous schema versions completely overlapped by new ones in valid time (like SV_1 in the example) can always be accessed by means of rollbacks on transaction time.

Considering the transactions in Section 5, their effects on the single pool are exactly the same as for transaction-time schema versioning described by Example 5.2 in Section 3.1.2.

Asynchronous Multi-pool Solution

As in the other multi-pool solutions, each data pool is exactly formatted according to the corresponding schema version.

Example 5.8 Considering Example 5 in Section 5, the effects of the transactions from TR_1 to TR_5 can be represented as follows. After TR_1 creates an empty data pool DP_1, the insertion in transaction TR_2 gives:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	$\{1/1/84..\infty\}$

The schema change TR_3 creates a new data pool DP_2: unlike the valid-time schema versioning case, even if the new schema version SV_2 completely overlaps the modified one SV_1, the schema version SV_1 and the corresponding data pool DP_1 are not eliminated but only archived (they could be accessed by means of a rollback). However, since the management is synchronous along transaction time, the temporal pertinence of their (transaction-time) data is restricted to be contained in the transaction-time pertinence of the corresponding schema versions. The result is as follows:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/84}

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/85..∞}

The data modification TR_4 only affects pool DP_2 with the following results:

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/84}

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/85..12/31/85}
Blanc	Bruxelles	{1/1/86..∞}

Transaction TR_5 creates a new data pool DP_3 associated to the new schema version SV_3. Notice that, since the affected and the new schema versions (i.e. SV_2 and SV_3) are both current, we eventually have current data in both pools DP_2 and DP_3. In particular, no timestamp restriction is required for data in DP_2.

DP_1

NAME	ADDRESS	CITY	T
Blanc	BD St Michel 87	Paris	{1/1/84..12/31/84}

DP_2

NAME	CITY	T
Blanc	Paris	{1/1/85..12/31/85}
Blanc	Bruxelles	{1/1/86..∞}

DP_3

NAME	CITY	PHONE	T
Blanc	Bruxelles	*Null*	{1/1/87..∞}

□

The pool management algorithm can briefly be described as follows.

Algorithm 9

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}$
• For each tuple $r \in DP_k$, consider the conversion from $SV_k$ to $SV_{n+1}$
    – for added columns: add new attributes padded with nulls
    – for enlarged domains: convert attribute values to the enlarged format
    – for dropped columns: eliminate the old attributes
    – for restricted domains: convert the attribute values to the restricted
      format
    – for changed time dimensions: apply the conversion map $M_{F_n F_{n+1}}$
    – store $r$ in $DP_{n+1}$
• For each affected data pool $DP_i$ (associated to the affected schema version
  $SV_i$, such that $\mathcal{T}_b(SV_i) \cap \{now..\infty\}_t \times \mathcal{T}_v(SC) \neq \emptyset$), then for each tuple $r \in DP_i$
    – if $F_i = t$: set $\mathcal{T}_t(r) := \mathcal{T}_t(r) \setminus \{now, \infty\}_t$
    – if $F_i = b$:  set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \setminus \{\texttt{now}, \infty\}_t \times \mathcal{U}_v$
• For each tuple $r \in DP_{n+1}$
    – if $F_{n+1} = t$:  set $\mathcal{T}_t(r) := \mathcal{T}_t(r) \cap \{\texttt{now}, \infty\}_t$
      (if $\mathcal{T}_t(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_{n+1} = b$:  set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \cap \{\texttt{now}, \infty\}_t \times \mathcal{U}_v$
      (if $\mathcal{T}_b(r) = \emptyset$ then delete $r$ from the data pool)
• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}$)
```

Synchronous Multi-Pool Solution

Transactions of our example in Section 5 (Example 2.1) are not applicable to this solution to highlight differences with the previous case, since the EMPLOYEE table is not versioned along valid time. Therefore, the results of the example do not differ from those illustrated in the previous subsection (remember the synchronous management along transaction time).

Let DP_k be the data pool associated with the current schema version SV_k. The schema versioning requires the creation of a new data pool, DP_{n+1}, to be associated with the new schema version SV_{n+1}. Extensional data are copied from DP_k to DP_{n+1}

As in the previous case, DP_k must still be retained after the schema change. However, since the management is synchronous, if data in the old and/or in the new data pool contain valid time, their timestamps must accordingly be restricted.

The pool management algorithm can briefly be described as follows.

Algorithm 10

```
• Initialize $DP_{n+1}$ according to $SV_{n+1}$
• For each tuple $r \in DP_k$, consider the conversion from $SV_k$ to $SV_{n+1}$
    – for added columns:  add new attributes padded with nulls
    – for enlarged domains:  convert attribute values to the enlarged format
    – for dropped columns:  eliminate the old attributes
    – for restricted domains:  convert the attribute values to the restricted
      format
    – for changed time dimensions:  apply the conversion map $M_{F_k F_{n+1}}$
    – Store $r$ in $DP_{n+1}$
• For each affected data pool $DP_i$ (associated to the affected schema version
  $SV_i$, such that $\mathcal{T}_b(SV_i) \cap \{\texttt{now}..\infty\}_t \times \mathcal{T}_v(SC) \neq \emptyset$), then for each tuple $r \in DP_i$
    – if $F_i = t$:  set $\mathcal{T}_t(r) := \mathcal{T}_t(r) \setminus \{\texttt{now}..\infty\}_t$
      (if $\mathcal{T}_t(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_i = v$:  set $\mathcal{T}_v(r) := \mathcal{T}_v(r) \setminus \mathcal{T}_v(SC)$
      (if $\mathcal{T}_v(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_i = b$:  set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \setminus \{\texttt{now}, \infty\}_t \times \mathcal{T}_v(SC)$
      (if $\mathcal{T}_b(r) = \emptyset$ then delete $r$ from the data pool)
• For each tuple $r \in DP_{n+1}$
    – if $F_{n+1} = t$:  set $\mathcal{T}_t(r) := \mathcal{T}_t(r) \cap \{\texttt{now}, \infty\}_t$
      (if $\mathcal{T}_t(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_{n+1} = v$:  set $\mathcal{T}_v(r) := \mathcal{T}_v(r) \cap \mathcal{T}_v(SC)$
      (if $\mathcal{T}_v(r) = \emptyset$ then delete $r$ from the data pool)
    – if $F_{n+1} = b$:  set $\mathcal{T}_b(r) := \mathcal{T}_b(r) \cap \{\texttt{now}, \infty\}_t \times \mathcal{T}_v(SC)$
      (if $\mathcal{T}_b(r) = \emptyset$ then delete $r$ from the data pool)
• Store the pointer to $DP_{n+1}$ in catalogues ($SV_{n+1}$)
```

4. QUERY LANGUAGE DESIGN

In this section we show which extensions to the query language are required in a system supporting one of the proposed schema versioning solutions. The extensions are still based on the temporal query language TSQL2, and are also useful for a better understanding of the different solutions.

4.1. TSQL2 Extensions

We have seen from the Introduction that the TSQL2 language is provided with a special SET SCHEMA statement in order to support schema version selection [15]. We also introduced, quite informally in our examples, the use of a SET SCHEMA statement for schema version selection in valid-time and bitemporal schema versioning. However, we have not yet discussed in detail the extensions to the query language required to consistently support schema selection in all the proposed schema versioning solutions.

The first extension we propose concerns the SET SCHEMA statement, whose complete form (in bitemporal schema versioning) is:

```
SET SCHEMA VALID < datetime value expression >
   AND TRANSACTION < datetime value expression >
```

It allows to set default values for valid and transaction time to be used for schema version selection. As in TSQL2, it is an instruction that alters the *state* of the database, in the sense that the temporal values set as defaults are used by every subsequent TSQL2 statement, until a new SET SCHEMA is executed or an end-of-transaction command is issued. Embracing the TSQL2 philosophy, the adoption of a stand-alone statement for schema selection avoids overloading the syntax of retrieval/modification statements with an additional schema selection clause. Notice that this solution tends to maximize the reuse of old applications: a SET SCHEMA statement can be used to select the schema versions according to which the application was coded before launching the application as it is. Without the SET SCHEMA, an additional clause for schema selection would be required in our approach also in ALTER statements for schema changes in valid-time and bitemporal schema versioning (where a VALID clause to specify the schema change validity has already been added).

In a straightforward syntactic variant, only one of the VALID and TRANSACTION parts of the SET SCHEMA statement can be specified and in that case only the corresponding default temporal value is changed. It should be remembered that default values set by the SET SCHEMA statement only concern *intensional* data selection and have no influence on extensional data selection (extant data may also be snapshot tables, as in the presumably common case of a non-temporal database supporting schema versioning).

Notice that, although the valid-time default is used to select the schema version for the execution of any TSQL2 instruction (schema changes, retrievals, updates), the transaction-time default is used only for retrievals (SELECT statements): owing to the definition of transaction time, *current* schema versions can only be modified and extensional data can only be inserted, deleted or updated through *current* schema versions. Therefore, for the execution of intensional/extensional changes the default transaction-time value set by the SET SCHEMA is simply ignored, and the current transaction time "now" is used instead for schema selection.

Actually, the two specifiers VALID and TRANSACTION in the SET SCHEMA are strictly needed only in bitemporal schema versioning. As there is no danger of ambiguity when a single temporal dimension is used for schema versioning, the basic TSQL2 SET SCHEMA without specifiers could be used in monotemporal schema versioning. A better alternative could also leave as *optional* the VALID and TRANSACTION specifiers in valid- and transaction-time schema versioning, respectively. When a SET SCHEMA setting one time constant without specifiers is found with bitemporal schema versioning, we can assume that a VALID modifier is understood. The examples in Section 3 agree with these conventions, as a VALID modifier is understood.

When extensional data are also temporal, standard TSQL2 temporal selection conditions can be used in SELECT statements. For instance, if EMPLOYEE is a valid-time table with valid-time schema versioning (and asynchronous management), the query:

```
SET SCHEMA VALID DATE '1992-01-01' ;
SELECT * FROM EMPLOYEE
   WHERE VALID(EMPLOYEE) OVERLAPS DATE '1990-01-01'
```

retrieves employee data valid at the beginning of 1990 through the schema version valid at the beginning of 1992.

However, if synchronous management is adopted, the two temporal selection conditions on intensional and extensional data would conflict. In order to maintain maximal compatibility with standard TSQL2, we propose that, in the case of conflicts, the temporal selection conditions in the WHERE clause prevail on the defaults, and their effect on the schema selection mechanism is local to the query execution. Therefore, in the query above, the date 1/1/90 will be used to select both data and schema versions, if synchronous management is adopted.

Consider also the most complex example of a query on bitemporal data with bitemporal schema versioning:

```
SET SCHEMA VALID DATE '1992-01-01'
   AND TRANSACTION DATE '1993-01-01';
SELECT * FROM EMPLOYEE
   WHERE VALID(EMPLOYEE) OVERLAPS DATE '1990-01-01'
     AND TRANSACTION(EMPLOYEE) OVERLAPS DATE '1991-01-01'
```

With asynchronous management (along valid time), the time constant 1/1/1991 is used to select by transaction time both the schema and the data, the time constant 1/1/1992 is used to select by valid time the schema, and 1/1/1990 is used to select by valid time the data.

With synchronous management (along both time dimensions), the time constant 1/1/1991 is used to select by transaction time both the schema and the data, while the time constant 1/1/1991 is used to select by valid time both the schema and the data, locally overriding the default valid-time value 1/1/1992 set for schema selection.

Since the management on transaction-time is in any case synchronous, the transaction-time default 1/1/93 is never used for schema version selection. It will however be used in the following example:

```
SELECT * FROM EMPLOYEE
   WHERE VALID(EMPLOYEE) OVERLAPS DATE '1990-01-01'
```

since no explicit selection condition on data transaction-time is present in the WHERE clause.

4.2. Schema Selection and Query Processing

Once a schema version has been selected, via implicit or explicit conditions, it must be used to access the underlying extant data and accordingly retrieve the required ones. The retrieval phase may require some filtering, depending on the type of schema versioning adopted.

In the case of multi-pool solution, a distinct data pool corresponds to each schema version. Therefore, the selection of a schema version automatically implies the selection of the corresponding data pool, from which extant data can be directly retrieved in the format they are stored.

In the case of single-pool, all the data are stored in the same "enlarged" format and thus, a filtering is required (which might include the conversion of temporal format) in order to adapt data to the specific schema version used for the access.

Furthermore, if synchronous versioning is adopted for the single-pool, the filtering phase must also restrict the time pertinence of retrieved data, along the same temporal dimension(s) used for synchronous versioning, to be completely contained within the temporal pertinence of the corresponding schema version.

For example, if schemata and data are both versioned along valid time with synchronous management, once a schema version has been selected, the whole single-pool must be scanned to retrieve the corresponding data. If $\mathcal{T}_v(SV)$ is the validity of the selected schema version and $\mathcal{T}_v(r)$ is the validity of the generic tuple r fetched from the single-pool, the following valid-time value is constructed:

$$\mathcal{T}_v(SV) \cap \mathcal{T}_v(r)$$

If the result is not empty, then it is used as the new timestamp of the tuple r which is put in the query result, otherwise r is discarded.

Therefore, notice that although there are no differences between the synchronous and the asynchronous management of the single-pool in response to schema changes illustrated in the previous Section; significant differences emerge in response to queries and data modifications.

Consider now data modification statements. With transaction-time schema versioning, the current schema version is always used for data insertion, deletion or update. If also data are versioned along transaction time, only their current portion is affected and the management results automatically synchronous. In the case of multi-pool, data are modified in the last created (current) pool only.

It should also be noticed that in the presence of synchronous management, data are always accessed through the schema version by means of which they were inserted or last modified.

Example 6 Let us assume, for a comprehensive example, we are dealing with a database with valid-time schema versioning in the following initial intensional state:

SVx [1/1/90 - 12/31/92]

```
EMPLOYEE(NAME,ADDRESS,CITY|T)
```

SVy [1/1/93 - ∞]

```
EMPLOYEE(NAME,CITY,PHONE|T)
```

We further assume that EMPLOYEE is a valid-time table initially empty, and the following command has been executed:

```
SET SCHEMA DATE '1992-01-01'
```

to select the default schema version SVx. Hence, the TSQL2 statement:

```
INSERT INTO EMPLOYEE(NAME,CITY)
   VALUES ('Schwarz','Munich')
     VALID PERIOD '[1991-01-01 - forever]'
```

concerns the insertion in the table EMPLOYEE of a new tuple valid from 1991 on.

If a single-pool solution is adopted, the effect of the insertion, that is the same in the synchronous and the asynchronous case, is:

DP	NAME	ADDRESS	CITY	PHONE	T
	Schwarz	*Null*	Munich	*Null*	{1/1/91..∞}

Unspecified attribute values have been stored as nulls, as in traditional SQL. However, with asynchronous management, the default schema version SVx has been used for data insertion. Therefore, a defined value could have been specified for attribute ADDRESS but not for attribute PHONE without running into an error condition, as SVx does not contain the attribute PHONE. With synchronous management, Schwarz's data valid from 1/1/1991 to 12/31/1991 would have been inserted through SVx and those valid from 1/1/1992 on would have been inserted through SVy. The final result does not change, but neither ADDRESS nor PHONE values could have been specified without error in this case.

Notice that, in the asynchronous case, the same information could have been inserted also as:

```
INSERT INTO EMPLOYEE
   VALUES ('Schwarz',Null,'Munich')
     VALID PERIOD '[1991-1-1 - forever]'
```

since it is correctly based on schema version SVx. No similar rewriting is allowed in the synchronous case.

After the insertion, Schwarz's data can be retrieved using the schema version by means of which they were inserted but also by means of SVy, due to the semantics of the single-pool. With asynchronous management, the query:

```
SELECT * FROM EMPLOYEE
```

using SVx produces the result:

NAME	ADDRESS	CITY	T
Schwarz	*Null*	Munich	$\{1/1/91..\infty\}$

and the query:

```
SET SCHEMA DATE '1-1-1996' ;
SELECT * FROM EMPLOYEE
```

using SVy produces the result:

NAME	CITY	PHONE	T
Schwarz	Munich	Null	$\{1/1/91..\infty\}$

With synchronous management, the first query produces the result:

NAME	ADDRESS	CITY	T
Schwarz	*Null*	Munich	$\{1/1/91..12/31/91\}$

and the latter the result:

NAME	CITY	PHONE	T
Schwarz	Munich	*Null*	$\{1/1/92..\infty\}$

With a multi-pool solution, the effects of the insertion are the following (with asynchronous management the default schema version SVx is used):

DPx

NAME	ADDRESS	CITY	T
Schwarz	*Null*	Munich	$\{1/1/91..\infty\}$

DPy

NAME	CITY	PHONE	T
(empty)			

in the asynchronous case, while the result is:

DPx

NAME	ADDRESS	CITY	T
Schwarz	*Null*	Munich	$\{1/1/91..12/31/91\}$

DPy

NAME	CITY	PHONE	T
Schwarz	Munich	*Null*	$\{1/1/92..\infty\}$

in the synchronous case.

Also this time, only the attribute ADDRESS (or PHONE with default selected schema version SVy) could have been specified in the insertion with asynchronous management, and none of ADDRESS and PHONE could have been with synchronous management. □

4.3. Multischema Queries

In the discussion and examples above, we made an implicit assumption on the schema version selection mechanism. For the sake of simplicity, we always assumed that a single schema version is selected by a SET SCHEMA statement and that exactly one schema version qualifies for temporal selection conditions in the WHERE clause.

For instance, in transaction-time schema versioning, there are no problems with modification and creation statements which always operate through the current schema version. Anyway, the answer to a query like the following one is not obvious.

```
SELECT * FROM EMPLOYEE
   WHERE TRANSACTION(EMPLOYEE) OVERLAPS PERIOD '[1990-01-01 - 1991-12-31]'
```

As a matter of fact, if exactly one schema version is defined in the transaction-time interval 1990–1991, it can be used for data retrieval without difficulties, but if more than one schema version is defined in that interval, the query cannot be answered in a simple way and we have to define its precise semantics.

In general, when several schema versions qualify for the temporal selection conditions expressed in a query, we basically have the following alternatives:

- **Query Rejection.** The query cannot be answered in a simple way, thus an error exception is raised and the transaction aborts;

- **Multischema Answer.** The result of the query is fragmented in as many partial results as the number of schema versions selected. Each answer fragment is timestamped and formatted according to the corresponding schema version;

- **Schema Reduction.** The query is forced to become a simple one through the use of a single schema version built from those actually selected. For example, as already proposed in [13, 15], the *constructed schema* can be derived as the union or intersection of all the qualifying schema versions.

The first solution is the simplest to implement, yet the most restrictive. Syntactic and semantic constraints must be enforced on legal queries. As in TSQL2 the SET SCHEMA statement only permits a datetime expression (i.e. a time point), other restrictions should be imposed on the formulation of temporal conditions in the WHERE clause†. A further restriction would involve the datetime expressions, which should be simple constants or expressions involving only constants, since the value of more general expressions could depend on the query answer or on intermediate results of query evaluation. In any case, if the selection of multiple schema versions is detected at run-time by the query processor, an error condition is generated. Notice that syntactic restrictions may not be in general sufficient to ensure correct queries: for example, "1990-01-01" is a datetime constant which can be used for schema selection, but if the time granularity is finer than a day, multiple schemata may have been defined for the same date (e.g. if a schema change can be recorded as occurred at noon). On the other hand, if an interval is used for schema selection and exactly one schema version is defined on that interval, there is no reason to reject the query. Therefore, some tradeoff between syntactic and semantic restrictions should be adopted.

The second solution is apparently very flexible. The results of a *multischema* query could be presented, for instance, as follows:

--- Schema {1/1/90..12/31/92}

NAME	ADDRESS	CITY
Brown	King's Road 15	London
Rossi	Via Veneto 7	Rome

†For example, accepted selection conditions should only consist of temporal predicates in the form OVERLAPS/CONTAINS a datetime expression, AND-ed with the rest of the WHERE clause predicates.

`--- Schema` {1/1/93..12/31/95}

NAME	ADDRESS
Brown	London
Rossi	Rome

`--- Schema` {1/1/96..∞}

NAME	ADDRESS	PHONE	T
Brown	London	226644	{1/1/90..6/15/94}
Brown	Dublin	132457	{6/16/94..∞}
Rossi	Rome	773355	{5/10/95..∞}

Although this kind of result is definitely acceptable for a human user, interactively working with a video terminal or reading a printout, it could be hardly used by a compiled application. Multischema applications should be expressly developed to deal with this kind of query answer and legacy applications could not be run against the returned data. Moreover, for the implementation of multischema applications, extensions of the mechanisms of data exchange between the database management system and applications are also needed (e.g. "multischema cursors" and related operations). The solution is thus attractive for the development of novel applications, able to exploit the full potentialities of a system supporting schema versioning (e.g. for automatic auditing).

The third solution represents a sort of tradeoff between the first two approaches. However, a loss of information with respect to the actual contents of the database is implied in the adoption of the completed/constructed schema and in its use to access extensional data. Applications adopting this solution may suffer from this semantic shortcoming (e.g. auditing procedures could not rely on retrieved data, which do not necessarily give an accurate picture of the database evolution).

Moreover, maximal system flexibility can be provided by the concurrent availability of the second and third solutions. For instance, the query answering process can usually be based on multischema answers, unless explicitly forced to use the completed schema by a special query syntax (e.g. by means of the TSQL2 "double asterisk" described in the Introduction).

Last but not least, if a query involves multiple schemata, but only columns common to all of them are required in query evaluation and final projection, the query should normally be answered by the system and its results used without special care. As a matter of fact, a *completed schema* based on single schemata intersection can be safely used in this particular case without loss of information.

5. FURTHER CONSIDERATIONS AND CONCLUSIONS

In this paper we have provided a study of valid-time, transaction-time and bitemporal schema versioning in a multitemporal environment. Moreover, two distinct solutions (single- and multi-pool) have been proposed for the management of extensional data in all the temporal types of schema versioning. A further distinction has been made between the synchronous or asynchronous management of intensional and extensional data. Our goal was to introduce at a logical level a wide spectrum of solutions and design options for the management of schema versioning and illustrate the potentialities of their different features.

We now provide a brief discussion and comparison of the solutions proposed. A complete comparison with other approaches is not possible since, as far as we know, valid time was not considered in current literature on schema versioning, and synchronous versus asynchronous management of the single- and the multi-pool was never discussed before. Previous proposals, like the TSQL2 schema versioning mechanism, based on transaction-time schema versioning with asynchronous management (with data also versioned along transaction-time) cannot be considered acceptable in the light of our schema versioning classification, unless the roll-back semantics of transaction time is distorted (see Section 2.5).

The comparison of the solutions proposed in this paper is mainly based on semantic properties, that is the independence of data versioning with respect to schema versioning and the operations which can be effected. To a limited extent, also storage requirements are taken into account. A more detailed discussion, involving design of physical data structures and analysis of maintenance and query processing costs was beyond the scope of the present paper and will be the subject of future work.

The evolution of data is conceptually independent of the evolution of schemata. This is the reason, for example, for the freedom of choice of different temporal dimensions for data and schema versionings, which can be made according to specific design requirements. However, such independence may be supported to a different degree, depending on the schema versioning solution adopted.

An effective independence is guaranteed if the temporal versioning of data and schemata are made along different time dimensions (e.g. transaction-time schema versioning and valid-time data versioning). If data and schemata are versioned along common temporal dimension(s), the independence is still effective for asynchronous management of data with respect to schemata.

On the contrary, synchronous management permits a lower degree of independence, since it forces the temporal pertinence of data to depend on the temporal pertinence of the corresponding schema versions. For instance, transaction-time schema versioning and transaction-time data versioning are always synchronous, thus giving rise to a sort of temporal dependence between data and schemata: current data only can evolve through current schemata only and data are always accessed through the schema by means of which they were inserted.

In the most general case, the temporal format of schema versioning and of data versioning may or may not have common dimensions (e.g. bitemporal schema versioning and valid-time data versioning have valid-time as the common time dimension). The synchronous or asynchronous management determines, respectively, dependence or independence along the common dimension(s). For example, with valid-time schema versioning and bitemporal data versioning, independence holds or fails to do so along valid time on the basis of asynchronous or synchronous management.

Furthermore, when data versioning management is independent of schema versioning, the choice of the single-pool solution guarantees the (independent) evolution of a single copy of data. The choice of the multi-pool solution gives a further degree of freedom, since it allows several distinct evolutions of data, each one relative to a specific schema version. The price to pay with the multi-pool solution is a complication in the underlying data model, in particular when extant data are temporal. Applications must not confuse data versioning at extensional level with data *metaversioning* at intensional level.

On the other hand, when synchronous management is adopted, the choice of a single-pool implementation further limits the independence, since the evolution of data connected to different schema versions is bound by concurrent updates of a single copy of stored data (see Section 2.4). However, this last kind of dependence is unacceptable and a multi-pool solution must be adopted if serializability of transactions using different schema versions has to be enforced, unless enhanced specific concurrency control policies are adopted. Locking protocols to preserve serializability with the single-pool solution can easily be devised, also introducing different degrees of isolation between operations relative to distinct schema versions. For instance, update transactions could only be allowed to work with exclusive locks on data, whichever schema versions they use; other trasactions working on the same data also through different schema versions would have to wait for the locks to be released. A more restrictive policy could allow updates only through the schema version by means of which the data were created. Another policy (equivalent to Roddick's partial schema versioning) could allow updates only through one designated schema version (e.g. the current one).

In general, application requirements influence the design choices for a specific schema versioning solution. The choice of the temporal dimensions to be used for data and schema versioning is dictated by the application requirements of historical recording, archiving and auditing at extensional and intensional levels. When independence is the most important requirement, the choice should be oriented to asynchronous management, if different temporal dimensions for data and schema versioning cannot be adopted. If independence is not an issue of concern, synchronous management can be adopted, so that storage space can also be saved. However, when storage space is critical, the choice is not straightforward and must first consider the single-pool versus the multi-pool.

A first quantitative analysis on the amount of storage space may lead to preferring the single-pool solution. Even if cost evaluation problems are not faced in this paper, it is quite evident that the single-pool solution in most cases minimizes the amount of memory used. The multi-pool solution implies the proliferation of the number of data-pools (as many as the number of schema versions) and therefore a large amount of duplicated data. The single-pool solution requires an enlargement of the data format and the possible introduction of a large quantity of null values but avoids duplication to a great extent.

When the multi-pool solution is chosen, its asynchronous management is more expensive than the synchronous one, since it requires an even higher degree of duplication. As a matter of fact, with synchronous management, the duplication after a schema change is limited to those data whose temporal pertinence extends across different pools and, thus, needs to be replicated in all such pools with the timestamps accordingly split.

Other considerations on the choice of synchronous versus asynchronous management could be made on the basis of the most frequent operations. For example, the multi-pool solution, in the presence of transaction-time schema versioning and valid-time data versioning, privileges operations which require the retrieval of complete valid-time histories of data through a specific transaction-time schema version, since a complete history can be retrieved by accessing exactly one pool (which is also small) and does not require a final filtering phase.

The single-pool behaves even worse when only small history portions of data are required, since a complete scan of the pool is necessary to select data by time. As a matter of fact, in the multi-pool solution with synchronous management, schema versions addressing the pools represent (at a logical level) a sort of temporal indexing of data. The larger the number of schema versions that have been defineu, the more selective the indexing mechanism. On the other hand, if complete histories are required, all the pools must be accessed.

Notice also that the retrieval of complete histories in the multi-pool solution with synchronous management implies a multischema query (with the difficulties highlighted in Section 4.3), whereas complete data histories can always be stored and accessed through a single schema version with any other schema versioning solution.

In conclusion, the single-pool solution avoids data duplication, even if it requires an enlargement of the data format required for storage. The multi-pool requires the duplication of the current portion of data in transaction-time schema versioning, and duplication of the whole pools affected by a schema change in the case of valid-time schema versioning (current affected pools in the case of bitemporal versioning). However, the multi-pool solution allows several independent evolutions of data, each one relative to a specific schema version, whereas a single copy is maintained in the single-pool. Furthermore, the single-pool forces data to be converted according to the desired schema version at query processing time, if the desired format is not the one according to which they are stored.

Future work will be devoted to a more complete comparison of the proposed solutions in terms of system performance, and taking into account distinct application requirements and workloads. A prototype implementation of the different schema versioning solutions is under development at the University of Bologna and will serve as a testbed for experimental evaluation.

Acknowledgements — A preliminary version of this paper [4] was presented at the VLDB-affiliated International Workshop on Temporal Databases in Zurich. The authors would like to thank all the colleagues for providing valuable feedback at the temporal Workshop, the anonymous referees for their solicitations, and R. Snodgrass for his useful comments on early proposals of TSQL2 extensions.

REFERENCES

[1] G. Ariav. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering*, **6**:451–467 (1991).

[2] C. De Castro, F. Grandi and M.R. Scalas. Semantic interoperability of multitemporal relational databases. In *Entity-Relationship Approach - ER '93*, Lecture Notes in Computer Science **823**, Springer (1994).

[3] C. De Castro, F. Grandi and M.R. Scalas. Management of schema versions in multitemporal relational databases (in *Italian*). In *Proceedings of the 2nd National Conference on Advanced Database Systems*, Rimini, Italy (1994).

[4] C. De Castro, F. Grandi and M.R. Scalas. On schema versioning in temporal databases. In *Proceedings of the International Workshop on Temporal Databases*, Z"urich, Switzerland (1995).

[5] ISO/IEC Standard 9075:1992. *Database Language SQL*. International Standard Organization, Geneva, Switzerland (1992).

[6] C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes and S. Jajodia, editors, C.E. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsoupoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M.R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A. Tansel, P. Tiberio and G. Wiederhold. A consensus glossary of temporal database concepts. *ACM SIGMOD Record* **23**(1):52–64 (1994).

[7] C.S. Jensen, M.D. Soo and R.T. Snodgrass. Unifying temporal data models via a conceptual model. *Information Systems* **19**(7):513–548 (1994).

[8] N. Kline. An update of the temporal database bibliography. *ACM SIGMOD Record* **22**(4):66–80 (1993).

[9] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented databases. *Data & Knowledge Engineering* **4**(1):43–67 (1989).

[10] E. Odberg. Schema Evolution Bibliography. World-Wide Web page,
URL: `http://www.fou.telenor.no/brukere/erikod/smm.txt`.

[11] G. Özsoyoğlu and R.T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* **7**(4):513–532 (1995).

[12] J.F. Roddick. Schema evolution in database systems-An annotated bibliography. *ACM SIGMOD Record* **21**(4):35–40 (1992).

[13] J.F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology* **37**(7):383–393 (1996).

[14] J.F. Roddick, N.G. Craske and T.J. Richards. A taxonomy for schema versioning based on the relational and entity relationship models. In *Entity-Relationship Approach - ER '93*, Lecture Notes in Computer Science **823**, Springer (1994).

[15] J.F. Roddick and R.T. Snodgrass. Schema versioning. In [23], chapter 22.

[16] M.R. Scalas, A. Cappelli and C. De Castro. A model for schema evolution in temporal relational databases. *Proceedings of the 7th IEEE European Computer Conference*, Paris Evry, France (1993).

[17] R.T. Snodgrass, I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y. Cliff Leung, N.Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada. TSQL2 language specification. *ACM SIGMOD Record* **23**(1):65–86 (1994).

[18] R.T. Snodgrass and F. Grandi. Schema specification. In [23], chapter 11.

[19] M.D. Soo. Bibliography on temporal databases. *ACM SIGMOD Record* **20**(1):14–23 (1991).

[20] M. Stonebraker. Hypothetical data bases as views. *Proceedings of the ACM-SIGMOD Conference on Management of Data*, Ann Arbor, Michigan (1981).

[21] A. Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev and R.T. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, Benjamin/Cummings (1993).

[22] V.J. Tsotras and A. Kumar. Temporal database bibliography update. *ACM SIGMOD Record* **25**(1):41–51 (1996).

[23] *The TSQL2 Language Design Commitee*: R.T. Snodgrass (ed.), I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y. Cliff Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada. *The TSQL Temporal Query Language*, Kluwer (1995).