24

Notions of Upward Compatibility of Temporal Query Languages

John Bair, Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass

Migrating applications from conventional to temporal database management technology has received scant mention in the research literature. This paper formally defines three increasingly restrictive notions of *upward compatibility* which capture properties of a temporal SQL with respect to conventional SQL that, when satisfied, provide for a smooth migration of legacy applications to a temporal system. The notions of upward compatibility dictate the semantics of conventional SQL statements and constrain the semantics of extensions to these statements. The paper evaluates the seven extant temporal extensions to SQL, all of which are shown to complicate migration through design decisions that violate one or more of these notions. We then outline how SQL–92 can be systematically extended to become a temporal query language that satisfies all three notions.

Keywords: upward compatibility, temporal upward compatibility, temporal query language, TSQL2, SQL

1 Introduction

A wide range of database applications manage time-varying information. These include financial applications such as portfolio management, accounting, and banking; record-keeping applications, including personnel, medical-record, and inventory; and travel applications such as airline, train, and hotel reservations and schedule management. In fact, it is difficult to identify a database application that does *not* involve time-varying data.

Currently, such applications typically use conventional relational systems. However, research in temporal data models and query languages [26, 18, 25] clearly demonstrates that applications that manage temporal data may benefit substantially from built-in temporal support in the database management system (DBMS). The potential benefits from such support are several. Application code is substantially simplified. Due to faster development of code, which is also easier to comprehend and thus maintain, higher programmer productivity results. With built-in support, more data processing may be left to the DBMS, leading to much better performance.

There is, however, still a chasm between the approaches now used for developing such applications and the new approaches that have been proposed by the temporal database research community. While 1200 papers on temporal databases have appeared, over 300 during the last two years alone [27], legacy systems and the migration of such systems to new technologies has been almost totally overlooked.

This paper considers *upward compatibility*, which has been claimed to offer several potential advantages (this is related to the notion of 'seamless', as in "the transition from classical databases to [a temporal] model should be conceptually and literally seamless." [9, p. 51]). In the context of migrating from a conventional DBMS to a temporal DBMS, upward compatibility offers an evolutionary means of introducing new technology. It provides a business enterprise with an upgrade path that preserves its investment in legacy databases. Implementers can incrementally build new features on top of existing products, by gradually learning and incorporating new language elements into their applications.

We assume that the DBMS interface is captured in a data model and thus talk about the migration of application code using an existing data model to using a new data model. We examine what it means for a temporal data model and query language to be upwardly compatible with a conventional data model, such as SQL. While the term has been used informally, we could find no formal definition (the same holds for the term 'seamless'). This paper examines the issues behind this intuitive idea, and formalizes several increasingly restrictive notions of upward compatibility, specifically, syntactic upward compatibility, upward compatibility, and temporal upward compatibility. We evaluate the seven extant temporal extensions to SQL, including two designed by us. We show that all seven violate one or more of these useful properties. Finally, we demonstrate how SQL–92 can be

extended to a temporal data model while simultaneously satisfying all three notions of upward compatibility.

2 Characterizing Upward Compatibility

We adopt the convention that a data model consists of three components, namely a set of data structures, a set of constraints on those data structures, and a language for updating and querying the data structures [28]. In this paper we emphasize the data structures and the data manipulation language. As we progress, it should be clear that the definitions and discussions of this section also apply to integrity constraints, although for simplicity we will not address these explicitly. Notationally, *M = (DS, QL)* then denotes a data model, *M*, consisting of a data structure component, *DS*, and a query language component, *QL*. Thus, *DS* is the set of all databases, schemas, and associated instances, expressible by *M*, and *QL* is the set of all query and modification statements in *M* that may be applied to some database in *DS*. We use db to denote a database; a statement is denoted by s and is either a query q or a modification m (e.g., in SQL–92, any `INSERT`, `DELETE`, or `UPDATE` statement).

As the existing model is given, the focus is on formulating requirements to the new data model. The definitions are conceptually applicable to the transition from any data model to a new data model. However, we have found it convenient to assume that the transition is from a non-temporal to a temporal data model, specifically from the SQL–92 standard [14] to (some) Temporal SQL.

2.1 Upward Compatibility

Perhaps the most important concern in ensuring a smooth transition of application code from an existing data model to a new data model is to guarantee that all application code without modification will work with the new system exactly the same as with the existing system. The next two definitions are intended to capture what is needed for that to be possible.

We define a data model to be *syntactic upward compatible* with another data model if all the data structures and legal query language statements of the latter model are contained in the former model.

Definition 1 Let $M_1 = (DS_1, QL_1)$ and $M_2 = (DS_2, QL_2)$ be two data models. Model M_1 is *syntactically upward compatible* with model M_2 if

- $\forall db_2 \in DS_2$ $(db_2 \in DS_1)$ and
- $\forall s_2 \in QL_2$ $(s_2 \in QL_1)$. □

The first condition states that all data structures of the (existing) model M_2 must be contained in the data structure component of the (new) model M_1; the second

condition states the same property, but for the query language component instead of for the data structures.

Note that this relationship between the two data models is asymmetric, thus providing credence to the adjective 'upward'.

Next, for a data model to be *upward compatible* with another data model, we add the requirement that all statements expressible in the existing language must evaluate to the same result in both models.

For a query language expression s and an associated database db, both legal elements of QL and DS of data model $M = (DS, QL)$, define $\langle\langle s(db)\rangle\rangle_M$ as the result of applying s to db in data model M. With this notation, we can precisely describe the requirements to a new model that guarantee uninterrupted operation of all application code.

Definition 2 Let $M_1 = (DS_1, QL_1)$ and $M_2 = (DS_2, QL_2)$ be two data models. Model M_1 is *upward compatible* with model M_2 if

- M_1 is syntactically upward compatible with M_2, and
- $\forall db_2 \in DS_2\ (\forall s_2 \in QL_2\ (\langle\langle s_2(db_2)\rangle\rangle_{M_2} = \langle\langle s_2(db_2)\rangle\rangle_{M_1}))$. □

The first condition, syntactic upward compatibility, implies that all existing databases and query language statements in the old system are also legal in the new system. The second condition in the definition states that for all data structures and associated query language statements in the (existing) model M_2, evaluating them in the (new and existing) models gives identical results. It thus guarantees that all existing statements compute the same results in the new system as in the old system. Thus, the bulk of legacy application code is not affected by the transition to a new system.

Figure 1 illustrates the relationship between Temporal SQL and SQL–92. In the figure, a conventional table is denoted with a rectangle. (In this paper, we use the terminology adopted in SQL: table, row, and column, rather than the terminology introduced by Codd [7]: relation, tuple, and attribute.) The current state of this table is the rectangle in the upper-right corner. Whenever a modification is made to this table, the previous state is discarded; hence, at any time only the current state is available. The discarded prior states are denoted with dashed rectangles; the right-pointing arrows denote the modifications that took the table from one state to the next.

When a query q is applied to the current state of a table, a resulting table is computed, shown as the rectangle in the bottom right corner. While this figure only concerns queries over single tables, the extension to queries over multiple tables is clear.

Upward compatibility states that (1) all instances of tables in SQL–92 are instances of tables in Temporal SQL, (2) all SQL–92 modifications to tables in

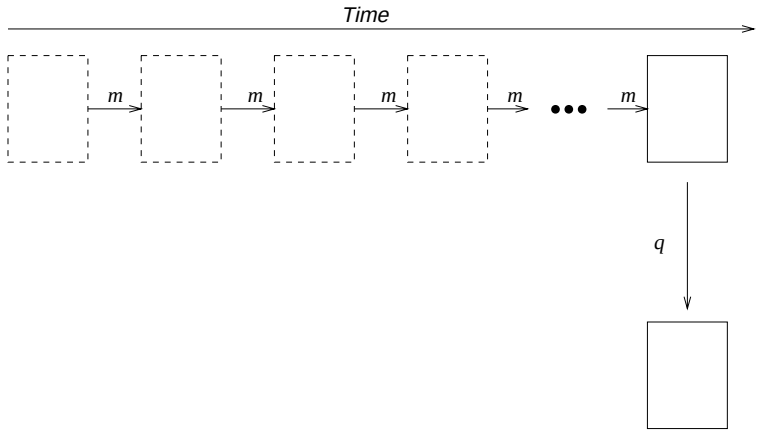

Figure 1: Upward Compatible Queries

SQL–92 result in the same tables when the modifications are evaluated according to Temporal SQL semantics, and (3) all SQL–92 queries result in the same tables when the queries are evaluated according to Temporal SQL.

By requiring that Temporal SQL is a strict superset (i.e., only *adding* constructs and semantics), it is relatively easy to ensure that Temporal SQL is upward compatible with SQL–92.

2.2 Temporal Upward Compatibility

The above minimal requirements are essential to ensure a smooth transition to a new temporal data model, but they do not address all aspects of migration. Specifically, assume that an existing data model has been replaced with a new temporal model. No application code has been modified, and all tables are thus snapshot tables. Upward compatibility ensures that all applications work as before, under the new temporal model.

Now, an existing or new application needs support for the temporal dimension of the data in one or more of the existing tables. This is best achieved by changing the snapshot table to become a temporal table (e.g., by using a statement of Temporal SQL).

It is undesirable to be forced to change the application code that accesses the snapshot table that is replaced by a temporal table. We thus formulate a requirement stating that the existing applications on snapshot tables must continue to work with no changes in functionality when the tables they access are altered to become temporal. Specifically, *temporal upward compatibility* requires that each query will return the same result on an associated snapshot database as on the temporal counterpart of the database. Further, this property is not affected by modifications to those temporal tables. The precise definition is given next and is explained in the following.

Definition 3 Let $M_T = (DS_T, QL_T)$ and $M_S = (DS_S, QL_S)$ be temporal and snapshot data models, respectively. Also, let $\mathcal{T}$ be an operator that changes the type of a snapshot table to the temporal table with the same explicit columns. Next, let $m_1, m_2, \ldots, m_n$ $(n \geq 0)$ denote modification operations. With these definitions, model M_T is *temporal upward compatible* with model M_S if

- M_T is upward compatible with M_S,
- $\forall db_S \in DS_S\ (\mathcal{T}(db_S) \in DS_T)$, and
- $\forall db_S \in DS_S\ (\forall m_1, \ldots, m_n (n \geq 0)$
  $(\forall q_S \in QL_S\ (\langle\!\langle q_S(m_n(m_{n-1}(\ldots(m_1(db_S)\ldots))))\rangle\!\rangle_{M_S} =$
  $(\langle\!\langle q_S(m_n(m_{n-1}(\ldots(m_1(\mathcal{T}(db_S))))))\rangle\!\rangle_{M_T}))))$. □

First, upward compatibility is required. The second condition states that when applying the type-change operator to any data structure (e.g., table) of the snapshot model, the result is a legal data structure in the temporal model. This is required for the third condition to be meaningful. To understand this condition, consider the two sides of the equality sign in the second line. On the two sides, the same sequence of modification statements is applied to a snapshot data structure and its temporalised counterpart, respectively. Then the same query from the snapshot model is applied to the two results of the modifications. The results of the queries when evaluated in the snapshot model and the temporal model, respectively, must be identical. The first line simply states that this must hold for all (meaningful) combinations of data structures, finite sequences of snapshot-model modification statements, and snapshot-model query language statements. We proceed to provide a more intutive and less technical explanation of this definition.

Assume that, when moving to the new system, some of the existing (snapshot) tables are transformed into temporal tables without changing the existing set of (explicit) columns. This transformation is denoted by $\mathcal{T}$ in the definition. Then the same sequence of modification statements, denoted by the m_i in the definition, is applied to the snapshot and the temporal databases. Next, consider any query in the snapshot model. Such queries are also allowed in the temporal model, due to upward compatibility being required. The definition states that any such query evaluated on the resulting temporal database, using the semantics of the temporal query language, yields the same result as when evaluated on the resulting snapshot database, now using the semantics of the snapshot query language.

Temporal upward compatibility is illustrated in Figure 2. When temporal support is added to a table, the history is preserved, and modifications over time are retained. In this figure, the rightmost dashed state was the current state when the table was made temporal. All subsequent modifications, denoted by the arrows, result in states that are retained, and thus are solid rectangles. Temporal upward compatibility ensures that the states will have identical contents to those states resulting from modifications of the snapshot table.

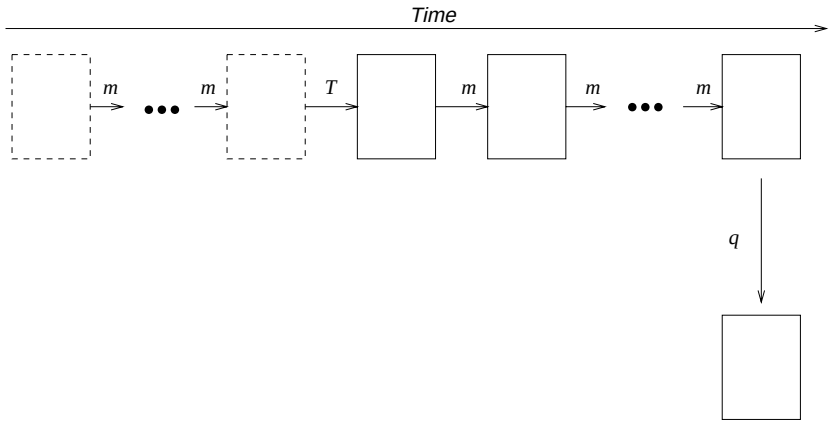

Figure 2: Temporal Upward Compatibility

The query q is an SQL–92 query. Due to temporal upward compatibility the semantics of this query must not change if it is applied to a temporal table. Hence, the query only applies to the current state, and a snapshot table results.

There is one unfortunate ramification to the above definition. Any extension that adds constructs involving new reserved keywords will violate upward compatibility, as well as temporal upward compatibility. The reason is that the user may have previously used that keyword as an identifier. Query language statements that use the keyword as an identifier will, in the extension, be disallowed.

Reserved words are added in all temporal query languages. This phenomenon also holds for non-temporal query languages. SQL–89 defined some 115 reserved words; SQL–92 added 112 reserved words, and the draft standard SQL3 adds another 97 reserved words.

To avoid being overly restrictive, we consider upward compatibility and temporal upward compatibility to be satisfied even when reserved words are added, as long as the semantics of all statements that do not use the new reserved words is retained in the temporal model.

3 Temporal Database Management Using SQL–92

As an initial application of these notions, we first consider an approach employed frequently to implement a temporal application: emulating a time-varying table with a conventional table. As we will see, this approach does not ensure temporal upward compatibility, leading to a number of difficulties.

The underlying model will be SQL–92, that is, QL_2 is the set of SQL–92 queries and modifications. Let us return to the `Employee` table, which has three columns, `Name`, `Manager`, and `Dept`. In SQL–92, one can require that all managers be employees, by stating that `Manager` is a foreign key for `Em-`

ployee.Name. We can easily express queries such as "List those employees who are not managers," as well as modifications, such as "Change the manager of the tools department to Bob."

To store historical information, we wish to emulate time-varying information, and so we will use the same model, SQL–92, that is, QL_T will be the set of SQL–92 queries and modifications. We also need an operator that changes the type of a snapshot table to a 'temporal' table. We will define $\mathcal{T}$ to be the following SQL–92 schema modification statements.

```
ALTER TABLE Employee ADD COLUMN Start DATE
ALTER TABLE Employee ADD COLUMN Stop DATE
```

The $\mathcal{T}$ operator must also initialize the value of the `Start` column to be the value `CURRENT_DATE` and the value of the `Stop` column to be the value `DATE '9999-12-31'`, the largest `DATE` value. This transforms the `Employee` table into a 'temporal' table (in the data model M_T, which is SQL–92).

Model M_T is certainly upward compatible with M_S, as all databases in $M_T = M_S =$ SQL–92. Interestingly, though, SQL–92, along with the transformation operator just defined, is *not* temporally upward compatible with itself. As but a simple example, let q_S be the query "`SELECT * FROM Employee`". It is certainly not the case that $\langle\!\langle q_S(db_S)\rangle\!\rangle_{\text{SQL}-92} = \langle\!\langle q_S(\mathcal{T}(db_S)\rangle\!\rangle_{\text{SQL}-92}$. Even the schemas do not match: the schema for the result of $\langle\!\langle q_S(db_S)\rangle\!\rangle_{\text{SQL}-92}$ has three columns, `Name`, `Manager`, and `Dept`, while the schema for the result of $\langle\!\langle q_S(\mathcal{T}(db_S)\rangle\!\rangle_{\text{SQL}-92}$ has five columns, including `Start` and `Stop`.

This violation of temporal upward compatibility has important practical ramifications. Assume that we have a 50,000-line application that manages the `Employee` table and other tables in a personnel database, allowing employees to be added and dropped, and the information about employees to be modified and queried in various ways. When this table is extended to store time-varying information, via the transformation $\mathcal{T}$ discussed above, many portions of this application break.

- The constraint that all managers are employees can no longer be expressed via SQL–92's foreign key constraint, which fails to take time into account. Instead, this constraint must be replaced with a complex assertion that includes in its predicate the `Start` and `Stop` columns.
- All queries must be examined, and most must be modified. Consider the query "List those employees who are not managers." A where predicate is now required to extract the *current* managers. Also, any query that mentions '`*`' must be modified, because the `Employee` table now has a different number of columns.
- Modifications must also be altered to take into account the `Start` and `Stop` columns. The modification "Change the manager of the tools department to

Bob" is now quite more involved than before.

In contrast, assume that instead of attempting to emulate the time-varying aspect using conventional tables, we use a temporal data model that is provably temporally upward compatible with SQL–92. We would be assured that *not a single line* of our 50,000-line application would have to be altered when transformation $\mathcal{T}$ was applied to render the `Employee` table time-varying.

4 Temporal Query Languages

As we just saw, the fact that an emulation of temporal tables using SQL–92 is not temporally upward compatible has several unfortunate ramifications in practice. We now turn to the temporal extensions to SQL that have been defined to date. Following an overview of the evaluation, we consider each temporal SQL in turn.

4.1 Overview of Temporally Extended SQL's

We are aware of seven temporal data models that extend SQL. We consider each of these in turn, starting with the earliest models, examining whether or not each model satisfies the requirements of upward compatibility (UC) and temporal upward compatibility (TUC) with respect to some variant of SQL, e.g., SQL–89, SQL–92, SQL3, or SQL dialects of commercial DBMSs.

Ideally, we prefer to be able to independently prove that a particular temporal data model satisfies or violates a requirement. However, the available documentation of the models often is not adequately comprehensive for this to be possible. With two exceptions, only the integration of the temporal query facilities with "core" subsets of SQL are documented, and which particular SQL dialect that is being extended is also not always mentioned. This makes it hard to determine whether models are (temporal) upward compatible with "the" full SQL or some subset of "an" SQL.

Aspects related to the use of regular SQL statements—modifications, in particular—on temporal tables or a combination of temporal and non-temporal tables are typically not defined. This makes it hard to verify temporal upward compatibility.

Finally, the definition of the syntax of several of the models is quite informal and incomplete. The semantics of the models are, at best, informal and, at worst, indicated by a few examples.

For the cases where we cannot prove that a temporal data model satisfies or violates a requirement, we will report the model as satisfying (or violating) a requirement if its designers claim that the property is satisified and we have not been able to disprove the claim with the available documentation. In addition, we will report satisfaction simply if we cannot prove dissatisfaction, again given the avail-

able documentation. Thus, we associate the following numbers with our findings, to indicate the confidence in the findings.

1. Neither satisfaction nor violation is claimed, nor can be proven.
2. Satisfaction claimed, but the claim cannot be proven nor invalidated.
3. Independently proven.

Clearly, the highest level of confidence is desired.

Table 1 gives an overview of our conclusions. As can be seen, different languages have different levels of satisfiability and different reasons for non-satisfiability. The first three models are documented rather sparsely for our purposes, but their designers emphasize that they satisfy upward compatibility. They do not satisfy temporal upward compatibility. The next model, TempSQL, introduces a concept of different types of users that may be used to obtain satisfaction of both compatibilities in certain circumstances. The subsequent model, IXSQL, is different from all the other models in that it does not provide support for implicit time; rather, it adds a parameterized abstract interval data type and associated facilities for modification and queries to SQL. ChronoSQL is one of the newer temporal models. The final model has been documented much more extensively than its predecessors, but its semantics are still given in an informal SQL-standards format. Note that no language satisfies both notions of upward compatibility.

4.2 Description of Temporally Extended SQL's

Next, we describe each of the temporally extended SQL's in some detail.

TOSQL

TOSQL [2] temporally extends a subset of an early version of SQL [1]. The extension is based on the TODM data model. The syntax of TOSQL is given in a BNF-like format. This syntax does not include modification statements, integrity constraints, nested queries, and queries involving aggregates using `HAVING`, etc. Hence, it appears that TOSQL is upward compatible with a *subset of* SQL, and is perhaps upward compatible with the full language.

It appears that the designer had a notion of temporal upward compatibility in mind when he wrote the following.

> "The default options are defined such that a query that omits the temporal portion retains the standard meaning of the corresponding SQL SELECT statement." [2, p. 513]

An example two pages later states the interpretation of a conventional SQL SELECT statement "is to specify that the query relates to *current* assignments, and uses the most up-to-date data about it." [2, p. 515]. The "current assignments"

Language	Reference	UC	TUC	Comments
TOSQL	[2]	yes[2]	no[3]	Only a subset of SQL is considered.
TSQL	[15] [16] [17]	yes[2]	no[3]	Not all snapshot tables can be made temporal. Some SQL views cannot be defined on temporal tables. Automatic coalescing violates TUC.
HSQL	[21] [22]	yes[2]	no[3]	SQL queries on temporal tables return temporal tables.
TempSQL	[3] [8] [9]	yes[2]	yes[3] (classical) no[3] (system)	Only a subset of SQL is considered. TUC is satisfied only for classical users. Means of specifying user types and defaults are not given.
IXSQL	[12] [10]	yes[2]	no[3]	Extension of SQL with a parameterized interval ADT with accompanying query-language facilities is proposed.
ChronoSQL	[5]	yes[3]	no[3]	Only a subset of SQL is considered. Does not restrict TUC queries to the current state.
TSQL2	[25]	yes[3]	no[3]	Full syntax given. Semantics defined informally in SQL-standard style.

Table 1: Summary of UC and TUC Compliance

refers to *now* in valid time; the "most up-to-date data" refers to *now* in transaction time.

The key phrase though is "that omits the temporal portion". The timestamp of a table in TOSQL appears as a column named RT. A non-time-varying table would not have such a column. The conversion operator $\mathcal{T}$ in Definition 3 would add this column. The problem is with queries involving '*'. Such queries on $\mathcal{T}(db_S)$ would return a different number of columns than queries directly on db_S. Hence, temporal upward compatibility is not satisfied.

TSQL

Navathe and Ahmed's *temporal relational model*, TSQL, supports, in addition to conventional tables, row timestamping for valid time by attaching two mandatory timestamp columns, *Time-start* (Ts) and *Time-end* (Te) to every time-varying relational schema [13, 15, 16, 17]. These timestamp columns correspond to the lower and upper bounds of time intervals in which rows are continuously valid.

It is stated that TSQL is upward compatible with SQL.

> "All legal SQL statements are also valid in TSQL, and such statements have identical semantics in the absence of a reference to time. [...] SQL, a subset of TSQL, remains directly applicable to non-time-varying relations in 1NF." [17, p. 99].

A simplified, 1.5 page BNF-like syntax is given for TSQL [15]. Statements such as updates, inserts, deletes, and view definitions are not addressed in the syntax or elsewhere in the documentation. Also, the use of regular SQL queries on temporal tables is not touched upon. While this makes it hard to examine the satisfaction of TUC, there are several indications that TUC is not satisfied.

In TSQL's data model, only tables that are in the so-called time normal form are allowed [15, p. 116]. Briefly, for a table to be in time normal form, it must be in Boyce-Codd normal form (disregarding the timestamp columns), and the non-key, non-timestamp columns must all be synchronous (i.e., they must change values simultaneously). As there are no such normal form requirements on snapshot tables, it follows that the $\mathcal{T}$ operator that turns a snapshot table into a temporal table is not defined for all snapshot tables. Also, regular SQL view definitions on temporal tables are not allowed when they lead to views that are not in time normal form. This is often the case for views that are joins.

Lastly, TSQL performs automatic coalescing of *value-equivalent* rows (i.e., rows with identical non-timestamp column values) that have consecutive or overlapping timestamps. This facility leads to a violation of TUC. For example, assume that we start out with an empty snapshot table, `R`, and insert two identical rows. Then `SELECT * FROM R` yields two rows. Now, we simultaneously insert the two rows into $\mathcal{T}$(`R`). The most reasonable assumption is that these two rows will be

given timestamps that result in them being coalesced into one row. Now, `SELECT * FROM` $\mathcal{T}$(`R`) yields one row.

HSQL

As the previous data model, Sarda's HDBMS also supports valid time; however, unlike the data model mentioned previously, HDBMS represent valid time in a valid-time table as a single non-atomic, implicit column [21, 22]. HSQL[1] is the query language of HDBMS.

It is emphasized that HSQL is upward compatible with respect to SQL (SQL–89, in fact).

> "HSQL is a superset of the popular query language SQL." [22, p. 123]
> "In fact, the standard clauses of SQL have identical meanings in HSQL." [22, p. 125]

Concerning TUC, the effects of the standard SQL insert, delete, and update statements are consistent with satisfying this requirement. However, a query `SELECT * FROM R` where `R` is a temporal table returns `R` and not the current (snapshot) state of `R`, as would be required in order to satisfy TUC [22, pp. 126–127].

TempSQL

Gadia's TempSQL is based on a N1NF temporal data model that is value timestamped [3, 8, 9]. A column of a row may have more than one (timestamped) value. The union of the timestamps of the values of each column must be the same for all columns throughout the entire row, resulting in a homogeneous temporal table.

Conventional tables are seen as temporal tables valid at a single time instant. Thus, each column value of each row in such a temporal table is timestamped with the same instant. Integration of snapshot tables into the data model this way is proposed partly in order to obtain upward compatibility.

> "By integrating it into our framework, we establish a smooth bridge for industry and its user community for migrating from classical databases to temporal databases. [...] We provide a framework for a smooth transition for industry, requiring no loss of investment in application programs developed by its user community." [9, p. 32]

The particular SQL that is being extended is not identified. No BNF is given. Further, only a subset of those facilities normally associated with SQL are mentioned, with several important aspects, e.g., advanced query facilities, integrity and

[1] In another paper, Sarda gave this extension to SQL the name TSQL [20]. We use HSQL because it was used in the most recent paper.

embedded queries, ignored. With these reservations, it is our contention that TempSQL is upward compatible with SQL. Determining whether temporal upward compatibility is satisfied is more difficult for this model than any of the other models.

TempSQL supports several types of users, e.g., system users and classical users, of a temporal DBMS. While system users have unrestricted access to the database, classical users can only access the currently valid values in the database. Thus, classical users see the current snapshots of temporal tables. Assuming that `T` is a temporal table, the query `SELECT * FROM T` returns `T` when issued by a system user and the current snapshot of `T` when issued by a classical user.

The absence of language syntax for specifying user types at the level of individual statements leads us to assume that, as indicated by the name, user types are fixed for individual users, and on a per-applications basis. (No information is given on how the mechanisms for different types of users interact with embedded application programs.) Had the intention been to be able to designate individual language statements as classical or temporal, we feel that the language should have provided syntax for this. We thus think about user types as being similar to ordinary SQL privileges. This seems reasonable, as user types do restrict access to data.

The choice of the default user type matters. If all users, and thus applications, are classical by default, then it is possible to avoid modifying the legacy applications when transitioning to a TempSQL system. Having the default user type be system leads to a violation of temporal upward compatibility—legacy applications then need to be modified to indicate that they are classical.

The next issue to consider is that of the application of legacy SQL modification statements on temporal tables. As the effects of such statements persist in the current states (i.e., the states of the temporal tables valid at the (ever-increasing) current time), the statements are consistent with TempSQL satisfying temporal upward compatibility.

Our conclusion is that for classical users, temporal upward compatibility is ensured. For system users, the opposite is true. The reason is that, for a system user, a conventional SQL query over a temporal table will return a temporal table.

TempSQL is thus fine when a non-temporal application is executed on a database that has been migrated to a temporal DBMS. Where TempSQL falls short is in further migration of that application, to exploit the very useful temporal constructs of that language. This requires that the user be a system user, because a classical user is not permitted to use any of the new constructs. As soon as the user transitions from classical to system, all of the query language statements in the application must be reevaluated, and many must be substantially rewritten. Had temporal upward compatibility been ensured for all users, this jarring transition would have been much smoother.

IXSQL

IXSQL [11, 12, 10] differs from all the other temporal query languages in that it does not provide support for a special, built-in notion of time. Rather, IXSQL adds the ability to define columns of a parameterized interval abstract data type, and it provides special query facilities for manipulating tables with rows that have such interval values.

Actually, there exists at least two different versions of IXSQL, an early version [11], and a later version [10]. The initial version was neither upward nor temporally upward compatible with SQL, in part because it did not permit duplicate rows in tables.

> "IXSQL actually differs from the standard SQL [reference to SQL–89], in that a relation may not contain duplicate tuples." [11, p. 4]

In the remainder, we consider the later version. This version was designed to be upward compatible with SQL–92:

> "IXSQL is syntactically and semantically upwards consistent with SQL2." [10, p. 1]

Next, we consider temporal upward compatibility. The first step is to decide on what the meaning of $\mathcal{T}$ should be in a model without an implicit notion of time in its tables. To be specific, let us simply assume that $\mathcal{T}$ adds an interval-valued column to each snapshot table, with value `[CURRENT_DATE, DATE '9999-12-31']` for each row. Other reasonable assumptions seem to lead to the same conclusions. The result of a legacy query such as `SELECT * FROM R` will differ from the result of `SELECT * FROM` $\mathcal{T}$`(R)`. In addition, legacy modifications to "temporal" tables will generally not be consistent with satisfying temporal upward compatibility, or they may fail altogether. In summary, legacy applications need to be rewritten when new columns are added to the tables then access.

ChronoSQL

ChronoSQL was designed and implemented as part of the ChronoLog project [5]. The main purpose was to illustrate how temporal concepts developed for deductive databases can be carried over to relational databases. ChronoSQL is tightly coupled with a Datalog-based language, which means that users can switch language any time.

This said, it comes as no surprise that not all language features of ChronoSQL have been worked out in detail. Specifically, the temporal extension was restricted to query statements; data manipulation statements and integrity constraints were not considered. Moreover, legacy queries over temporal tables are not restricted to the current state. This clearly violates temporal upward compatibility.

Upward compatibility looks more promising. ChronoSQL adds a couple of non-mandatory syntactic constructs to SQL. No other syntactic changes are proposed. This ensures syntactic upward compatibility. Furthermore, the semantics of legacy statements over nontemporal tables remains unchanged [5, p.69], meaning that upward compatibility is ensured as well.

TSQL2

TSQL2 [25] is the most comprehensively documented temporal query language. Its syntax was given as an extension of the syntax of SQL–92 as presented in the official standard, and the semantics of TSQL2 was also given in the format of the SQL–92 standard. Some 500 pages of technical commentaries accompany these specifications. Upward compatibility of TSQL2 is studied in [4].

In TSQL2, there are six kinds of tables: snapshot tables, valid-time event tables, valid-time state tables, transaction-time tables, bitemporal event tables, and bitemporal state tables. The first is the kind of table found in the relational model; the remaining five are temporal tables. As all the schema specification statements of SQL–92 are included in TSQL2, it follows that the data structures of TSQL2 include those in SQL–92.

TSQL2 is also a strict superset of SQL–92 in its query facilities. In particular, if an SQL–92 select statement does not incorporate any of the constructs added in TSQL2, and mentions only snapshot tables in its from clause(s), then the language specification states explicitly that the semantics of this statement is identical to its SQL–92 semantics.

It should be noted that the preliminary TSQL2 language specification released in March, 1994 [19] did not have that property. In particular, SQL–92 `INTERVAL`s were termed `SPAN`s in the preliminary TSQL2 specification, and TSQL2 `INTERVAL`s were not present at all in SQL–92. The final TSQL2 language specification [25] retained SQL–92 `INTERVAL`s and added the `PERIOD` data type, which was previously called `INTERVAL` in preliminary TSQL2 (confusing, isn't it?). Additional changes to the datetime literals were also made to ensure that TSQL2 was a strict superset of SQL–92.

Hence, TSQL2 is upwards compatible with SQL–92. However, TSQL2 is not temporally upward compatible with SQL–92, for several reasons. First, SQL–92 tables that contain duplicates have no counterparts in TSQL2 where tables with value-equivalent rows (and thus duplicates, either in a timeslice, or in the temporal table itself) are not allowed. A second reason that TSQL2 is not temporally upward compatible with SQL-92 is that when the keyword `SNAPSHOT` is *not* specified in a select statement in TSQL2, a temporal table results. Hence, an SQL–92 query over a temporal table will result not in a conventional table, but rather in a temporal table.

5 Ensuring Temporal Upward Compatibility

This section explains a sequence of steps that lead to a temporal upward compatible SQL–92 extension. Implications to syntax and semantics are discussed and illustrated with examples. Temporal upward compatible extensions allow to independently migrate data structures and application code. Specifically, it permits migration of data structures without also requiring changes to application code (c.f. Definition 3). The examples that have been stated in prose in Section 2 are reconsidered and formulated in the temporal extension of SQL–92.

5.1 Syntax of a Temporal Upward Compatible Extension of SQL

Temporal upward compatibility does not put an upper limit on syntactic extensions to a language. It, however, defines a lower limit. First, all legacy statements must be retained. (This requirement is independently established by upward compatibility.) Second, a possibility must be provided to migrate nontemporal data structures to temporal data structures. The first requirement is met by adding (non-mandatory!) syntactic constructs to the base language. No syntactic constructs may be deleted or changed. Migrating non-temporal to temporal data structures can be achieved in different ways. We discuss two possibilities to illustrate the design space and the possible consequences to the data model.

If we want to emphasize different table types (snapshot tables, valid time tables, transaction time tables, and bitemporal tables) a reasonable syntactic choice is to extend the <alter table action> production of SQL–92 [14, p.511], by adding two options.

<alter table action> ::= <add column definition>
| <alter column definition>
| <drop column definition>
| <add table constraint definition>
| <drop table constraint definition>
| <add time dimension>
| <drop time dimension>

<add time dimension> ::= ADD <time dimension>

<drop time dimension> ::= DROP <time dimension> <drop behavior>

<time dimension> ::= VALID | TRANSACTION

Adding valid time turns a snapshot table into a valid time table and a transaction time table into a bitemporal table. Adding transaction time turns a snapshot

table into a transaction time table and a valid time table into a bitemporal table. This is the approach chosen by TSQL2 [25].

If instead we want to emphasize the conventional relational data model with tables that support time through special-purpose columns, an alternative approach would be to enhance the productions <add column definition> and <drop column definition> respectively.

<add column definition> ::= ADD [COLUMN] <column definition>
| ADD [COLUMN] <time dimension>

<drop column definition> ::=
DROP [COLUMN] <column name> <drop behavior>
| DROP [COLUMN] <time dimension> <drop behavior>

Further syntactic alternatives can also be envisioned. It is, however, critical that all of them support the semantics discussed in the next section.

5.2 Semantics of a Temporal Upward Compatible Extension of SQL

This section discusses the semantics of various temporally upward compatible statement categories, i.e., standard SQL–92 statement categories evaluated over temporal databases. The categories include queries, views, assertions, column constraints, referential integrity constraints, insertions, deletions, and updates. This ensures a broad coverage of the functionality of a database system. Nevertheless, there are certain statement categories that are not considered explicitly, e.g., triggers. These categories do not introduce fundamentally new problems with respect to temporal upward compatibility. Instead, semantics and techniques discussed for other categories can be applied directly.

When we discuss the semantics of legacy statement categories over temporal tables we can differentiate between *non-destructive statements*, e.g., queries, views, and integrity constraints, and *modification statements*, e.g., data manipulation statements. As we will see, these two sets of categories have to be treated differently.

Below we discuss the semantics for each of the two sets of categories. Within each set all categories are analyzed and illustrated with an example. We initially consider only valid time, then discuss the impact of adding transaction-time support.

The very first step is of course to migrate the data structures.

```
ALTER TABLE Employee ADD VALID
ALTER TABLE Salary ADD VALID
```

Both tables are turned into valid-time tables, such that all information stored in the tables can be annotated with its valid time (transaction time is discussed at the end of this section).

Non-destructive Valid-time Statements

Non-destructive statements retrieve from or check parts of the database. They do not change the contents of the database. To get the exact same semantics that a nontemporal database would provide, we have to restrict the retrieval and checking to the current state.

Queries are supported by adding an implicit selection condition to the `WHERE` clause that selects current rows. Moreover, defaults, e.g., '`*`' in the select clause, may not expand to include time. As an example, assume a query that determines who manages the high-salaried employees. The 'temporal' query is straightforward.

```
SELECT Manager
FROM   Salary AS S, Employee AS E
WHERE  S.Name = E.Name
AND    S.Amount > 3500
```

Whenever the temporal database system identifies one or more temporal in an SQL–92 statement, it must perform the actions dictated by temporal upward compatibility. In this case, it must restrict the set of rows to the current ones.

Views are similar to queries. This becomes obvious if we remember that a view is a virtual table defined by a query. The query that defines the view is enhanced along the lines outlined above. As an example, consider a view that yields high-salaried employees.

```
CREATE VIEW High_salary AS
  SELECT *
  FROM Salary
  WHERE Amount > 3500;
```

A selection condition that limits the query expression to current salaries has to be added. Moreover, the default used in the select clause has to be extended to `SELECT Name, Amount` (or an equivalent relational algebra projection) so that the valid time is not part of the result.

Integrity constraints come in different flavors. The most general form are assertions [14, p.211ff]. Consider the assertion that ensures that all employees get a salary, i.e., an assertion that checks that no employees without a salary exist.

```
CREATE ASSERTION CONSTRAINT Emp_has_sal CHECK
NOT EXISTS (SELECT *
            FROM Employee AS E
            WHERE NOT EXISTS (SELECT *
                              FROM Salary AS S
                              WHERE
                               E.Name = S.Name))
```

The general approach to check an assertion is to negate it and to execute it as a query, i.e.,

```
SELECT *
FROM Employee AS E
WHERE NOT EXISTS (SELECT *
                  FROM Salary AS S
                  WHERE E.Name = S.Name)
```

If the query result is empty, i.e., if no rows are returned, the assertion is respected; otherwise it is violated. With this background, temporal upward compatible assertions can be achieved easily, because we showed above how to do so with queries.

Modification Statements on Valid-time Tables

Modification statements change the contents of the database. An obvious (but naive) approach is to carry over the semantics from the previous section and to modify the current state. Imagine the insertion of an employee into the database.

```
INSERT INTO Employee
VALUES ('Liliane', 'Brandt', 'Tools')
INSERT INTO Salary
VALUES ('Liliane', 1000)
```

If we inserted Liliane only in the current state, subsequent queries would not return this row. When we later issue a query, time will have progressed and Liliane will no longer be in the (new) current state. Of course this is not the behavior we expect from a nontemporal database. In order to get the expected behavior, we have to make sure that Liliane remains in the changing current state. This may be achieved by using the period from `CURRENT_DATE` to `9999-12-31` (the largest `DATE` value) as the timestamp of Liliane's tuples. But it may also be achieved using as the end point `NOBIND(CURRENT_DATE)`, where `NOBIND` has the effect of storing in the timestamp a *variable* that evaluates to `CURRENT_DATE` when accessed, rather than storing the current value of `CURRENT_DATE`. Indeed, any now-relative variable [6] that evaluates to a time between these two end points may be used. We will adopt the simplest choice, the date `9999-12-31`.

An equivalent observation holds for delete and update statements. Assume that we want to change the manager of the tools department to Bob.

```
UPDATE Employee
SET    Manager = 'Bob'
WHERE  Dept = 'Tools'
```

If we only updated the current state, subsequent queries would not access the corrected database state. Again, we have to ensure that the update persists in the changing current state to get the exact same behavior a nontemporal database provides.

Achieving temporal upward compatibility for modification statements is slightly more complicated than achieving temporal upward compatibility for non-modification statements. The reason is that certain rows may be valid from some point in the past until some point in the future, i.e., they overlap the current time. Because temporal upward compatible statements only affect the current and future times, the modifications must not change the row during the entire time range. Let us consider each type of modification statement in turn.

Insert statements have to set the valid-time start to the current time and the valid-time end to `DATE '9999-12-31'`, as discussed above. This ensures that, until the row is deleted or modified, it will be valid.

Next we consider delete statements. Historical data, i.e., qualifying rows with a valid time end before the current time, is left untouched. Current data, i.e., qualifying rows with a valid-time start after the current time (including a valid time end equal to `DATE '9999-12-31'`), has to be deleted as of the current time. This is done by changing valid time end to the current time. For future knowledge two choices exist. If we decide not to delete it, today's future knowledge will become valid eventually. This behavior can be quite surprising for applications employing temporal upward compatibility exclusively. An alternative is to delete qualifying future knowledge. This ensures a more intuitive behavior of legacy applications, but it might not be the semantics temporal applications envision.

The most complex statements are update statements. First, rows with a valid-time start before the current time and a valid-time end after the current time (including a valid-time end equal to `DATE '9999-12-31'`) are duplicated. The valid-time end of the original row and the valid-time start of the duplicated row are set to the current time. Then the update statement is applied to all rows with a valid time start that is equal or after the current time. Again we have the choice not to update future knowledge (c.f. previous paragraph).

Transaction Time

With respect to temporal upward compatibility, transaction time behaves almost identically to valid time. Exactly the same semantics applies to transaction-time tables and valid-time tables.

Even bitemporal tables behave quite similarly. In non-destructive statements and insertions, both time dimensions inherit the unitemporal semantics. Deletions and updates are somewhat more complicated, due to the nature of transaction time which guarantees that at each point in time, it is possible to reconstruct previous database states. A temporal upward compatible deletion of a bitemporal row triggers the following steps.

1. Qualifying rows with a transaction-time end equal to `9999-12-31` are duplicated. The transaction-time end of the original row and the transaction-time

start of the duplicated row are set to the current time.

2. The valid-time deletion is applied to qualifying rows with a transaction-time end equal to `9999-12-31`.

The first step saves the current state and thus ensures reconstructability, whereas the second step performs the valid-time deletion. Update follows a similar pattern.

6 Conclusion

Upward compatibility aids in the smooth migration of applications from a conventional to a temporal data model. The definitions introduced here allow a specific temporal language to be evaluated as to the degree that it ensures upward compatibility. The extant temporal extensions to SQL are all deficient in one or more ways, rendering migration more difficult. We subsequently showed how SQL–92 can be extended to yield a temporal data model satisfying all three notions of upward compatibility. Applications can be much more easily migrated to this new data model.

The notion of temporal upward compatibility can be viewed as a form of logical data independence. In the same way that an external schema can ensure that applications are not impacted by changes to the logical schema, temporal upward compatibility ensures that applications are not impacted by a specific kind of change to the logical schema: adding or removing temporal support. Logical data independence is an important benefit provided by modern data models, in particular by the relational data model, and the specific kind discussed here provides similar advantages.

The approach we espouse here to providing temporal upward compatibility relative to SQL was adopted in the SQL/Temporal proposals [23, 24]. These language constructs were explicitly designed to ensure upward compatibility *and* temporal upward compatibility with the entire SQL–92 standard. The constructs have been proposed to the American ANSI and international ISO SQL committees for inclusion into the next ISO SQL standard.

Several directions for further research are promising. First, there is a need for exploring different implementation alternatives for upward compatible temporal SQL extensions. Alternatives range from stand-alone implementations to implementations that maximally reuse the functionality offered by existing DBMS's with an SQL interface. Second, it is felt that much could be learned from conducting actual case studies of the migration of legacy applications to temporal platforms. Third, the transition from explicit to implicit temporal knowledge should be investigated. Strategies must be designed to assist the user in migrating nontemporal tables with explicit time columns to temporal tables. This is essential to maximally exploit the capabilities of temporal database systems.

7 Acknowledgments

M. H. Böhlen and C. S. Jensen were supported in part by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056. R. T. Snodgrass was supported in part by NSF grants ISI-9202244 and ISI-9632569 and by a grant from DuPont.

References

[1] M. M. Astrahan and D. D. Chamberlin. Implementation of a Structured English Query Language. *Communications of the ACM*, 18(10):580–588, October 1975.

[2] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[3] G. Bhargava and S. K. Gadia. Relational database systems with zero information loss. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):76–87, February 1993.

[4] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating the Completeness of TSQL2. In *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*, pages 153–172, Zürich, Switzerland, September 1995. Springer, Berlin.

[5] M. Böhlen. *Managing Temporal Knowldege in Deductive Databases*. PhD thesis, Departement für Informatik, ETH Zürich, Switzerland, 1994.

[6] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "NOW" in Temporal Databases. *ACM Transactions on Database Systems*, to appear 1997.

[7] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[8] S. K. Gadia and G. Bhargava. SQL-like Seamless Query of Temporal Data. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, Texas, June 1993.

[9] S. K. Gadia and S. S. Nair. *Temporal Databases: A Prelude to Parametric Data*, chapter 2 of [26], pages 28–66. 1993.

[10] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, to appear 1996.

[11] N. Lorentzos. Management of Intervals and Temporal Data in the Relational Model. Technical Report 49, Agricultural University of Athens, 1991.

[12] N. Lorentzos. *The Interval-extended Relational Model and Its Application to Valid-time Databases*, chapter 3 of [26], pages 67–91. 1993.

[13] N. G. Martin, S. B. Navathe, and R. Ahmed. Dealing with temporal schema anomalies in history databases. In P. Hammersley, editor, *Proceedings of the Thirteenth International Conference on Very Large Databases*, pages 177–184, Brighton, England, September 1987.

[14] J. Melton and A. R. Simon. *Understanding the new SQL: A Complete Guide*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[15] S. B. Navathe and R. Ahmed. TSQL - A Language Interface for History Databases. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, pages 113–128. AFCET, May 1987.

[16] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Information Systems*, 49(2):147–175, 1989.

[17] S. Navathe and R. Ahmed. *Temporal Extensions to the Relational Model and SQL*, chapter 4 of [26], pages 92–109. 1993.

[18] G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

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

[20] N. Sarda. Algebra and Query Language for a Historical Data Model. *IEEE Computer Journal*, 33(1):11–18, February 1990.

[21] N. Sarda. Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):220–230, June 1990.

[22] N. Sarda. *HSQL: A Historical Query Language*, chapter 5 of [26], pages 110–140. 1993.

[23] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. ANSI X3H2-96-151r1, ISO–ANSI SQL/Temporal Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL MCI-142, May 1996.

[24] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. ANSI X3H2-96-152r, ISO–ANSI SQL/Temporal Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL MCI-143, May 1996.

[25] R. T. Snodgrass (editor). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.

[26] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/-Cummings Publishing Company, Inc., Redwood City, California, 1993.

[27] V. J. Tsotras and A. Kumar. Temporal Database Bibliography Update. *SIGMOD Record*, 25(1):41–51, March 1996.

[28] D. C. Tsichritzis and F. H. Lochovsky. Data models. In *Software Series*. Prentice-Hall, 1982.