Seamless Integration of Time into SQL

Michael H. Böhlen Christian S. Jensen

Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK–9220 Aalborg Øst, DENMARK
<boehlen | csj>@cs.auc.dk

Abstract

A wide range of database applications manage time-varying data. Although temporal database technology has reached a level of maturity and sophistication where it is evident that these applications may benefit substantially from built-in temporal support in the database management system, these applications typically run on top of conventional relational systems. This state of affairs may be explained by the fact that it is not clear how, or even "that," it is possible to smoothly migrate from a non-temporal to a temporal DBMS. Enterprises generally maintain bulks of legacy application code that must be dealt with appropriately.

This paper investigates how to smoothly migrate from a conventional relational system to a temporal system, a topic that has only received scant attention so far, but nonetheless is vital in order to unlock the potential of temporal technology for application in practice. At the outset, important requirements to a temporal system that may facilitate a smooth transition are motivated, formalized, and discussed. No temporal data model and query language satisfies the requirements. It is then demonstrated how it is possible to temporally extend SQL–92 while fulfilling each of the requirements. It is emphasized how the requirements shape the language, termed ATSQL. ATSQL is formally defined via a denotational-semantics-style mapping to well-defined algebraic expressions. Specifically, a temporal relational algebra along with a correct mapping from ATSQL to a combination of this and the relational algebra is given. A prototype implementation of ATSQL is publicly accessible.

1 Introduction

A wide variety of prominent applications manage substantial amounts of time-varying data. They include financial applications such as portfolio management, budgeting, accounting, and banking; record-keeping applications, such as personnel, medical-record, insurance policies, and inventory; and they include travel applications such as airline, train, and hotel reservations and schedule management.

Thus, numerous database applications manage substantial quantities of time-varying data. This has held true for as long as databases have been maintained [Wie73, Sno90]. Along with the continued improvement of storage technologies and new, data-intensive applications such as decision support and data warehousing, old versions of data are retained longer in the databases. This yields very large databases with all data exhibiting a prominent temporal dimension.

In stark contrast, conventional relational database technology provides only little support for temporal data management and is incapable of exploiting the time dimension to achieve better performance. In response to this unfulfilled potential for improvement, much work on temporal database management has been conducted over the past decade or two, leading to, e.g., a wide variety of data models and query languages and to numerous performance-enhancing

implementation techniques. Recent query languages (e.g., IXSQL [Lor93, LM96], TempSQL [Gad88], and TSQL2 [Sno95]) demonstrate that temporal application development may benefit substantially from built-in temporal support in the query language.

However, whether or not temporal database technology will gain wide acceptability in practice is not only determined by the availability and quality of temporal languages and features galore, but also (and perhaps mainly!) by the ease of transitioning from the existing technology. Despite its importance, this issue has never been discussed thoroughly. In this paper, we identify crucial transitioning requirements.

We assume that prospective users of temporal database technology are already users of non-temporal database technology—relational technology, to be specific—and must protect large investments in bulks of legacy code. It is thus of essence that legacy code remains operational when transitioning to a temporal database system; otherwise, adoption of temporal technology may not be feasible. Briefly, the first requirement, *upward compatibility*, guarantees that replacing the existing DBMS with a new, temporal DBMS does not affect the functioning of any application code.

When transitioning to a temporal system, the advantages of that system are not harvested instantaneously, but only incrementally, as new code is developed and legacy code revised. Hence, it is important that old and new code can coexist harmoniously. This occurs when existing applications are insensitive to their underlying relations being changed to become temporal. The requirement, *temporal upward compatibility*, guarantees exactly that.

In combination, the two requirements mentioned so far provide protection of the large investments in legacy code while allowing for a gradual exploitation of the new database technology. Next, the temporal extension should exploit the programmers' familiarity with the existing system. This may be accomplished by offering, for each non-temporal query, a syntactically similar, temporal statement that "naturally" generalizes the non-temporal query to yield a temporal result. The requirement that the temporal data model be a *syntactically similar, snapshot-reducible* extension of the existing data model guarantees that a core subset of the temporal data model maximally builds on the existing data model, making the temporal query language easy to use for programmers familiar with the existing query language. This requirement thus aims at protecting the investments in programmer expertise and training.

When developing a general-purpose temporal data model and query language, two temporal aspects of data attract special attention. The *valid time* of a database fact (e.g., a tuple) is the times when the fact was or will be true in the modeled reality. The *transaction time* of a database fact is the times when the fact has been stored as current in the database. All database facts have a valid time and a transaction time, and we consider both of these times in this paper. However, there is no requirement that a database explicitly records either of these aspects for its facts. We will use the modifiers *temporal* or *time-varying* for databases if one or both of valid and transaction time are associated with their facts.

The paper shows how a systematic and comprehensive temporal extension, ATSQL, of the current SQL standard may be designed so that it satisfies the requirements. ATSQL, in addition to select statements, includes data definition of relations with duplicates (in the sense of SQL–92); it includes modification statements; and it includes integrity constraints.

Three complementing descriptions of the language are provided. First, it is shown how the requirements shape the skeleton of ATSQL. Second, a guided tour (that runs on the prototype implementing ATSQL) is included to convey a practical feel for the features of the language and their underlying concepts. Third, a formal definition of the semantics of the query language is provided by means of a denotational-semantics-style mapping to well-defined algebraic expressions. This mapping assumes a mapping of SQL–92 to relational algebra and defines ATSQL statements in terms of their mapping to well-defined relational and temporal relational algebra expressions. The temporal relational algebra used here is efficiently implementable in that the

evaluation of its expressions relies only on the end points of periods and not on intermediate points, making evaluation granularity independent.

Upon having defined ATSQL, its properties are subjected to scrutiny. First, it is shown that its definition indeed satisfy the migration-related requirements. Second, the additional properties given to the language are considered. In particular, it is argued that the language is suitable for managing both so-called point-based and interval-based temporal data. Coalescing is available for point-based data management; and without coalescing, the language is faithful to, or respects, the particular periods associated with the tuples.

A prototype system implementing ATSQL is accessible from Aalborg University's web server, via URL `http://www.cs.auc.dk/general/DBS/software`.

The paper is structured as follows. The next section is concerned with the formulation of the requirements to a temporal data model that, when satisfied, will guarantee a smooth transition of legacy applications from a non-temporal DBMS to a temporally enhanced DBMS. Section 3 then proceeds by illustrating how the current SQL standard my be systematically enhanced to provide built-in support for temporal database management. First, it is shown how the requirements from Section 2 shape the design of the temporal SQL, termed ATSQL. Then, a guided tour illustrates the concepts and language features of ATSQL. Having provided the rationale and intuition behind the language design, Section 4 gives a concise and yet precise and comprehensive semantics for the language. This provides a solid footing for the exploration of language properties—the topic of Section 5. Related research is explored in some detail in Section 6. Finally, Section 7 summarizes and points to promising opportunities for future research. A number of appendices with detailed technical matter complete the paper.

2 The Smooth Migration to a Temporal DBMS

Initially, an overview and a description of the assumed context is given. The subsequent sections explore the problems that may occur when migrating database applications from an existing to a new temporal DBMS, and they precisely formulate a number of requirements to the temporal DBMS that must be satisfied to facilitate a smooth migration.

2.1 Overview and Context

The potential users of temporal database technology are enterprises with applications[1] that manage potentially large amounts of time-varying data. These applications may benefit substantially from built-in temporal support in the DBMS. Temporal queries that are shorter and more easily formulated are among the potential benefits. This leads to improved productivity, correctness, and maintainability. It is also a matter of fact that these enterprises are already managing time-varying data, with the applications already in place and working. Indeed, the uninterrupted operation of the existing applications is likely to be of vital importance to any enterprise. The question is then how the enterprise can smoothly migrate from its current DBMS to a temporal DBMS.

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 will adopt the convention that a data model consists of two components, namely a set of data structures and a language for querying the data structures [TL82]. For example, the central data structure of the relational model is the relation, and the central, user-level query language is SQL. As we progress, it should be clear that the definitions and discussions of this section

[1] We use "database application" nonrestrictively, for denoting any software system that uses a DBMS as a standard component.

also apply to views and 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 update and query 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 an update *u* (in SQL–92, any modification, i.e., `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 [MS93] to (some) temporal SQL.

2.2 Upward Compatibility

Perhaps the most important aspect of ensuring a smooth migration of application code is to guarantee that all code without modification will work with the new model, exactly with the same functionality as with the existing system. The next two definitions capture the essence of what is needed for that to be possible.

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

Definition 2.1 (syntactical upward compatibility) 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 iff

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

When transitioning from one system to a new system, it is important that the new data model contains the existing data model. If that is the case, all existing application code will remain syntactically correct.

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. In addition to the previous syntactical requirement, we add the requirement that all queries expressible in the existing model must evaluate to the same results in the existing and new models.

Definition 2.2 (upward compatibility) 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 iff

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

This concept captures the conditions that need to be satisfied in order to allow a smooth transition from a current system, with data model M_2, to a new system, with data model M_1. The first condition implies that all existing databases and query expressions in the old system

are also legal databases in the new system. The second condition guarantees that all existing queries 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.

There is one unintended ramification of the above definition. Any temporal extension that includes new reserved keywords will violate upward compatibility. The reason is that legacy query language statements may have employed such keywords as identifiers. Under the semantics of the new model, such statements will be illegal. Reserved words are added in all temporal query languages, and it is impractical to exclude them. This also holds for non-temporal query languages. For example, SQL–92 added some 112 reserved keywords to the 115 reserved keywords of its predecessor, SQL–89[2]. One proposed solution is to use *quoted* identifiers in legacy code where identifiers conflict with new reserved keywords and in new code, to avoid future problems. In conclusion, to follow current practice and to avoid being overly restrictive, we consider upward compatibility to be satisfied even when new keywords are added in the extension.

To explore the relationship between SQL–92 and a temporal extension of it that satisfies the requirements, a series of figures will be introduced. In Figure 1, a conventional relation is denoted with a rectangle. The current state of this relation is the rectangle in the upper-right corner. Whenever a modification is made to this relation, 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 updates that take the relation from one state to the next.

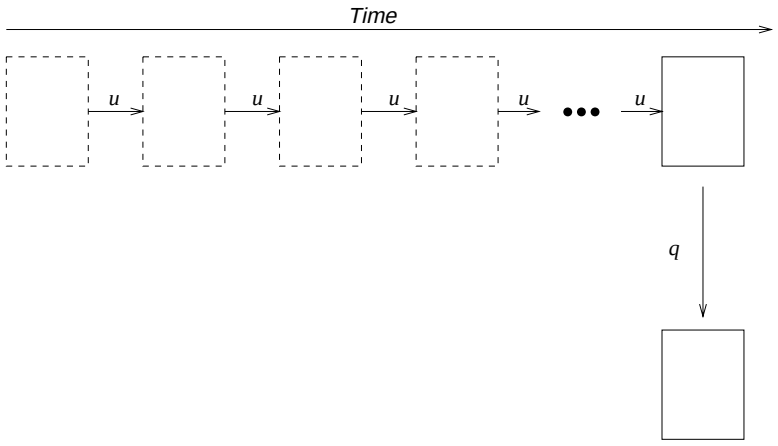


Figure 1: Evaluation of an SQL–92 Query on a Snapshot Relation

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

Upward compatibility states that (1) all instances of relations in SQL–92 are instances of relations in the temporal extension, (2) all SQL–92 modifications to relations in SQL–92 result in the same relations when the modifications are performed by the temporal system, and (3) all SQL–92 queries result in the same relations when the queries are evaluated by the temporal system.

By requiring that a temporal extension is a strict superset (i.e., only *adding* constructs and semantics), it is relatively easy to ensure that the temporal extension is upward compatible with SQL–92. Still, it should be noted that upward compatibility does place strict constraints on the

[2]Reference [MS93] provides a list of 10 items with incompatibilities among SQL–89 and SQL–92, with the keyword aspect being one item.

temporal extension. The temporal language must be "in the spirit" of and must live with any peculiarities of the language it extends. As a simple example, when extending SQL–92—which does not have an interval data type—with a data type for intervals, the string "interval" cannot be used in the syntax because this string is already used for the data type of durations.[3]

With each of the requirements that we present, we associate a subset of the functionality of a temporal extension of SQL–92. This leads to a division of the functionality of a temporal extended model into four levels, with each level adding functionality to the previous part. With upward compatibility, we associate the SQL–92 subset of the temporal extension. As the requirements are defined, we characterize the consequent levels.

2.3 Temporal Upward Compatibility

While essential, upward compatibility is only a first step. Upon adopting a temporal data model, the benefits of the built-in temporal support are only realized incrementally, by modifying existing application code or developing new application code that exploits the temporal capabilities. A next step is thus to formulate requirements that aim at ensuring a harmonious coexistence of legacy application code and new, temporally-enhanced application code.

There is no friction between existing code and new code if disjoint sets of relations are used. Rather, the potential problem occurs when new application code needs to use a combination of existing and new, temporal relations or needs to change existing relations to become temporal. To understand the problem, assume that the new temporal model is in place. No application code has been modified, and all relations are thus snapshot relations. Now, an existing or new application needs support for the temporal dimension of the data in one of the existing relations. To accommodate this need, there are two options.

The first is to create a completely new temporal relation. At first, this appears to not affect existing application code, but considering update reveals a problem. When the existing application updates the snapshot relation, those updates should also be reflected in the temporal relation. The new application cannot know when the existing application updates the snapshot relation, so the only solution is to modify the existing application to also update the temporal relation when the snapshot relation is updated. In addition, this possibility introduces data redundancy that may lead to consistency problems and may adversely affect performance.

The second possibility is to change the existing snapshot relation to become a temporal relation (using the `alter` statement of SQL–92). This alternative is attractive because it avoids replicating data and because the modifications (relevant) to existing application code (i.e., the `alter` statement) are isolated and kept at a minimum. But note that it is very undesirable to be forced to change the (legacy) application code that accesses the snapshot relation that is replaced by a temporal relation.

Based on this observation, we formulate a requirement stating that the existing applications on snapshot relations must continue to work unmodified when the relations are altered to become temporal. Intuitively, the requirement is that a query q must return the same result on an associated snapshot database db as on the temporal counterpart of the database, $\mathcal{T}(db)$. Further, updates should not affect this. The precise definition is given next and is explained in the following.

Definition 2.3 (temporal upward compatibility) Let a temporal and a snapshot data model be given by $M_T = (DS_T, QL_T)$ and $M_S = (DS_S, QL_S)$, respectively. Also, let $\mathcal{T}$ be an operator that changes the type of a snapshot relation to the temporal relation with the same explicit attributes. Next, let $\mathcal{U} = u_1, u_2, \ldots, u_n$ $(n \geq 0)$ denote a sequence of update operations. With these definitions, model M_T is *temporal upward compatible* with model M_S iff

[3] This is the reason why we generally use the SQL3 term "period" for intervals.

- M_T is upward compatible with M_S and
- $\forall db_S \in DS_S\ (\forall \mathcal{U}\ (\forall q_S \in QL_S\ (\langle\!\langle q_S(\mathcal{U}(db_S))\rangle\!\rangle_{M_S} = (\langle\!\langle q_S(\mathcal{U}(\mathcal{T}(db_S)))\rangle\!\rangle_{M_T}))))$. ∎

Figure 2 illustrates the idea in the requirement. With the temporal data model in place, the results of queries (i.e., q_S) in legacy code that also includes updates (as indicated by $\mathcal{U}$) are not affected by (hence the "=") changes to the type of relations (the transformation $\mathcal{T}$); and update statements on the new relations have the same effects on query results as the corresponding update statements on the original relations.

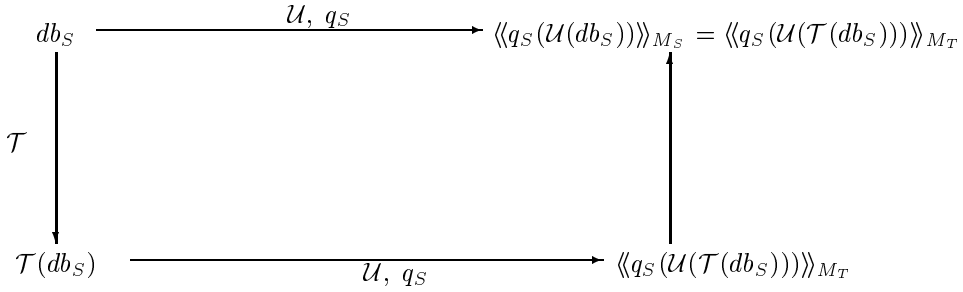


Figure 2: Illustration of Temporal Upward Compatibility

Using the graphical notation introduced in the previous section, temporal upward compatibility is illustrated in Figure 3. When temporal support is added to a relation, the history is preserved, and modifications over time are retained. In this figure, the rightmost dashed state was the current state when the relation 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 current states will have identical contents to those states resulting from modifications of the snapshot relation.

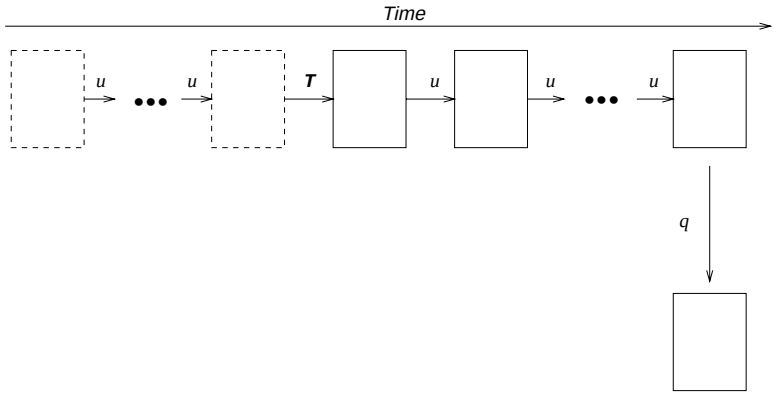


Figure 3: Evaluation of an SQL–92 Query According to Temporal Upward Compatibility

The query q is an SQL–92 query. Due to temporal upward compatibility, the semantics of this query must not change when it is applied to a temporal relation.

The subset of the functionality of a temporal data model that corresponds to temporal upward compatibility consists of all SQL–92 language constructs, the ability to create temporal relations, and the ability to apply SQL–92 queries and updates to temporal relations.

It is instructive to consider temporal upward compatibility in more detail. When designing larger information systems, two general approaches have been advocated. In the first approach, the system design is based on the *function* of the enterprise that the system is intended for (the "Yourdon" approach [You82]); in the second, the design is based on the *structure* of the reality that the system is about (the "Jackson" approach [Jac83]). It has been argued that the latter approach is superior because structure may remain stable when the function changes while the opposite is generally not possible. Thus, a more stable system design, needing less maintenance, is achieved when adopting the second design principle. This suggests that the data needs of an enterprise are relatively stable and only change when the actual business of the enterprise changes.

Enterprises currently use non-temporal database systems for database management, but that does not mean that enterprises manage only non-temporal data (a comprehensive list of temporal-data management applications was given earlier). Temporal data may be accommodated by non-temporal database systems in several ways. For example, a pair of explicit time attributes may encode a valid-time period associated with a tuple.

Temporal database systems offer increased user-friendliness and productivity, as well as potentially significantly better performance, when managing temporal data. The typical situation, when replacing a non-temporal system with a temporal system, is one where the enterprise is not changing its business, but wants the extra support offered by the temporal system for managing its temporal data. Thus, it is atypical for an enterprise to suddenly desire to record temporal information where it previously recorded only snapshot information. Such a change would be motivated by a change in the business.

The typical situation is likely to be rather more complicated. The non-temporal database system is likely to already manage temporal data, which is encoded using snapshot relations, in an ad hoc manner. When adopting the new system, upward compatibility guarantees that it is not necessary to change the database schema or application programs. However, without changes, the benefits of the added temporal support are also limited. Only when defining new relations or modifying existing applications, can the new temporal support be exploited. The enterprise then gradually benefits from the temporal support available in the system.

Nevertheless, the concept of temporal upward compatibility is still relevant, for several reasons. First, it provides an appealing, intuitive notion of a temporal relation: the semantics of queries and modifications are retained from snapshot relations; the only difference is that intermediate states are also retained. Second, in those cases where the snapshot relation contained no temporal information, temporal upward compatibility affords a natural means of migrating to temporal support. In such cases, not a single line of the application need be changed when the relation is made temporal. Third, snapshot relations that do contain temporal information and that have been converted to temporal relations can still be queried and modified by conventional SQL–92 statements in a consistent manner.

In summary, with temporal upward compatibility, existing applications containing both queries and update statements are not affected when relations are made temporal. New applications that use temporal relations may thus coexist with existing applications. This ability of non-temporal legacy relations and associated code to coexist with temporal relations makes it feasible for new applications to take incrementally advantage of the built-in temporal support now available in the DBMS.

2.4 Syntactically Similar, Snapshot Reducible Temporal Extensions

The reducibility requirement presented here aims at protecting the investments in programmer training and at ensuring continued efficient, cost-effective application development upon migration to a temporal model. This is achieved by exploiting the fact that programmers are likely

to be comfortable with the non-temporal query language, e.g., SQL–92.

The requirement states that the query language of the temporally extended data model must offer, for each query in the non-temporal query language, a syntactically similar temporal query that is its "natural" generalization. With this requirement satisfied and assuming that SQL–92 is the non-temporal query language, slightly modified versions of the SQL–92 queries on temporal relations are temporal queries with semantics that are easily ("naturally") understood in terms of the semantics of the the corresponding SQL–92 queries on snapshot relations. The familiarity of the similar syntax and "naturally" extended semantics is intended to make it possible for programmers to immediately and easily write a wide range of temporal queries, with little need for expensive training, few errors, and no significant initial drop in productivity.

We first define the notion of snapshot reducibility among query languages. We will use r and r^{bi} for denoting a snapshot and a bitemporal relation instance, respectively. Similarly, db and db^{bi} are sets of snapshot and bitemporal relation instances, respectively.

The definition uses a bitemporal timeslice operator $\tau^{M^{bi},M}_{(c^{tt},c^{vt})}$ (e.g., [Sch77, BM94]) which takes as arguments a bitemporal relation r^{bi} (in the data model M^{bi}) and a bitemporal instant (c^{tt}, c^{vt}) and returns a snapshot relation r (in the data model M) containing all tuples current at time c^{tt} and valid at time c^{vt}. In other words, r consists of all tuples of r^{bi} whose associated time includes the time instant (c^{tt}, c^{vt}), but without the valid and transaction time.

Definition 2.4 (snapshot reducibility) [Sno87] Let $M = (DS, QL)$ be a snapshot relational data model, and let $M^{bi} = (DS^{bi}, QL^{bi})$ be a bitemporal data model. Data model M^{bi} is *snapshot reducible with respect to* data model M if

$$\forall q \in QL\ (\exists q^{bi} \in QL^{bi}\ (\forall db^{bi} \in DS^{bi}\ (\forall c\ (\tau^{M^{bi},M}_{(c^{tt},c^{vt})}(q^{bi}(db^{bi})) = q(\tau^{M^{bi},M}_{(c^{tt},c^{vt})}(db^{bi})))))).$$ ∎

Graphically, snapshot reducibility implies that for all query expressions q in the snapshot model, there must exist a query q^{bi} in the temporal model so that for all db^{bi} and for all time arguments, the commutativity diagram shown in Figure 4 holds.

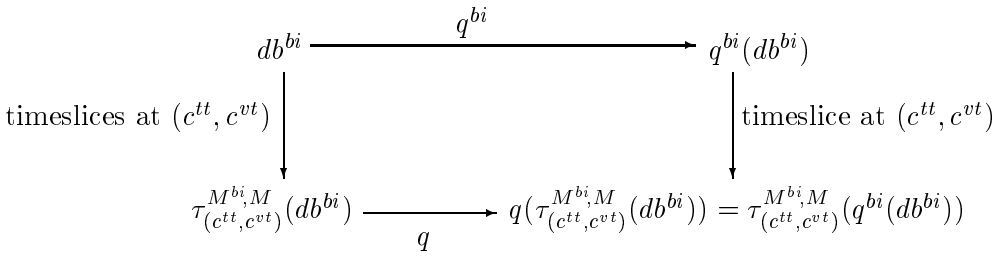


Figure 4: Snapshot Reducibility of Query q^{bi} with respect to Query q

Observe that q^{bi} being snapshot reducible with respect to q poses no syntactical restrictions on q^{bi}. It is thus possible for q^{bi} to be quite different from q, and q^{bi} might be very involved. This is undesirable, as we would like the temporal model to be a straight-forward extension of the snapshot model. Consequently, we require that q^{bi} and q be syntactically similar.

Definition 2.5 (syntactically similar snapshot-reducible extension) [BJS95] Let $M = (DS, QL)$ be a snapshot data model, and let $M^{bi} = (DS^{bi}, QL^{bi})$ be a bitemporal data model. Data model M^{bi} is a *syntactically similar snapshot-reducible extension* of model M if

1. data model M^{bi} is snapshot reducible with respect to data model M and
2. there exist two (possibly empty) strings, S_1 and S_2, such that each query q^{bi} in QL^{bi} that is snapshot reducible with respect to a query q in QL is syntactically identical to $S_1 q S_2$.

If the two strings S_1 and S_2 are both the empty string, the extension is termed a syntactically identical snapshot reducible extension. ∎

If the temporal data model treats temporal relations as new types of relations, it is possible to use the same syntactical constructs (i.e., q^{bi} and q are identical) for querying snapshot and temporal relations. In this case, the type of the argument relations determine the meaning of the construct.

Next, note that temporal upward compatibility implies that a non-temporal query on snapshot relations and the same temporal query, i.e., where the argument snapshot relations have been altered to be temporal, return the same result. Making the reasonable assumptions that temporal queries return temporal relations and that there is no separate, global context, it is thus not possible to achieve an extension that is *both* temporal upward compatible *and* syntactically identical snapshot-reducible.

The requirement specified above has associated the subset of the functionality of a temporally extended data model that extends the functionality previously introduced to include the snapshot reducible syntactically similar counterparts of the functionality of the model being extended.

Figure 5 illustrates snapshot reducibility. The temporal query q' is snapshot reducible to the snapshot query q. Query q' is applied to the temporal relation (the sequence of states across the top of the figure) and results in a temporal relation, which is represented by the sequence of states across the bottom.

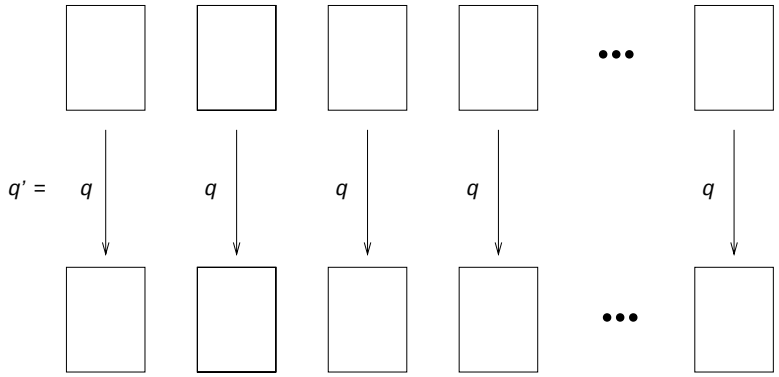


Figure 5: Evaluation of a Snapshot Reducible Temporal Query q' on a Temporal Relation

We would like the semantics of q' to be easily understood by the SQL–92 programmer. Satisfying snapshot reducibility along with the syntactical similarity requirement makes this possible. Specifically, the meaning of q' is precisely that of evaluating the syntactically similar SQL–92 query q at each state of the argument temporal relation, producing a state of the output relation for each such evaluation. As a result, temporal queries are easily formulated and understood.

Consider a temporal query q' that reduces to a snapshot query q. Figure 5 illustrates how the reducibility requirement constrains the result of query q' to be given as *some* integration into a temporal relation of the *sequence* of snapshot results obtained from evaluating query q on a *sequence* of snapshot states. Two comments are in order.

First, motivated by the sequences of states in Figure 5, we will use the term *sequenced* for queries that respect the reducibility requirement.

Second, it should be noted that the reducibility requirement merely constrains the definition of sequenced queries—it does not define them. Stated briefly, many different results of a

sequenced temporal query may satisfy reducibility, the difference being the timestamps of the result tuples. As a simple example, if a result $\{< X \parallel 1-5 >\}$ satisfies reducibility, so does the result $\{< X \parallel 1-2 >, < X \parallel 3-5 >\}$ (symbol X denotes explicit attribute values and symbol $\parallel$ separates them from the implicit timestamps).

To further constrain the language design with the purpose of providing appropriate timestamps of the result tuples, additional requirements may (and should!) be imposed. Specifically, two different requirements have been proposed, namely the point-based and the interval-based language approaches[4] [BJS95, Tom96].

In a point-based language, the difference among results such as the two above are considered insignificant. More generally, snapshot equivalent relations [JCE+94, JSS94] are considered to have the same information content in a point-based language. Thus, point-based languages often employ temporal elements as timestamps (e.g., TSQL2 [Sno95] and TempSQL [Gad88]), but may also employ points or periods.

In contrast, relations only have the same information content in an interval-based language when the relations are identical. In interval-based languages, special care therefore must be taken when deciding what timestamps should be given to result tuples.

Sequenced queries may be either point-based or interval-based. Some researchers prefer point-based languages, while other researchers prefer interval-based languages. In ATSQL, we will thus pursue an approach that is interval-based by default; and in addition, we will provide the option of *coalescing* argument and result relations. (In coalescing, value-equivalent tuples are merged when the union of their timestamps is a legal timestamp value (see, e.g., [Sno87, BSS96]); further details are given in Sections 3 and 4.) This yields a flexible language that integrates the two views, leaving the choice of a point or an interval basis to the user. We will revisit these issues when we define and study the language design.

The reducibility requirement also applies to *modifications*, as illustrated in Figure 6. A bitemporal modification destructively modifies states as illustrated by the curved arrows. As with queries, the modification is applied on a state-by-state basis. Hence, the semantics of the temporal modification is a natural extension of the SQL–92 modification statement that it generalizes.

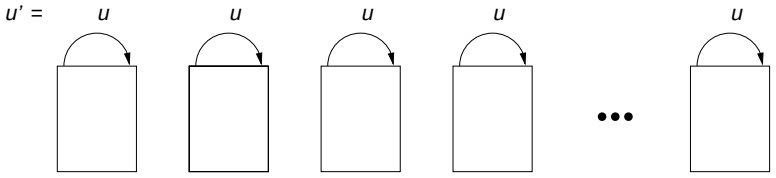


Figure 6: Evaluation of a Snapshot Reducible Temporal Modification u' on a Temporal Relation

In general and independently of snapshot reducibility, the more built-in temporal support a temporal extension provides, the better. Built-in support is typically present in the form of well-chosen defaults that perform the most typical timestamp processing (i.e., predicates and computations on timestamps) when no explicit processing is indicated in the query. The snapshot reducibility requirement is a means of requiring that built-in support be systematic and wide-ranging.

2.5 Non-Sequenced Queries and Modifications

In a snapshot reducible query, the information in a particular state of the resulting temporal relation is derived solely from information in the state at that same time of the source relation(s).

[4]These requirements are not related to migration and are not covered in detail here.

However, there are many reasonable queries that cannot be expressed as sequenced queries. So, while we do not define additional requirements, we emphasize that a temporal query language should also allow *non-sequenced* queries. Such queries are illustrated in Figure 7, in which each state of the resulting relation requires information from possibly all states of the source relation. With these queries included, we have the full functionality that may be expected from a temporal query language.

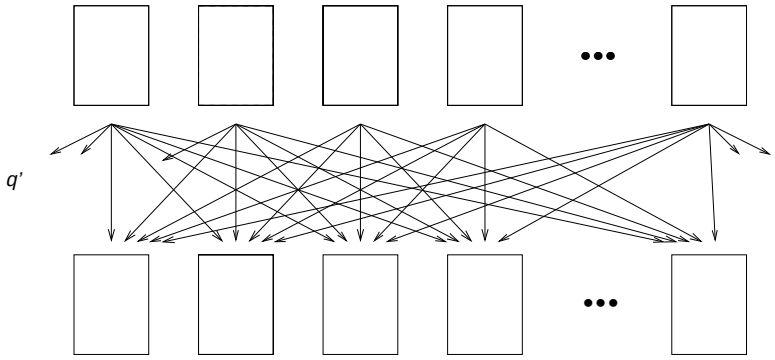


Figure 7: Evaluation of a Non-sequenced Temporal Query q' on a Temporal Relation

In Figure 7, two temporal relations are shown, one consisting of the states across the top of the figure, and the other, the result of the query, consisting of the states across the bottom of the figure. A single query q performs the possibly complex computation, with the information usage illustrated by the downward pointing arrows. Whenever the computation of a single state of the result relation may utilize information from a state at a different time, that query is non-sequenced.

The concept of non-sequenced queries naturally generalizes to modifications. *Non-sequenced modifications* destructively change states, with information retrieved from possibly all states of the original relation. In Figure 8, each state of the temporal relation is possibly modified, using information from possibly all states of the relation before the modification.

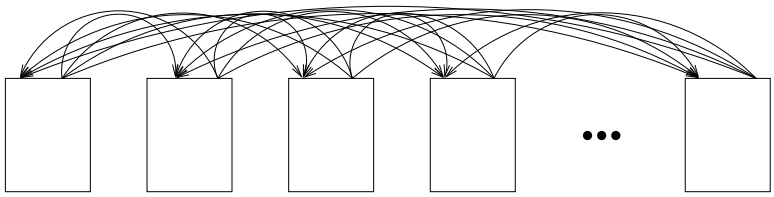


Figure 8: Evaluation of a Non-sequenced Temporal Modification u' on a Temporal Relation

Non-sequenced queries and modifications are more complex than snapshot reducible queries and modifications in the sense that the user gets less built-in support. The query language must provide a set of functions and predicates so that the user can express arbitrary temporal relationships and perform manipulations and computations on timestamps. This requires new constructs in the query language. These constructs are, however, easy to integrate into SQL–92 because they require changes at the level of built-in predicates and functions only.

2.6 Summary

In this section, we have formulated requirements that we believe are essential for the data model of a temporal DBMS to satisfy in order to ensure a smooth transition from a non-temporal

DBMS to the temporal system. We review each in turn.

Upward compatibility guarantees that replacing the existing DBMS with a new, temporal DBMS does not affect the functioning of any application code. Temporal upward compatibility guarantees that, with the new DBMS in place, it is possible to incrementally exploit more and more of the built-in temporal support. Specifically, changing existing snapshot relations to become temporal relations does not affect the functioning of any legacy code. Together, these two requirements aim at making it possible to benefit from temporal support while protecting the investments in legacy application code.

The requirement that the temporal data model be a syntactically similar snapshot-reducible extension of the existing data model guarantees that a core subset of the temporal data model maximally builds on the existing data model and makes the temporal query language easy to use for programmers familiar with the existing query language. This requirement thus aims at protecting the investments in programmer expertise and training.

It follows from the requirement that a temporal extension of SQL–92 should contain all SQL–92 statements on SQL–92 relations and should permit all SQL–92 statements to be expressed on temporal relations. In addition, the language should allow the formulation of similar sequenced temporal versions of all SQL–92 queries. For sequenced queries, the temporal system automatically computes the timestamps of the result queries. This is very attractive because the alternative is the users explicitly formulate in the queries the predicates necessary to correctly compute the timestamps, which is often a complicated and error-prone activity, leading to involved and hard-to-understand queries. Finally, when the semantics of sequenced queries are not adequate, the language should provide good support for expressing the intended timestamps and results. For example, a set of predicates on timestamps should be available that allow for the convenient expression of the possible ordering relations among timestamps.

3 ATSQL—A Prototypical Migration

The previous section motivated and defined requirements for transitioning from a non-temporal to a temporal data model without basing itself on restrictive assumptions about the properties of particular query languages and data models.

This section proceeds by demonstrating how it is possible to design a query language that satisfies the requirements. While the requirements apply to a wide range of languages, to be concrete we present a specific extension of SQL–92, termed ATSQL. (Indeed, similar requirements have been applied to develop a temporal extension of a Datalog-based language; and application to, e.g., object-oriented languages seems possible as well.)

In the first part of the section, we informally indicate how the requirements shape the syntax and semantics of ATSQL (formal definitions follow in Section 4). We then present a guided tour that illustrates the core concepts of ATSQL. A world-wide-web site has been set up that allows the reader to run the tour on-line.

3.1 From Requirements to Language Design

To make the correspondence between requirements and the language design clear, we initially consider the effect on the new language of each of the requirements in turn. We then flesh out the resulting skeleton design, adding details (e.g., the choice of specific key words) not dictated by requirements.

3.1.1 Global Impact of Requirements

Upward compatibility dictates that ATSQL must contain all statements of SQL–92. This means that SQL–92 must be an integral part of ATSQL. For example, SQL–92 contains the data type `INTERVAL` of duration values. Thus, ATSQL should also use `INTERVAL` for durations, and another key word must be chosen for the interval data type of ATSQL—we choose `PERIOD`. As another implication, the temporal extension must contend with *all* the facilities of SQL–92, e.g., nested queries, aggregates, null values, and duplicates.

For ATSQL to satisfy temporal upward compatibility, it is necessary that all SQL–92 statements be extended to work also on temporal relations as well as on snapshot relations, as described in detail in the previous section. This is achieved by letting SQL–92 modification statements on temporal relations modify the current and future states of the relations. The statements thus take effect on the states current at the time of the modification, and the effects persist in the (dynamically changing) current state from that time on and until affected by other modifications. Queries, views, and constraints then simply consider only the snapshot states of the argument temporal relations that are current and valid at the times they are evaluated. This semantics guarantees that adding time to existing snapshot relations has no effect on the applications that use them.

The requirement that there should exist syntactically similar, snapshot reducible temporal counterparts of all SQL–92 queries in ATSQL also affects the design. For each SQL–92 query, we must be able to pre- or append a fixed text string, a *flag*, to get the corresponding temporal query. While many different text strings may be (and were!) considered, we chose to prepend `SEQUENCED VALID` for valid-time relations, `SEQUENCED TRANSACTION` for transaction-time relations, and combinations of the two for bitemporal relations (details will follow).

These "sequenced" queries offer built-in, or default, timestamp-related processing—the temporal DBMS rather than the application does the potentially very complex processing involving timestamps. Thus, it is an attractive property of the new query language that as many queries as possible can be formulated as sequenced queries. This reduces the complexity of application code, with many associated benefits. To increase the utility of sequenced queries in ATSQL, we add domain specifications to these, making it possible to restrict the parts of the argument tuples considered in queries to certain time periods. We also add range specifications that allow the specification of the timestamps of tuples. These specifications are integral parts of the flags of the queries.

While the built-in semantics of sequenced queries are "natural" in the specific technical sense defined earlier, there are many queries that cannot be formulated using these default semantics. Rather, it must be possible to formulate a much wider range of queries where the application programmer is in complete control of, and responsible for, the timestamp manipulation. Such queries need another flag, different from that of sequenced queries. Using no flags is not an option, due to interaction with the semantics of SQL–92 queries on temporal relations, as dictated by temporal upward compatibility. We choose the flags `NONSEQUENCED VALID` and `NONSEQUENCED TRANSACTION`. In these nonsequenced queries, no default timestamp-related processing is built into the query language. Rather, the timestamps of temporal relations are made available in the query, essentially as regular, explicit attributes. The new period data type, `PERIOD`, is used for the timestamps. In addition, built-in facilities for constructing periods and for end-point extraction are provided along with a host of predicates on the data type. This arrangement is chosen because it gives maximum flexibility to formulate queries on temporal relations.

3.1.2 Adding Detail to the Design

The requirements shape the overall design of ATSQL as discussed above. When we move to a more detailed level in the design, good design practice (e.g., generality and orthogonality) rather

than the requirements guides the design. Below, we add detail to the general design (Section 4 provides precise semantics).

We have seen that different semantics are given to different ATSQL statements: (i) SQL–92 statements, (ii) statements with semantics dictated by temporal upward compatibility, (iii) snapshot reducible, temporal statements ("sequenced semantics"), and (iv) statements with nonsequenced semantics. Now, we study in more detail the syntax of the flags that control these semantics.

Section 2 simply requires that a flag is placed at the beginning or end of a statement and that it applies to the statement as a whole. Within these restrictions, there are several possibilities for the positioning of the flags for the different types of statements in ATSQL. We provide an EBNF syntax for each of our specific extensions to SQL–92. We thus focus on the temporal extensions and gloss over the details of SQL–92. In addition, we will focus on queries, which are the most complex statements, but will also consider other statements. In the EBNF productions that follow, terminals are of the form `"xxx"`, i.e., enclosed in quotation marks. Non-terminals of the form `<xxx>` derive from the SQL–92 standard [MS93, p.481ff], and new non-terminals are of the form `xxx` (without quotation marks). Omitting these new non-terminals yields the original (slightly simplified) SQL–92 productions.

- In *queries* and cursor expressions (termed `<cursor specification>` in SQL–92) the flags are placed at the outermost level, right at the beginning.

```
<cursor specification> ::=
   flags <query expression> [ <order by clause> ] |
   "(" flags <query expression> ")" coal [ <order by clause> ]
```

  The scope of the semantics implied by the flags is all parts of the query (e.g., including nested queries), with the exception of derived table expressions in the from clause. The non-terminal `coal` is used for specifying coalescing, to be discussed later in this section.

- In *views*, the flags are placed immediately following the `AS` keyword.

```
<view definition> ::=
   "CREATE" "VIEW" <table_name> [ "(" <view column list> ")" ] "AS"
   ( flags <query expression> |
     "(" flags <query expression> ")" coal )
```

- Flags can be associated with *derived table expressions* in from clauses. The motivation is that derived tables may be meaningfully computed independently of the computation of the remainder of the containing query. Put differently, derived table expressions have their own scope and may be replaced by views or auxiliary tables. This presents an opportunity to allow derived tables expressions to have their own individual flags. This adds flexibility to the language and improves its usefulness. As a syntactic shorthand, coalescing is also allowed after table names in the from clause (in order to facilitate point-based queries).

```
<table reference> ::=
   <table name> coal [ [ "AS" ] <correlation name> ] |
   "(" flags <query expression> ")" coal [ "AS" ] <correlation name>
```

Note that, while syntactically similar, derived tables in the from clause are quite different from subqueries in the where clause. Subqueries can be correlated with the main query and cannot be evaluated independently.

- In an *assertion*, the flags are placed right after the `CHECK` keyword.

```
<assertion definition> ::=
   "CREATE" "ASSERTION" <constraint Name>
   CHECK flags "(" <search condition> ")"
```

- *Table and column constraints* are syntactic shorthands for assertions. The flags are placed right in front of the table and column constraints, respectively.

```
<column definition> ::=
   <column name> flags <column constraint definition>

<table constraint definition> ::=
   <constraint name definition> flags <table constraint>
```

- As with queries, the flags are placed in front of *modification statements*.

```
<SQL data change statement> ::=
   flags <insert statement> |
   flags <delete statement> |
   flags <update statement>
```

Summarizing, flags are associated with all "statements" that can be evaluated meaningfully on their own. Examples include queries, data manipulation statements, assertions, integrity constraints, and views. In general, flags are placed in front of statements to emphasize their impact upon the entire statement.

Before we exemplify the different statement classes with a guided tour, we continue with an EBNF syntax for the flags and discuss their meaning.

```
flags     ::= [ flag [ "AND" flag ] ] [ "SET" "VALID" vt_range ]
flag      ::= [ modifier ] dimension [ domain ]
modifier  ::= "SEQUENCED" | "NONSEQUENCED"
dimension ::= "TRANSACTION" | "VALID"
domain    ::= period_constant
vt_range  ::= period_expression
```

The meaning of the flags naturally divides into four orthogonal parts, namely the specification of the core semantics, the time-domain specification, the time-range specification, and specification of coalescing. We discuss each in turn.

The following three types of flags determine the *core semantics* of ATSQL statements. Each of the three types of flags apply independently to valid time and transaction time.

<*empty flag*> A missing flag for a time dimension (i.e., valid or transaction) dictates upward compatibility (UC) when neither of the underlying argument relations support that time; otherwise, evaluation according to temporal upward compatibility (TUC) is dictated. For queries, the time dimension will not be present in the result relation.

`SEQUENCED` When this keyword is present for a time dimension, evaluation consistent with sequenced semantics (SEQ), i.e., built-in timestamp-related processing, is dictated for the time dimension. The time dimension will be present in relations that result from queries.

`NONSEQUENCED` This keyword signals nonsequenced semantics (NONSEQ), i.e., timestamp processing that is controlled by the application rather than the temporal DBMS. The affected time dimension is not present in query results (with this flag, the time effectively becomes an explicit attribute that can be included in the result similarly to how other explicit attributes are included).

With two time dimensions, the three cases lead to a total of nine kinds of statements, as summarized in Table 1. For simplicity, we have omitted permutations of the valid and transaction time flag, and we abbreviate `VALID` by `VT`, `TRANSACTION` by `TT`, `SEQUENCED` by `SEQ`, and `NONSEQUENCED` by `NS`.

syntax	semantics	
	vt	tt
<SQL–92>	(T)UC	(T)UC
`SEQ VT` <SQL–92>	SEQ	(T)UC
`NS VT` <SQL–92>	NONSEQ	(T)UC
`SEQ TT` <SQL–92>	(T)UC	SEQ
`NS TT` <SQL–92>	(T)UC	NONSEQ
`SEQ VT AND SEQ TT` <SQL–92>	SEQ	SEQ
`SEQ VT AND NS TT` <SQL–92>	SEQ	NONSEQ
`NS VT AND SEQ TT` <SQL–92>	NONSEQ	SEQ
`NS VT AND NS TT` <SQL–92>	NONSEQ	NONSEQ

Table 1: The Basic Usage of Flags in ATSQL

The next step is to add time-domain and time-range specifications. The *time domain* is a period constant that may be placed right after the `VALID` and `TRANSACTION` keywords, respectively. It restricts the database to the part that is valid or current during the respective period. A domain restriction is applied prior to the evaluation of a statement, i.e., in a preprocessing step.

For valid time, it can be meaningful to specify the valid time of the result, i.e., the *time range*. The `SET VALID` clause is used for this purpose. Note that it makes no sense to provide a similar clause for transaction time. Transaction-time semantics forbids this kind of user interaction [SA85]. The time range is set in a postprocessing step, i.e., after the evaluation of a query. Examples of time-domain and time-range specifications will be given in the guided tour that follows.

Finally, coalescing merges tuples with overlapping or adjacent timestamps, and identical corresponding attribute values (termed value equivalent), into a single tuple. Coalescing is allowed at the levels where the flags are also allowed. In addition, as a syntactic shorthand, a coalescing operation is permitted directly after a relation name in the from clause. In this case a coalesced instance of the relation, rather than the uncoalesced one, is considered.

```
coal ::= { "(" dimension ")" }
dimension ::= "valid" | "transaction"
```

The semantics of coalescing depends on the type of relation it is applied to. A snapshot relation cannot be coalesced. A valid-time relation can be coalesced in valid time only, and the equivalent is true for transaction-time relations. With a single time dimension, coalescing degenerates to

a merging of value-equivalent tuples with overlapping or adjacent time periods. In this case, the meaning is straightforward (performance aspects of one-dimensional coalescing have been studied elsewhere [BSS96]).

We thus turn our attention to the coalescing of bitemporal relations where the semantics are more subtle. Here, overlapping or adjacent time regions (rectangles) of value-equivalent tuples have to be merged. In the general case, overlapping rectangles do not coalesce into a single rectangle, which means that several result tuples have to be generated. This can be done in two ways: with the resulting rectangles maximized in valid time or in transaction time. We use `(VALID)` for the former and `(TRANSACTION)` for the latter. Figure 9 exemplifies bitemporal coalescing. The first picture displays the rectangular shapes defined by the timestamps of four

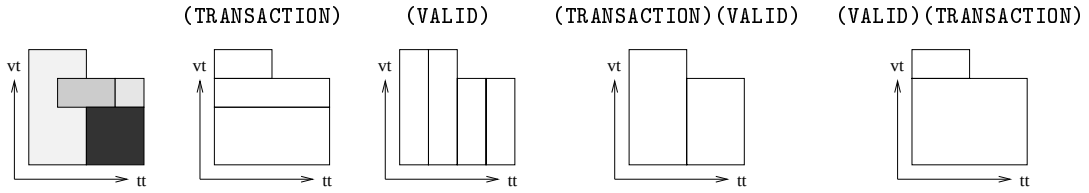


Figure 9: Different Forms of Coalescing

value-equivalent tuples. The second and third pictures illustrate the two basic coalescing operations; coalescing in transaction time and coalescing in valid time. These two basic operations can be combined to `(TRANSACTION)(VALID)`, which means that we first coalesce in transaction time and then in valid time. As exemplified by the last two pictures, the sequence of coalescing operations matters. Sequence `(TRANSACTION)(VALID)` results in maximal valid-time periods, whereas `(VALID)(TRANSACTION)` results in maximal transaction-time periods.

This concludes the syntax definition of ATSQL. The outlined semantics will be formalized in the next section. Before doing so, we present a guided tour that illustrates how ATSQL supports a seamless integration of time into SQL–92.

3.2 A Guided Tour

The tour is divided into the levels associated with the requirements discussed in Section 2. Throughout, the functionality is exemplified with input to and corresponding output from the prototype system. The reader may find it instructive to execute the sample statements on the prototype. In the examples, executable statements are displayed in `typewriter style` on a line of their own starting with the prompt "`> `".

We consider a medical application. Initially, the time support is restricted to those features supported by nontemporal databases. We gradually formulate temporal requirements to the application and show how ATSQL supports them. Doing this, we step-by-step migrate from a nontemporal to a temporal database.

The query language contains constructs needed to specify time domains and time ranges, as well as predicates on periods, i.e., language constructs to access valid and transaction time, to manipulate timestamps, and to compare periods. These constructs are defined formally in Table 2 in Section 4 where `VTIME(`r`)` and `TTIME(`r`)` return the valid and transaction-time period, respectively, of the timestamp associated with tuple variable[5] r. The remaining constructs will be explained during the tour.

[5] In SQL terminology, r is a correlation name. For simplicity, we employ commonly accepted scientific terminology throughout the paper, including in the discussion of SQL-related issues. As another example, we use "relation" in place of "table."

3.2.1 Upward Compatibility

We first define two relations. The patient relation concerns current patients. Each patient has a unique identification number, a name, an address, and a birth date. The second relation records the patients' prescriptions. With each prescription, the day it was issued and the doctor that issued it is stored. We also define a view `addicted` that computes all patients with more than one prescription.

```
> CREATE TABLE patient(Id        INTEGER PRIMARY KEY,
                       Name      VARCHAR(32),
                       Address   VARCHAR(90),
                       Birthdate DATE);

> CREATE TABLE prescription(Patient_Id INTEGER REFERENCES patient(Id),
                            Drug_Id    INTEGER,
                            Doctor     VARCHAR(32),
                            Issue_Date DATE);

> CREATE VIEW addicted (Patient_Name, Num_Of_Prescriptions) AS
    SELECT Name, COUNT(*)
    FROM patient, prescription
    WHERE patient.Id = prescription.Patient_Id
    GROUP BY Name
    HAVING COUNT(*) > 1;
```

Standard data manipulation commands are used to populate the relations, and existing retrieval facilities can be used to extract data from the relations, e.g., via the view.

3.2.2 Temporal Upward Compatibility

At some point, the hospital decides to automate the handling of the patients' medical histories. Doing this, the hospital would like to exploit the facilities of ATSQL. The migration from their relational DBMS to the temporal DBS is a substantial change in the hospital's information system, and the hospital will only implement this change provided certain guarantees are given. Specifically, the information system must continue to be operational during and after the transition—the change must not affect any legacy applications. Because the temporal DBMS satisfies upward and temporal upward compatibility, these guarantees are met (recall the discussions in Section 2). As a result, the hospital replaces its current DBMS with the temporal DBMS.

With the improved temporal support available, the patient relation is turned into a valid-time relation so that the period(s) during which a patient was, is, or will be under medical treatment may be captured in the database. The prescription relation is changed to become a bitemporal relation, thus recording the valid time of prescriptions as well as the past states of the relation. The transaction with these changes was executed on August 15, 1992.

```
> ALTER TABLE patient ADD VALID;
> ALTER TABLE prescription ADD VALID;
> ALTER TABLE prescription ADD TRANSACTION;
```

Apart from these statements, all other query language statements in the hospital's application programs remain as they were—no syntactical additions beyond SQL–92 are introduced as of yet. The legacy statements retain their usual semantics, meaning that the applications are not affected by the database having been made temporal. The following two legacy statements are executed on April 5 and 10, 1993, respectively.

```
> DELETE FROM prescription WHERE Patient_Id = 1734654 AND Drug_Id = 16584;
> INSERT INTO patient VALUES (8839782, 'Frank Zappa', 'Los Angeles', '1940/12/21');
```

The DELETE statement removes all parts of the identified prescription that are valid beyond (i.e., after) the current time (i.e., April 5). Thus, subsequent queries will no longer see the prescription, and the statement has exactly the same effect as it had on the nontemporal database in the nontemporal DBMS. The insertion adds a patient with an implicitly given valid time that extends from the time of the insertion (i.e., April 10) and until the (moving) current time. The new patient tuple will thus remain currently valid until it is affected by a deletion or an update statement.

After a series of legacy update operations (see the online tour for details), the two tables have the following tuples.

patient

Id	Name	Address	Birthdate	VTIME
5596544	Bob Marley	Miami	1945/02/06	1992/08/15−*NOW*
1846549	Jim Morison	Paris	1943/12/08	1992/08/15−*NOW*
9734859	Jerry Garcia	Forest Knolls	1942/08/01	1992/08/15−*NOW*
1734654	Janis Joplin	San Francisco	1943/01/19	1992/08/15−*NOW*
8839782	Frank Zappa	Los Angeles	1940/12/21	1993/04/10−*NOW*
9365822	Kurt Cobain	Seattle	1967/02/20	1994/01/03−*NOW*

prescription

Patient_Id	Drug_Id	Doctor	Issue_Date	TTIME	VTIME
5596544	45477	Dr Quincy	1992/07/22	1992/08/15−*NOW*	1992/08/5−*NOW*
5596544	17575	Dr Hook	1992/08/15	1992/08/15−*NOW*	1992/08/5−*NOW*
1734654	16584	Dr Jekyll	1991/12/25	1992/08/15−1993/04/04	1992/08/15−*NOW*
1734654	16584	Dr Jekyll	1991/12/25	1993/04/05−*NOW*	1992/08/15−1993/04/04

As do other schema changes, altering nontemporal relations to become temporal forces a recompilation (and optimization) of the affected statements. This recompilation ensures that the legacy statements on newly-temporal tables behave as if they were evaluated over a nontemporal database. For example, before making the schema changes, we defined a primary key, a referential integrity constraint, and a view. To implement temporal upward compatible semantics, all of these are recompiled following the schema change. This done, the query given next returns all patients that are *currently* addicted.

```
> SELECT * FROM addicted;

PATIENT_NAME  NUM_OF_PRESCRIPTIONS
---------------------------------
Bob Marley    2
```

Note the semantics of the symbol '*' in the select clause, which is used to retrieve all columns of a relation. Symbol '*' returns all explicit attributes, but not the implicit time attributes. This is essential to achieve temporal upward compatibility.

Next, the following two insertions are rejected because they violate the primary key and referential integrity constraint, respectively.

```
> INSERT INTO patient VALUES
    (5596544, 'Freddie Mercury', 'London', '1946/09/05');
> INSERT INTO prescription VALUES
    (7356378, 45477, 'Dr Quincy', SYSDATE);
> COMMIT;
```

3.2.3 Sequenced Language Constructs

So far, we have issued regular SQL–92 statements on temporal relations. This has allowed us to store (some) temporal information, but not to query it. The hospital's motivation for altering the relations to become temporal is to query the additional information now being recorded. This means that temporal queries not found in SQL–92 have to be used.

Assume that John Lennon is to get a new prescription that is to start on June 20, 1996. On June 12, 1996, when about to enter this information into the database, the medical assistant realizes that some very old prescription information for Lennon has not yet been recorded in the database.

The assistant first enters the missing, old prescription information for Lennon. This information dates back to 1984, at which time Lennon lived in Los Angeles. The prescription was given by Dr. Hook. He then enters the updated patient information and the new prescription.

```
> SET VALID PERIOD '1984/6/26 - 1984/11/30'
    INSERT INTO patient VALUES
      (7565836, 'John Lennon', 'London', '1940/10/09');
> SET VALID PERIOD '1984/6/26 - 1984/11/30'
    INSERT INTO prescription VALUES
      (7565836, 38799, 'Dr Hook', '1984/7/11');

> SET VALID PERIOD '1996/6/20 - NOW'
    INSERT INTO patient VALUES
      (7565836, John Lennon, 'New York City', '1940/10/09');
> SET VALID PERIOD '1996/6/20 - NOW'
    INSERT INTO prescription VALUES
      (7565836, 69111, 'Dr Hook', '1996/6/12');
> COMMIT;
```

With sequenced (and nonsequenced) queries, it becomes possible to query past and future information stored in temporal relations. The following sequenced query retrieves all prescription information that was current as of January 8, 1994.

```
> SEQUENCED VALID AND SEQUENCED TRANSACTION PERIOD '1994/1/8'
    SELECT * FROM prescription;

TTIME       VTIME                  PATIENT_ID  DRUG_ID  DOCTOR     ISSUE_DATE
-------------------------------------------------------------------------------
1994/01/08  1992/08/15-NOW         5596544     45477    Dr Quincy  1992/07/22
1994/01/08  1992/08/15-NOW         5596544     17575    Dr Hook    1992/08/15
1994/01/08  1992/08/15-1993/04/04  1734654     16584    Dr Jekyll  1991/12/25
1994/01/08  1994/01/05-1994/01/10  9365822     17575    Dr Hook    1994/01/03
1994/01/08  1994/01/13-1994/01/20  9365822     38799    Dr Quincy  1994/01/08
```

At this point, the hospital realizes the name Jim Morrison has been misspelled. A sequenced update is issued to make the correction apply to all times.

```
> SEQUENCED VALID
    UPDATE patient
    SET    Name = 'Jim Morrison'
    WHERE  Name = 'Jim Morison';
> COMMIT;
```

Next, the hospital is interested in knowing which patients have prescriptions from more than one doctor at the same time. This information is used to evaluate the hospital's policy, stating that all patients should be treated by their personal physician. This calls for a sequenced query:

for each point in time, it is checked whether more than one prescription, and by several doctors, is in effect. Without built-in support for temporal joins, this query is difficult to formulate. With ATSQL, it may be formulated as follows.

```
> SEQUENCED VALID
    SELECT p1.Patient_Id
    FROM prescription p1, prescription p2
    WHERE p1.Patient_Id = p2.Patient_Id
    AND   p1.Doctor <> p2.Doctor;
```

The query reveals that Bob Marley (from August 15, 1992 until NOW) and Kurt Cobain (from January 13, 1994 until January 15, 1994) had prescriptions by multiple doctors (the doctors involved were Dr. Quincy and Dr. Hook).

It is an important property of transaction-time support in relations that the relations' previously current states are retained. For some applications, this property is essential. For example, state laws or company policies might necessitate that past states be retained, to ensure accountability and traceability.

To explore this aspect, assume that Kurt Cobain died and that the forensic report (from another hospital) reveals that the death was caused by Kurt Cobain having simultaneously received Morphine and Prozac. An investigation is started. First, the relevant prescription information is retrieved with a query that is sequenced in valid time and has temporal upward compatible semantics in transaction time.

```
> SEQUENCED VALID
    SELECT proz.Doctor, VTIME(proz), morph.Doctor, VTIME(morph)
    FROM prescription proz, prescription morph
    WHERE proz.Drug_Id = 17575 /* Prozac */
    AND   proz.Patient_Id = 9365822 /* Kurt Cobain */
    AND   morph.Drug_Id = 38799 /* Morphine */
    AND   morph.Patient_Id = 9365822 /* Kurt Cobain */;

VTIME                  PROZ.DOCTOR  VTIME(PROZ)           MORPH.DOCTOR  VTIME(MORPH)
-------------------------------------------------------------------------------------
1994/01/13-1994/01/15  Dr Hook      1994/01/05-1994/01/15  Dr Quincy     1994/01/13-1994/01/20
```

The answer shows that the two drugs were indeed prescribed simultaneously between January 13 and 15. Then, who is to be blamed? —Dr. Quincy or Dr. Hook? Dr. Hook argues that it is his colleague that has to take the blame because Kurt Cobain started on Prozac on January 5, one week before Dr. Quincy started him on Morphine. Dr. Quincy maintains his innocence and cannot believe that he did not notice that Kurt Cobain was taking Prozac when he prescribed Prozac. He always checks very carefully.

Fortunately the prescription relation is bitemporal, making it possible to check exactly what Dr. Quincy would have seen when he prescribed Prozac, had he checked carefully.

```
> SEQUENCED TRANSACTION AND SEQUENCED VALID
    SELECT *
    FROM prescription
    WHERE Patient_Id = 9365822 /* Kurt Cobain */;

TTIME                  VTIME                  PATIENT_ID  DRUG_ID  DOCTOR     ISSUE_DATE
-------------------------------------------------------------------------------------------
1994/01/04-1994/01/11  1994/01/05-1994/01/10  9365822     17575    Dr Hook    1994/01/03
1994/01/08-NOW         1994/01/13-1994/01/20  9365822     38799    Dr Quincy  1994/01/08
1994/01/12-NOW         1994/01/05-1994/01/15  9365822     17575    Dr Hook    1994/01/03
```

The result reveals that Dr. Hook entered a Prozac prescription on January 4, to be valid from January 5 to January 10. Dr. Quincy entered the Morphine prescription on January 8, to be valid from January 13 to January 20. Kurt Cobain had no other prescriptions in this period, so there was no problem. The problem occurred when Dr. Hook on January 12 updated the database to record that Kurt Cobain was to take Prozac from January 5 to January 15. Dr. Hook made the update because his previous entry was in error, but it was he who forgot to check the prescription status for the longer, correct period. The correct action would have been to ensure that one of the prescriptions was terminated before it was too late. This way, Dr. Hook was found to be the culprit.

The sample queries we have seen so far illustrate the convenience of sequenced queries. The reader may find it instructive to formulate the queries in plain SQL–92. This is indeed possible, but also quite cumbersome.

The convenience of sequenced statements becomes even more apparent in statements that involve aggregates, integrity constraints, duplicates, or set difference. Without built-in sequenced support, such statements become exceedingly complicated to formulate. To illustrate this we consider two constraints. The first is contained in the definition of a new relation.

```
> CREATE TABLE drug(Drug_Id  INTEGER SEQUENCED VALID PRIMARY KEY,
                    Name     VARCHAR(32),
                    Supplier VARCHAR(32) NONSEQUENCED VALID NOT NULL) AS VALID;
```

The sequenced valid primary-key constraint ensures that at each point in time, i.e., in each snapshot, attribute `Drug_Id` is the primary key of the relation. This allows the hospital to reuse `Drug_Id` values at later points in time. If `Drug_Id` was to be a a primary key over all of time, the constraint would have been a nonsequenced one. Writing such constraints in SQL–92 is substantially more complicated.

Another constraint makes it impossible for doctors to contemporary prescribe Prozac and Morphine. The constraint is expressed in terms of an assertion.

```
> CREATE ASSERTION not_prozac_and_morphine CHECK
    SEQUENCED VALID PERIOD '1996/7/1 - NOW'
      ( NOT EXISTS ( SELECT *
                     FROM  prescription proz, prescription morph
                     WHERE proz.Drug_Id = 17575 /* Prozac */
                     AND   morph.Drug_Id = 38799 /* Morphine */
                     AND   morph.Patient_Id = proz.Patient_Id ));
```

A domain restriction is used to state that the assertion did not hold before July 1996, and that it is not to be enforced on hypothetical, future prescriptions.

Another construct that is mainly used in conjunction with sequenced queries is coalescing. Assume we want to monitor prescription times, i.e., for each of the patients, we want to know when the patient took some prescription(s).

```
> ( SEQUENCED VALID
      SELECT Patient_Id, Name
      FROM prescription, patient
      WHERE prescription.Patient_Id = patient.Patient_Id
  )(VALID)

VTIME                 PATIENT_ID  NAME
-----------------------------------------------
1992/08/15-1993/04/04  1734654     Janis Joplin
1992/08/15-1996/08/23  5596544     Bob Marley
1984/06/26-1984/11/30  7565836     John Lennon
1996/06/20-1996/08/23  7565836     John Lennon
1994/01/05-1994/01/20  9365822     Kurt Cobain
```

Without coalescing, the prescription time(s) would be returned in pieces that reflect the times of single prescriptions. While appropriate in some cases, this does not support the kind of overview we want here.

3.2.4 Non-Sequenced Statements

Nonsequenced statements are used whenever a statement involves an explicit temporal relationship such as "before," "after," "contains," "overlaps," etc. Indeed, it is a good rule of thumb that all statements involving such temporal relationships lead to nonsequenced ATSQL statements that mirror the respective relationships. Stated differently, and perhaps more directly, nonsequenced statements are employed when the built-in snapshot reducibility provided by sequenced statements is inappropriate.

Assume that we want to identify *patients who have changed doctor*, i.e., patients who first got a prescription from one doctor and later a prescription from a different doctor. This is a nonsequenced query because it cannot be answered by considering single snapshots.

```
> NONSEQUENCED VALID
    SELECT p1.Patient_Id
    FROM prescription p1, prescription p2
    WHERE p1.Patient_Id = p2.Patient_Id
    AND   p1.Doctor <> p2.Doctor
    AND   VTIME(p1) PRECEDES VTIME(p2);
```

It is an important property that sequenced and nonsequenced statements may coexist. Both types of statements manipulate or constrain facts and their associated timestamps. Sequenced statements provide built-in (default, or implicit) processing while nonsequenced statements specify the desired timestamp processing explicitly. Therefore, it is natural and also convenient to allow nonsequenced statements to contain embedded sequenced statements. In a sense, the previous statement is an example of exactly this. The timestamps of facts that we explicitly constrain in the where clause are those stored in the prescription relation. The tuple variables `p1` and `p2` thus range over sequenced "queries" in which the facts and timestamps are already available and do not have to be computed. If this was not the case, it would have been necessary to define the tuple variables in the from clause with other sequenced statements.

To further illustrate the coexistence of sequenced and nonsequenced statements, recall the sequenced statement we used to monitor prescription times (cf. Section 3.2.3). Often, monitoring is to be restricted to exceptional cases, i.e., to patients who get prescriptions over a long period. A restriction such as this is easily added by embedding a sequenced statement into a nonsequenced statement that restricts the duration.

```
> NONSEQUENCED VALID
    SELECT Patient_Id, Name, DURATION(VTIME(some_prescription), MONTH)
    FROM ( SEQUENCED VALID
             SELECT patient.Patient_Id, Name
             FROM prescription, patient
             WHERE prescription.Patient_Id = patient.Patient_Id
         )(VALID) AS some_prescription
    WHERE DURATION(VTIME(some_prescription), MONTH) > 20
```

When executed in October 1996, three patients qualify, namely Janis Joplin, Bob Marley, and Kurt Cobain.

4 A Formal Semantics

We define the semantics of ATSQL in terms of a mapping to standard and temporal relational algebra, both of which are defined here. To avoid the tedious complications related to duplicates, which have been explored in the past, we have chosen to assume a set-based framework in the semantics given here. This yields a concise coverage where the novel aspects of the general approach stand out more clearly. However, we emphasize that ATSQL follows the data model of SQL–92 and is thus not set-based.

4.1 Translating ATSQL Statements to Relational Algebra Expressions

The translation to (temporal) relational algebra expressions consists of two parts. First, we consider constructs at the level of functions and predicates. This step is straightforward and is discussed in the first section. The translation at the statement level, i.e., the translation of statements enhanced with flags, is much more involved (and important!). It is discussed in the subsequent three sections.

4.1.1 Constructs for Timestamp Manipulation

Temporal query languages generally define a variety of constructs to manipulate their various timestamp types. These include constructors (to create instances of the timestamp types), extractors (to extract constituent parts from timestamps), predicates (boolean-valued, for comparison), and operations (to create new timestamps from existing ones). Many constructs exist in the literature [Sno95, pp. 251–291]. They are relatively easy to define, and adding one more construct to a language has only a localized effect on the language design. Therefore, we only define a relatively small number of constructs here.

We will assume the timestamp representation adopted in the prototype that implements ATSQL. The prototype is built as a front end to Oracle, and four `TIMESTAMP` attributes (`VT$S`, `VT$E`, `TT$S`, and `TT$E`, denoting valid time start, valid time end, transaction time start, and transaction time end, respectively) are used to represent valid and transaction time. This representation leads to the definitions given in Table 2, where *tp* and *iv*, possibly indexed, denote a time point of type `TIMESTAMP` and a time duration of type `INTERVAL`, respectively. Also, *per* is a shorthand for `PERIOD` '$tp_1 - tp_2$' and *granule* $\in \{$`YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, `SECOND`$\}$ denotes a granularity. The constructs `VTIME` and `TTIME` that extract timestamps return errors if the tuple variables they are applied to do not support valid time and transaction time, respectively. Note also that `INTERSECT` returns an illegal period if the two argument periods do not overlap. We will use the constructs defined in Table 2 throughout, including in relational algebra expressions, e.g., in selection predicates. This makes the expressions more readable. It is straightforward to adapt these definitions to different representations, e.g., a representation that is based on the `PERIOD` data type of the evolving part SQL/Temporal of the SQL3 standard.

4.1.2 Query Expressions

Recall that we define the meaning of ATSQL query expressions by translating them to well-defined algebraic expressions. As a precursor, we introduce the notation that we will use in the algebra expressions.

We use $\langle t \rangle$, $\langle t \| VT \rangle$, $\langle t \| TT \rangle$, and $\langle t \| VT, TT \rangle$ to denote tuple variables ranging over snapshot, valid-time, transaction-time, and bitemporal relations, respectively. The vertical double-bar "$\|$" is used to separate the explicit attributes from the implicit timestamps. The valid time is referred to as VT, the transaction time as TT.

ATSQL	*Semantics*
$[\![\texttt{PERIOD '}tp_1 - tp_2\texttt{'}]\!]_{ATSQL}$	$\texttt{TIMESTAMP'}tp_1\texttt{'}, \texttt{TIMESTAMP'}tp_2\texttt{'}$
$[\![\texttt{FIRST(TIMESTAMP'}tp_1\texttt{'}, \texttt{TIMESTAMP'}tp_2\texttt{')}]\!]_{ATSQL}$	$min(tp_1, tp_2)$
$[\![\texttt{LAST(TIMESTAMP'}tp_1\texttt{'}, \texttt{TIMESTAMP'}tp_2\texttt{')}]\!]_{ATSQL}$	$max(tp_1, tp_2)$
$[\![\texttt{VTIME}(r)]\!]_{ATSQL}$	$\texttt{TIMESTAMP'}r\texttt{.VT\$S'}, \texttt{TIMESTAMP'}r\texttt{.VT\$E'}$
$[\![\texttt{TTIME}(r)]\!]_{ATSQL}$	$\texttt{TIMESTAMP'}r\texttt{.TT\$S'}, \texttt{TIMESTAMP'}r\texttt{.TT\$E'}$
$[\![\texttt{BEGIN}(per)]\!]_{ATSQL}$	$[\![\texttt{FIRST}([\![per]\!]_{ATSQL})]\!]_{ATSQL}$
$[\![\texttt{END}(per)]\!]_{ATSQL}$	$[\![\texttt{LAST}([\![per]\!]_{ATSQL})]\!]_{ATSQL}$
$[\![per_1\ \texttt{PRECEDES}\ per_2]\!]_{ATSQL}$	$[\![\texttt{END}(per_1)]\!]_{ATSQL} < [\![\texttt{BEGIN}(per_2)]\!]_{ATSQL}$
$[\![per_1\ \texttt{MEETS}\ per_2]\!]_{ATSQL}$	$[\![\texttt{END}(per_1)]\!]_{ATSQL} = [\![\texttt{BEGIN}(per_2) -^{granule} 1]\!]_{ATSQL}$
$[\![per_1\ \texttt{OVERLAPS}\ per_2]\!]_{ATSQL}$	$[\![\texttt{END}(per_1)]\!]_{ATSQL} \geq [\![\texttt{BEGIN}(per_2)]\!]_{ATSQL} \wedge$ $[\![\texttt{END}(per_2)]\!]_{ATSQL} \geq [\![\texttt{BEGIN}(per_1)]\!]_{ATSQL}$
$[\![per_1\ \texttt{CONTAINS}\ per_2]\!]_{ATSQL}$	$[\![\texttt{BEGIN}(per_2)]\!]_{ATSQL} \geq [\![\texttt{BEGIN}(per_1)]\!]_{ATSQL} \wedge$ $[\![\texttt{END}(per_2)]\!]_{ATSQL} \leq [\![\texttt{END}(per_1)]\!]_{ATSQL}$
$[\![per + \texttt{INTERVAL'}iv\texttt{'}]\!]_{ATSQL}$	$[\![\texttt{BEGIN}(per)]\!]_{ATSQL} + iv,$ $[\![\texttt{END}(per)]\!]_{ATSQL} + iv$
$[\![\texttt{INTERSECT}(per_1, per_2)]\!]_{ATSQL}$	$max([\![\texttt{BEGIN}(per_1)]\!]_{ATSQL}, [\![\texttt{BEGIN}(per_2)]\!]_{ATSQL}),$ $min([\![\texttt{END}(per_1)]\!]_{ATSQL}, [\![\texttt{END}(per_2)]\!]_{ATSQL})$
$[\![\texttt{DURATION}(per, granule)]\!]_{ATSQL}$	$([\![\texttt{END}(per)]\!]_{ATSQL} -^{granule} [\![\texttt{BEGIN}(per)]\!]_{ATSQL})$

Table 2: Definition of Simple ATSQL Constructs

In the definitions, we need auxiliary operators that timeslice relations and turn timestamps into regular, explicit attributes. These operators are overloaded to apply to valid-time, transaction-time, and bitemporal relations, and they have variants for both valid and transaction time. Their formal definition is provided in Appendix A. There are two timeslice operations. The first, τ_{tp}, selects all tuples in the argument relation with a timestamp that overlaps time point tp. The time dimension used in this selection is not present in the result relation. The second timeslice operation, δ_{per}, returns all argument tuples that overlap with period per. The timestamp of a result tuple is the intersection of per with the tuple's original timestamp. The snapshot operation SN turns a time dimension into an explicit attribute. Note that SN is not needed at the implementation level, where all attributes are explicit (cf. Section 4.1.1).

With these conventions in place, Table 3 gives the semantics for core ATSQL statements (cf. Table 1). In the table, $[\![$<SQL–92>$]\!]_{SQL-92}$ evaluates to the standard relational algebra expression that corresponds to <SQL–92> [CG85, GT91]. Next, $[\![$<SQL–92>$]\!]_T$, where $T \in \{vt, tt, bi\}$, evaluates to the same algebraic expression as does $[\![$<SQL–92>$]\!]_{SQL-92}$, except that every nontemporal relational algebra operator (e.g., $\times, \sigma, \pi$) is replaced by the corresponding temporal relational algebra operator (e.g., $\times^T, \sigma^T, \pi^T$). The algebras are defined in Section 4.2. The following two examples illustrate the definition.

Example 4.1 The ATSQL query, Q_1, below is an example of a non-sequenced query. The argument relations are assumed to be bitemporal.

```
NONSEQUENCED VALID
  SELECT p.X
  FROM p, q
  WHERE p.X = q.X
```

$$[\![\text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{tt}_{now}(\tau^{vt}_{now}(r_1)), \ldots, \tau^{tt}_{now}(\tau^{vt}_{now}(r_n)))$$

$$[\![\texttt{SEQ VT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{vt}(\tau^{tt}_{now}(r_1), \ldots, \tau^{tt}_{now}(r_n))$$

$$[\![\texttt{NS VT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{tt}_{now}(SN^{vt}(r_1)), \ldots, \tau^{tt}_{now}(SN^{vt}(r_n)))$$

$$[\![\texttt{SEQ TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{tt}(\tau^{vt}_{now}(r_1), \ldots, \tau^{vt}_{now}(r_n))$$

$$[\![\texttt{NS TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{vt}_{now}(SN^{tt}(r_1)), \ldots, \tau^{vt}_{now}(SN^{tt}(r_n)))$$

$$[\![\texttt{SEQ VT AND SEQ TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{bi}(r_1, \ldots, r_n)$$

$$[\![\texttt{SEQ VT AND NS TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{vt}(SN^{tt}(r_1), \ldots, SN^{tt}(r_n))$$

$$[\![\texttt{NS VT AND SEQ TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{tt}(SN^{vt}(r_1), \ldots, SN^{vt}(r_n))$$

$$[\![\texttt{NS VT AND NS TT } \text{<SQL–92>}]\!]_{ATSQL}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–92>}]\!]_{SQL-92}(SN^{tt}(SN^{vt}(r_1)), \ldots, SN^{tt}(SN^{vt}(r_n)))$$

Table 3: Semantics of Core ATSQL Queries

```
AND VTIME(p) PRECEDES VTIME(q)
```

This query is defined by the relational algebra expression given next.

$$[\![Q_1]\!]_{ATSQL}(p,q) = \pi_{p.X}(\sigma_{p.X=q.X}(\sigma_{VTIME(p)\ PRECEDES\ VTIME(q)}(SN^{vt}(\tau^{tt}_{now}(p)) \times SN^{vt}(\tau^{tt}_{now}(q)))))$$

Note that the mapping from SQL–92 queries to relational algebra is still the same. The temporal selection condition can be viewed as a syntactic shorthand for a standard selection condition (cf. Table 2). The only addition is the "adjustment" of the relations (SN^{vt} and τ^{tt}_{now}) to fit the non-sequenced evaluation mode in valid time dimension and the temporal upward compatible evaluation mode in transaction time dimension. ∎

Example 4.2 The following ATSQL query, termed Q_2, is sequenced in both valid and transaction time.

```
SEQUENCED VALID AND SEQUENCED TRANSACTION
  SELECT p.X
  FROM p, q
  WHERE p.X = q.X
```

It is defined by the following temporal relational algebra expression.

$$[\![Q_2]\!]_{ATSQL}(p,q) = \pi^{bi}_{p.X}(\sigma^{bi}_{p.X=q.X}(p \times^{bi} q))$$

Apart from the superscripts, which are added to the relational algebra operators, the translation between SQL–92 queries and relational algebra expressions has not changed at all. ∎

4.1.3 Domain and Range Specifications

Next, we define the semantics of domain and range specifications. A time-domain restriction restricts the argument relations in a query to contain only tuples that are valid during a specific period. Thus, only the parts of argument tuples that intersect with the time-domain restriction are considered when the query is evaluated. This is formalized in Table 4.

$$[\![\text{<modifier> VALID <domain> <ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq [\![\text{<modifier> VALID <ATSQL>}]\!]_{ATSQL}(\delta^{vt}_{\text{<domain>}}(r_1),\ldots,\delta^{vt}_{\text{<domain>}}(r_n))$$

$$[\![\text{<modifier> TRANSACTION <domain> <ATSQL>}]\!]_{ATSQL}(r_1,\ldots,r_n) \triangleq [\![\text{<modifier> TRANSACTION <ATSQL>}]\!]_{ATSQL}(\delta^{tt}_{\text{<domain>}}(r_1),\ldots,\delta^{tt}_{\text{<domain>}}(r_n))$$

Table 4: Definition of ATSQL Domain Restrictions

Next, we can also specify a time range, using the flag "SET VALID <range>" where <range> is period valued, that determines the valid times of the result tuples. There are two different situations. First, if the core statement is a SEQUENCED VALID statement then the automatically computed valid time is replaced by the value resulting from evaluating the time-range specification. Second, for all other core statements, prepending SET VALID <range> results in the inclusion of valid time into the result. Because these core statements return results that do not contain valid-time timestamps, the type of the result is changed. The valid time of a tuple is that resulting from evaluating <range>. The details are given in Table 5.

$[\![$`SET VALID` <range><ATSQL>$]\!]_{ATSQL}(r_1, \ldots, r_n) \triangleq$

$$\begin{cases} \{\langle t \| VT \rangle \mid \langle t \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge VT = \text{<range>}(t)\} \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a snapshot relation} \\ \{\langle t \| VT \rangle \mid \langle t \| VT' \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge VT = \text{<range>}(t)\} \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a valid-time relation} \\ \{\langle t \| VT, TT \rangle \mid \langle t \| TT \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge VT = \text{<range>}(t)\} \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a transaction-time relation} \\ \{\langle t \| VT, TT \rangle \mid \langle t \| VT', TT \rangle \in [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \wedge VT = \text{<range>}(t)\} \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a bitemporal relation} \end{cases}$$

Table 5: Definition of ATSQL Range Specifications

4.1.4 Coalescing

Any ATSQL query that returns a temporal relation may be coalesced. To define coalescing, let <ATSQL> denote any ATSQL query. If this query returns a valid-time relation, it may be modified to `(`<ATSQL>`)(VALID)`, to return the coalesced version of the valid-time relation. The obvious corresponding result holds when replacing valid time by transaction time. If the query returns a bitemporal relation, it may be coalesced in valid time, in transaction time, or in a combination of the two. Table 6 provides the definitions. Definitions of representative versions of operator *coal* will be given shortly.

$[\![$<ATSQL>`(VALID)`$]\!]_{ATSQL}(r_1, \ldots, r_n) \triangleq$

$$\begin{cases} coal^{vt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a valid-time relation} \\ coal^{bi}_{vt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a bitemporal relation} \end{cases}$$

$[\![$<ATSQL>`(TRANSACTION)`$]\!]_{ATSQL}(r_1, \ldots, r_n) \triangleq$

$$\begin{cases} coal^{tt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a transaction-time relation} \\ coal^{bi}_{tt}([\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![\text{<ATSQL>}]\!]_{ATSQL}(r_1, \ldots, r_n) \text{ evaluates to a bitemporal relation} \end{cases}$$

Table 6: Definition of ATSQL Coalescing

4.2 The ATSQL Temporal Relational Algebra

Having provided mappings from ATSQL to a combination of conventional and temporal relational algebra expressions, the next step is to define the algebra operators that may occur in these expressions. This completes the definition of the semantics of ATSQL queries.

We start by reviewing Codd's relational algebra. In the definitions given in Figure 10, c is a predicate and f is a generalized projection function that roughly corresponds to the select list

of an SQL–92 statement.

$$
\begin{array}{lcl}
\sigma_c(r) & \triangleq & \{t \mid t \in r \wedge c(t)\} \\
\pi_f(r) & \triangleq & \{t_1 \mid \exists t_2(t_2 \in r \wedge t_1 = f(t_2))\} \\
r_1 \cup r_2 & \triangleq & \{t \mid t \in r_1 \vee t \in r_2\} \\
r_1 \times r_2 & \triangleq & \{t_1 \circ t_2 \mid t_1 \in r_1 \wedge t_2 \in r_2\} \\
r_1 \setminus r_2 & \triangleq & \{t \mid t \in r_1 \wedge t \notin r_2\}
\end{array}
$$

Figure 10: The Snapshot Relational Algebra

We proceed by defining the temporal relational algebra operators. With the exception of the Cartesian product, the operators respect snapshot reducibility (Section 5 studies this in detail). In addition, two other properties of the algebra are noteworthy. First, the algebra is interval-based, as opposed to point-based, in that it preserves the timestamps entered into the relations. It thus generally matters for query results whether, e.g., one tuple with valid time 10–20 or two (value-equivalent) tuples with valid times 10–15 and 16–20, appear in an argument relation. Second, care was taken to only consider end points of valid and transaction timestamps when implementing the operators—intermediate time points are never used. This allows for an efficient (essentially, granularity independent) implementation.

Figure 11 contains the definition of the valid-time version of the temporal relational algebra. The transaction-time version is omitted because it is similar, the only difference being that the temporal operations are performed on the transaction-time attribute rather than on the valid-time attribute. The definition uses function *intersect* (on two periods) and the predicate *overlaps* (on two periods), both of which were defined in Table 2. The symbol "$\circ$" denotes tuple concatenation.

$$
\begin{array}{lcl}
\sigma_c^{vt}(r) & \triangleq & \{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r \wedge c(\langle t, VT\rangle)\} \\
\pi_f^{vt}(r) & \triangleq & \{\langle t_1 \| VT\rangle \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge t_1 = f(\langle t_2, VT\rangle))\} \\
r_1 \cup^{vt} r_2 & \triangleq & \{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r_1 \vee \langle t \| VT\rangle \in r_2\} \\
r_1 \times^{vt} r_2 & \triangleq & \{\langle \langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle \| VT\rangle \mid \langle t_1 \| VT_1\rangle \in r_1 \wedge \langle t_2 \| VT_2\rangle \in r_2 \wedge \\
 & & \quad VT = \mathit{intersect}(VT_1, VT_2) \wedge \\
 & & \quad VT_1 \mathit{\ overlaps\ } VT_2\} \\
r_1 \setminus^{vt} r_2 & \triangleq & \{\langle t \| VT\rangle \mid \exists VT_1(\langle t \| VT_1\rangle \in r_1 \wedge \\
 & & \quad (\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\
 & & \quad (\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\
 & & \quad VT^- < VT^+ \wedge \\
 & & \quad \neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \ \wedge VT_4 \mathit{\ overlaps\ } VT))\}
\end{array}
$$

Figure 11: The Valid–Time Algebra

Clearly the most complex operation is temporal difference. In the general case, three tuples are required to determine one result tuple, namely one tuple from r_1 and two tuples from r_2. This is illustrated in Figure 12. The second line identifies all potential starting points for periods of result tuples. Result periods may start where a period from an r_1 tuple starts and where

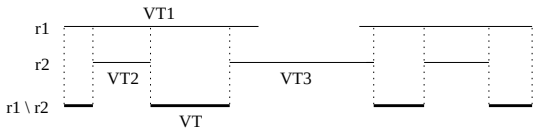


Figure 12: Valid-Time Difference

a period of a r_2 tuple ends. The second line then identifies all potential end points of periods of result tuples. The last two lines of the definition then exclude "false" result tuples: The third line eliminates meaningless combinations of starting and ending points, and the last line eliminates tuples with excessive periods.

The only operation without a non-temporal counterpart is coalescing. It is also special because it destroys the representation of timestamps in order to enforce a particular representation (maximum periods). Furthermore, coalescing removes duplicates and, therefore, violates snapshot reducibility. By definition, coalescing must merge (chains of) overlapping or adjacent value-equivalent tuples as illustrated in Figure 13. While it is not possible to compute

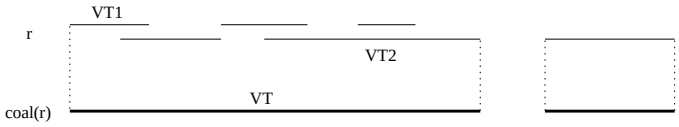


Figure 13: Coalescing a Valid-Time Relation

arbitrary transitive closures in SQL–92, coalescing is possible in SQL–92 because time is *linear* [Cel95, BSS96].

$$
\begin{aligned}
coal^{vt}(r) \quad \triangleq \quad & \{\langle t \| VT \rangle \mid \exists VT_1 \; \exists VT_2 (\langle t \| VT_1 \rangle \in r \wedge \langle t \| VT_2 \rangle \in r \wedge \\
& VT_1^- < VT_2^+ \wedge VT^- = VT_1^- \wedge VT^+ = VT_2^+ \wedge \\
& \forall VT_3 (\langle t \| VT_3 \rangle \in r \; \wedge VT^- < VT_3^- < VT^+ \Rightarrow \\
& \qquad \exists VT_4 (\langle t \| VT_4 \rangle \in r \wedge VT_4^- < VT_3^- \leq VT_4^+)) \wedge \\
& \neg \exists VT_5 (\langle t \| VT_5 \rangle \in r \wedge (VT_5^- < VT^- \leq VT_5^+ \vee VT_5^- \leq VT^+ < VT_5^+)))\}
\end{aligned}
$$

The two tuples introduced in the first line serve to define the starting (VT_1^-) and end (VT_2^+) points of a coalesced tuple, as specified in the second line. The third and fourth lines ensure that there are no gaps between VT^- and VT^+. This is done by ensuring that every tuple with a start time between VT^- and VT^+ is extended towards VT^-, i.e., there must exist another tuple with a valid time containing the respective start time. Finally, on the last line we make sure that the valid time of the result tuple is maximal, i.e., there may not exist another tuple that contains either VT^- or VT^+.

The bitemporal relational algebra is a natural extensions of the valid-time (and transaction-time) algebra. However, both time dimensions must be handled simultaneously, meaning that rectangles rather than periods must be handled. While this does not change the basic ideas, it adds to the complexity of the definitions; for this reason, it is deferred to Appendix B.

4.3 Summary of Semantics

The syntax of ATSQL queries was specified in Section 3. This section then gave the semantics of ATSQL queries, in three steps. First, the semantics of constructs for timestamp manipulation was given. The second step was to define the semantics of core ATSQL queries, as well as

queries with domain and range specifications and coalescing. Specifically, mappings to relational and temporal relational algebraic expressions were given. Finally, the relational and temporal relational algebras were defined.

Taking into consideration that ATSQL is a language more complicated than SQL–92, the language it extends, the semantics are quite concise. This has been achieved by giving the semantics of ATSQL in terms of the semantics of SQL–92 (specifically, in terms of a mapping of SQL–92 queries to relational algebra). This, in turn, is possible because ATSQL by design systematically and faithfully extends SQL–92.

5 Properties of ATSQL

This section provides discussions of the properties of ATSQL. Specifically, we argue that ATSQL indeed, by design, satisfies the compatibility and reducibility properties that were introduced in Section 2. Next, we characterize the scope of the impact on the language design of the snapshot reducibility requirement, thereby showing that this requirement only constrains part of the language definition and that other important design decisions went into the "sequenced" design of the language. Finally, we contrast ATSQL's approach to built-in semantics with the default-based approach. For brevity and to avoid tedious details, we cover the valid-time dimension only.

5.1 Compatibilities and Syntactic Restrictions

By design, ATSQL is upward compatible with respect to SQL–92. The whole approach adopted for defining the syntax and semantics of the language emphasizes this property. The syntax of ATSQL was given by *extending* the syntax of SQL–92 with non-mandatory constructs. Thus, ATSQL contains all legal SQL–92 statements. The approach taken to define ATSQL also makes it straightforward to verify that in ATSQL, all SQL–92 statements retain their SQL–92 semantics. Specifically, the first definition in Table 3 covers SQL–92 statements.

Observe that in defining the syntax of ATSQL, we introduced a few reserved words (e.g., `SEQUENCED`) not part of SQL–92. As already discussed in Section 2, this strictly speaking leads to a violation of upward compatibility. This also is a reason why SQL–92 is not upward compatible with SQL–89. It would have been possible to reuse instead existing reserved words, yielding a more strict upward compatibility, but that would have obscured the semantics of the extensions.

ATSQL is also temporal upward compatible with SQL–92. This follows from the upward compatibility, the first definition in Table 3, and the definition of the ATSQL meaning of SQL–92 modification statements when applied to temporal relations. These statements are covered in detail elsewhere [BBJS97].

Finally, snapshot reducible ATSQL statements (discussed below) are syntactically similar with respect to SQL–92. This again follows from the definition of the language. Definition 2.5 constrains the differences between an SQL query and the corresponding snapshot reducible ATSQL query to be at most two fixed strings (i.e., the "flags" in ATSQL), prepended and appended to the SQL query, respectively. The fixed strings do not depend on the particular query, but are the same for all queries. Satisfying this requirements leads to a wholesale "semantics" approach to defaults, as will be discussed in Section 5.4.

5.2 Snapshot Reducibility

In this section the focus of attention is the snapshot reducibility property of ATSQL with respect to SQL–92. The satisfaction of this property follows from the design of the mapping and the definition of the temporal algebraic operators. Below, we discuss first how the definition of the

temporal algebra is shaped to make the preservation of the property possible. Then follows a discussion of the top-level mapping of ATSQL statements (as given in Table 3).

Recall the definition of snapshot reducibility (Definition 2.4). We first show that the valid-time relational algebra (Figure 11) almost (!) has this property with respect to the snapshot relational algebra (Figure 10).

Theorem 5.1 The valid-time relational algebra (Figure 11) satisfies the reducibility properties below with respect to the snapshot relational algebra (Figure 10). Recall that *tp*, c, and f denote a time point, a predicate, and a projection list, respectively.

- $\forall tp\ (\tau^{vt}_{tp}(\sigma^{vt}_{c}(r)) \equiv \sigma_c(\tau^{vt}_{tp}(r)))$
- $\forall tp\ (\tau^{vt}_{tp}(\pi^{vt}_{f}(r)) \equiv \pi_f(\tau^{vt}_{tp}(r)))$
- $\forall tp\ (\tau^{vt}_{tp}(r_1 \cup^{vt} r_2) \equiv \tau^{vt}_{tp}(r_1) \cup \tau^{vt}_{tp}(r_2))$
- $\forall tp\ (\pi^{-}_{r_1.VT,r_2.VT}(\tau^{vt}_{tp}(r_1 \times^{vt} r_2)) \equiv \tau^{vt}_{tp}(r_1) \times \tau^{vt}_{tp}(r_2))$
- $\forall tp\ (\tau^{vt}_{tp}(r_1 \setminus^{vt} r_2) \equiv \tau^{vt}_{tp}(r_1) \setminus \tau^{vt}_{tp}(r_2))$

The equivalences hold for arbitrary relations, with the only restrictions being that in the first two equivalences c and f refer to explicit attributes only and that the relations be union compatible in the third and fifth equivalence. Also, $\pi^{-}_{X}(r)$ is given by $\pi_{r.*\setminus X}(r)$ where $r.*$ denotes all the attributes of r. ∎

The proofs of these properties may be found in Appendix 5.1.

It follows that the valid-time selection, projection, union, and difference operators are snapshot reducible to their snapshot counterparts. Thus, all valid-time algebra statements involving only these operators are snapshot reducible to the snapshot algebra statements obtained by simply removing the vt superscripts.

However, the equivalence involving the Cartesian products attracts attention: this operator is not reducible to the snapshot Cartesian product! While it is straightforward to define a temporal Cartesian product that is snapshot reducible to the snapshot Cartesian product, we have chosen a definition that violates snapshot reducibility. Let us explore why this is a good design decision.

Initially, note that the "problem" with our temporal Cartesian product is that it retains the implicit valid-time attributes of its argument relations and turns them into explicit attributes. The operator π^{-} is introduced to eliminate these "extraneous" attributes. Now, when mapping a (sequenced) temporal query to its algebraic equivalent, we would like to exploit the standard mapping used when mapping SQL queries to relational algebra. Consider the following query.

```
SEQUENCED VALID
  SELECT <L>
  FROM p, q, r
  WHERE <P>
```

We would like to map this query to

$$\pi^{vt}_{<L'>}(\sigma^{vt}_{<P'>}((p \times^{vt} q) \times^{vt} r))$$

where $\langle L' \rangle$ and $\langle P' \rangle$ are slight syntactical variations of `<L>` and `<P>`, respectively. One possible choice for predicate `<P>` would be

```
DURATION(VTIME(p),DAY) + DURATION(VTIME(q),DAY) < DURATION(VTIME(r),DAY)
```

With our definition of the valid-time Cartesian product, we can express the corresponding algebra predicate $\langle P' \rangle$ as follows because the timestamps of the argument tuples are retained as explicit attributes.

$$DURATION(p.VT, DAY) + DURATION(q.VT, DAY) < DURATION(r.VT, DAY)$$

Using a snapshot-reducible Cartesian product would make it impossible to construct a corresponding predicate $\langle P' \rangle$. The information required to evaluate the predicate would be lost. This observation holds for any tuple timestamped and any homogeneous [Gad88] attribute-value timestamped data model. Snapshot-reducible temporal Cartesian products for such models are unable to serve the role during the mapping of temporal SQL queries to algebraic expressions that the snapshot Cartesian product serves when mapping SQL queries to relational algebra.

One approach to retain the simple mapping and also retain a snapshot reducible temporal Cartesian product is to introduce an additional (information-preserving) Cartesian product that produces results with *two implicit* valid times. But this latter product returns results that are not valid-time relations and thus breaks the closedness property of the algebra, an undesirable complication. This approach was adopted in the algebra for the HSQL data model [Sar93] that includes both a reducible "Concurrent Product" and an information-preserving "Cartesian product."

Another approach that will ensure that the necessary information is available in the algebra for evaluating any predicate `P` is to introduce an n-ary valid-time join that can then be defined to be snapshot reducible. The transformation to algebra would then be as follows.

$$\pi^{vt}_{\langle L' \rangle}(\bowtie^{vt}_{\langle P' \rangle} (p, q, r))$$

This approach was adopted for the algebra proposed for TSQL2 [SJS95]. While the added complexity of an n-ary operator may be undesirable, there is another problem with this approach. Consider the sample `<L> = p.X, VTIME(p)` that specifies that the implicit valid-time attribute of relation `p` is to be present in the result as an explicit attribute.

With the n-ary join approach, it is not possible to produce an equivalent $\langle L' \rangle$. Specifically, the original valid times of tuples also from `p` cannot be inferred from the result of the join. With our Cartesian product, we have $\langle L' \rangle = p.X, p.VT$.

Most temporal algebras have operators that are snapshot reducible with respect to the snapshot Cartesian product (e.g., the TJOIN [NA89], the Concurrent Product Operator [Sar93], the cross-product [NG93], (temporal) equijoin [CCT93], and the valid-time Cartesian product[6] [Sno93]; reference [MS91] gives a survey).

The simple binary temporal Cartesian product defined here permits the use of the standard mapping from SQL to algebra without imposing any restrictions on the contents of the `SELECT` and `WHERE` clauses. As we discuss next, the non-reducibility of the operator does not lead to violations of the reducibility of ATSQL to SQL.

In ATSQL (and SQL) queries, Cartesian products are specified in the `FROM` clause of the `SELECT-FROM-WHERE` statement. For an ATSQL query to be reducible, the result of evaluating it must not include the implicit valid-time attributes of argument tuples as explicit attributes. In reducible queries, it is not possible to select a time dimension of a relation; and defaults, e.g., `SELECT *`, also do not expand to include the implicit time attributes. We conclude that due to the presence of subsequent projections in the definition of reducible queries, the presence of the additional explicit time attributes in the results of Cartesian products do not compromise snapshot the reducibility of ATSQL.

[6] This operator is defined in a non-homogeneous attribute-value timestamped data model. Unlike any other product we have seen, this operator reduces to the snapshot Cartesian product and yet does not possess the two deficiencies.

5.3 Sequentiality Versus Snapshot Reducibility

In Section 2, we formulated a reducibility requirement to a temporal extension of a query language. In this section, we put focus on what parts of the temporal query language design that this property shapes and to which extent the property constrains the design of these parts. We first observe that this property of a query language does not define the language, but merely constrains the definition of a subset of it. We then characterize this subset and describes to which extent the subset is constrained. Subsequently, we explore the properties of the remaining, unconstrained parts of ATSQL.

5.3.1 Pure Sequenced Queries

To characterize the parts of ATSQL affected by the reducibility property, we syntactically characterize two classes of query language statements.

Definition 5.2 An ATSQL statement is *sequenced* if it contains the reserved keyword `SEQUENCED`.

Definition 5.3 A sequenced ATSQL statement is *pure* if it does not contain any of the the reserved keywords `VTIME`, `TTIME`, and `SET VALID`.

Both classes of statements are defined syntactically, and it is apparent from the definitions that the pure sequenced queries are strictly contained in the class of sequenced queries.

The implicit times of relations are accessed explicitly in a statement using either `VTIME(...)`, `TTIME(...)`, or `SET VALID ...`. The definition of pure statements thus enumerates and prohibits all ways of explicitly referencing the implicit time. The class of pure ATSQL statements, we denote by ATSQL^{pure}.

To exemplify non-pure sequenced statements, consider the following two statements.

```
SEQUENCED VALID
  SELECT *
  FROM p, q
  WHERE p.X = q.X
  AND DURATION(VTIME(p),YEAR) > 5
```

```
SEQUENCED VALID
  SELECT *
  FROM p, q
  WHERE p.X = q.X
```

In the first statement, we we temporally join relations `p` and `q`, but are only interested in `p`-tuples with a valid time longer than 5 years. Through the term `VTIME(p)` the valid time of `p` is accessed, making the the query non-pure. The statement must be answered by considering multiple snapshots at a time because we cannot decide whether a tuple is valid for more than 5 years by considering individual snapshots in isolation. It is intuitively clear that snapshot reducibility cannot by itself be used to define the semantics of this statement. The second statement also temporally joins relations `p` and `q`, but no additional restriction on the valid time of `p`-tuples is enforced. This makes the statement pure sequenced.

We have already shown that for each SQL–92 statement, there exists a (syntactically similar) ATSQL statement that is (snapshot) reducible to it. It further holds true that these ATSQL statements are pure. The following property thus holds.

Theorem 5.4 The subset ATSQL^{pure} of ATSQL is syntactically similar snapshot reducible with respect to SQL–92. ∎

There are then at least as many ATSQL^{pure} statements as there are SQL–92 statements. It also follows that the reducibility requirement imposed on ATSQL is satisfied by and thus only constrains the definition of a small subset of ATSQL, namely ATSQL^{pure}. The requirement does by itself not constrain any other parts of ATSQL—in particular, it does not impose requirements on the non-pure, sequenced statements.

The opposite of Theorem 5.4 also holds.

Theorem 5.5 For each ATSQL^{pure} statement, there exists a unique SQL–92 statement that the ATSQL^{pure} statement is reducible to. ∎

Together, the two theorems state that there is a one-to-one correspondence between the sets of ATSQL^{pure} and SQL–92 statements. The pure query language statements are exactly those that are reducible to statements in the underlying non-temporal query language.

Although snapshot reducibility in itself does not constrain the definition of non-pure ATSQL statements, these statements are still an integrated part of the query language and thus in practice are constrained to observe the spirit of sequentiality. (They are consistent with viewing a temporal database as a sequence of nontemporal database states.)

To illustrate this, recall the first sample query above. The non-pure part of this temporal join query is the where clause condition `DURATION(VTIME(p),YEAR) > 5`. As explained above, this condition cannot be evaluated by considering individual snapshots in isolation. However, the temporal join itself can still be conceptualized as a nontemporal join evaluated on each snapshot. Snapshot reducibility can thus be used to constrain the semantics of the pure, "nontemporal constructs" (e.g., a join, difference, or subquery) of a non-pure sequenced ATSQL statement, but it cannot be used to define temporal constructs that provide explicit reference to the timestamps.

5.3.2 Reducibility and Beyond

Simply requiring that a query language respects snapshot reducibility does not amount to defining the language; rather, snapshot reducibility merely constrains the possible definitions, leaving important design decisions open. Next, we thus consider some of the design considerations that went into defining ATSQL, but that are beyond the scope of the snapshot reducibility requirement.

Snapshot reducibility is point-based in nature. In contrast, ATSQL is interval-based in that tuples are timestamped with time periods. An example illustrates that how snapshot reducibility falls short in specifying exactly how tuples should be timestamped. Consider a view `prescr_period` from the medical application in Section 3.2 that records the periods during which patients took prescriptions. It is important to keep the individual prescription periods separate, but neither the doctor nor prescription names are of interest. Four different instances of the view are shown in Figure 14.

prescr_period

Name	VTIME
Kurt Cobain	1994/1/2−1994/1/5
Kurt Cobain	1994/1/6−1994/1/9

prescr_period$_1$

Name	VTIME
Kurt Cobain	1994/1/2−1994/1/6
Kurt Cobain	1994/1/7−1994/1/9

prescr_period$_2$

Name	VTIME
Kurt Cobain	1994/1/1−1994/1/9

prescr_period$_3$

Name	VTIME
Kurt Cobain	1994/1/2−1994/1/4
Kurt Cobain	1994/1/5−1994/1/7
Kurt Cobain	1994/1/8−1994/1/9

Figure 14: Snapshot Equivalent Relations

Clearly, the relations are different; and yet, in the eyes of snapshot reducibility, the relations are the same. This is so because to snapshot reducibility, a relation is no more than a sequence of snapshot relations; and the four relations are mutually snapshot equivalent, i.e., for all time points tp, the snapshots $\tau_{tp}^{vt}(prescr\_period)$, $\tau_{tp}^{vt}(prescr\_period_1)$, $\tau_{tp}^{vt}(prescr\_period_2)$, and $\tau_{tp}^{vt}(prescr\_period_3)$ are identical. This is then an example where four different relations—with quite different meanings in terms what is important for the application—cannot be told apart by snapshot-equivalence-based properties. Rather, additional design decisions are needed to fully define timestamps of query results.

One decision would be to require result tuples to be coalesced. This would make $prescr\_period_2$ the unique query result. This solution, however, has consequences that may be unwanted. The actual valid times of tuples inserted into the underlying relation are no longer preserved. In the example, the view can no longer be used for capturing how many prescriptions were issued and for what periods, but only when a person was on prescriptions. Thus, with coalesced results, this view does not record how many times Kurt received a prescription—without coalescing, this is possible.

It may be argued that if it is important to retain the periods of individual prescriptions and if the model enforces coalescing, an attribute such as "PrescriptionNumber" can simply be added. However, this creates a potential consistency problem because the relation then stores redundant information. The PrescriptionNumber ordering of tuples for each patient must be kept consistent with the timestamp ordering of the tuples.

Next, coalescing violates snapshot reducibility because it eliminates duplicates from snapshots [BJS95], e.g., $\{\langle a \| 5-9\rangle, \langle a \| 7-10\rangle\}$ would be coalesced to $\{\langle a \| 5-10\rangle\}$. A snapshot at time 8 would on each instance would yield $\{\langle a\rangle, \langle a\rangle\}$ and $\{\langle a\rangle\}$, respectively. Thus, a simple tuple-count query would yield different results.

Because it is our contention that there seems to be advantages to preserving timestamps as originally entered into the database, as well as to coalescing, we have chosen to allow both in ATSQL. The default is to preserve the timestamps—being an irreversible operation, coalescing by default makes little sense. Thus, we have introduced special, convenient syntax for specifying coalescing. This means that ATSQL users are free to preserve timestamps or to coalesce them.

5.4 Built-in Semantics and Defaults

The additional user-friendliness achieved when extending a query language with temporal support seems to have two sources. First, the addition of *new temporal data types* with associated constructors, predicates, and operations makes temporal data management more convenient. For example, adding a period data type to SQL–92 makes it easier to manage the valid time of tuples. To illustrate this, assume that snapshot relation `p` has attributes `a`, `b`, and `VT`, with the latter being period-valued and recording valid time. Similarly, let relation `q` have attributes `b`, `c`, and `VT`. A temporal natural join of these two relations is expressed as follows.

```
SELECT p.a, p.b, q.c, INTERSECT(p.VT, q.VT) AS VT
FROM p, q
WHERE p.b = q.b AND p.VT OVERLAPS q.VT
```

Without a data type for time periods, using instead pairs of time-point valued attributes, this query gets significantly harder to formulate and understand.

Second, providing *built-in timestamp processing* adds user-friendliness. For example, in languages such as TSQL2 and ATSQL, it is possible to write a temporal natural join essentially as a regular natural join, as follows.

```
TSQL2:                      ATSQL:
   SELECT p.a, p.b, q.c       SEQUENCED VALID
```

```
FROM p, q                    SELECT p.a, p.b, q.c
WHERE p.b = q.b              FROM p, q
                             WHERE p.b = q.b
```

In these queries, we assume that `p` and `p` are valid-time relations. The two queries are essentially the same, the only difference being that TSQL2 by default performs a sequenced computation when the argument tables are temporal and that ATSQL requires a sequenced flag to do the same.

For a simple join query, the added user-friendliness of using built-in processing over simply using a new data type is clear, but not substantial. This is so because it is relatively straightforward to generalize a snapshot natural join to a temporal natural join. When considering complicated SQL–92 queries, possibly involving aggregates, the generalized queries frequently become very difficult to formulate in SQL–92, with or without new data types. This is where the power of built-in processing stands out. For example, in ATSQL *any* query is generalized by simply prepending `SEQUENCED VALID`.

Two approaches to built-in processing may be identified. The first approach is represented by TSQL2 [Sno95, e.g., pp. 291–297], which provides comprehensive built-in processing. In this approach, built-in processing is provided by *syntactically defined defaults*. For example, the TSQL2 join above is a shorthand for the following query.

```
SELECT p.a, p.b, q.c
VALID INTERSECT(p, q)
FROM p, q
WHERE p.b = q.b
```

When the valid clause of TSQL2 is missing from a query and all argument relations are valid-time relations, the meaning of the query is given by adding a valid clause that generates a timestamp of result tuples that is given by the intersection of the timestamps of the argument tuples.

The second approach is the *"semantic" approach* we have adopted for ATSQL. Rather than defining query language statements that provide built-in processing in terms of syntactical additions to them, the built-in processing is defined semantically, i.e., to be consistent with snapshot reducibility.

The syntactic approach has been shown to be problematic. We have previously identified TSQL2 statements that have no obvious semantics [BJS95]. The problem stems in part from the syntactic defaults. For example, the rule above becomes unclear when select statements are included in the where clause. It turns out that it is exceedingly difficult to define built-in processing in SQL–92 via syntactic defaults in a manner that is comprehensive and also systematic and thus sufficiently easily comprehensible for it to be practically useful.

The problem with syntactic defaults is one of scalability over language constructs, and SQL–92 has numerous constructs with subtle semantics and also lacks orthogonality. With syntactic defaults, it must be possible to state in the query language the default for a wide range of statements. Accomplishing this is quite challenging because each syntactic default is dependent on the specifics of the statement that it is the default for. In this way, the complexity of specifying the actual default is comparable to the complexity of specifying the entire language.

Semantic defaults behave quite differently. No attempt is made to actually define defaults that could be used instead of the default itself. Specifically, ATSQL does not try to syntactically map sequenced statements to semantically equivalent (non-sequenced) ATSQL statements. (Our experience with compiling ATSQL to SQL suggests that this is impractical.) This makes ATSQL conceptually simple, and it becomes much more robust with respect to extensions and dialects of SQL because the SQL part of an ATSQL statement essentially can be treated as a black box.

In fact, ATSQL may be seen as the result of replacing in TSQL2 the syntactic defaults by the systematic, semantically-based built-in processing from ChronoLog [Böh94], thereby fixing fundamental problems in TSQL2 [BJS95]. The semantic approach leads to a syntactically identifiable class of queries with built-in support and thus provides a systematic and wholesale approach to built-in default processing.

6 Related Work

We divide the discussion of related research into three aspects. Initially, we discuss the background of the language requirements from Section 2, as well as related requirements. Then we position the language with respect to earlier languages and ongoing efforts. Finally, we briefly evaluate the existing temporal SQL's with respect to the requirements.

The formulation of the two compatibility requirements in Section 2 evolved in part from studies of TSQL2 [Sno95] and were developed with Richard Snodgrass and John Bair. Many discussions with Richard Snodgrass in the context of developing proposals for the SQL/Temporal part [SBJS96a, SBJS96b] of the evolving SQL3 standard also shaped the formulation of these requirements.

The formulation of the sequentiality requirement borrowed the fundamental notion of snapshot reducibility, which we believe first appeared in the literature in 1987 [Sno87]. Using snapshot reducibility, it became possible to precisely define an informal requirement that was initially termed temporal semi-completeness [BJS95] and which was developed as a requirement of ChronoLog [Böh94], a temporal deductive database system. Again, studies of TSQL2 helped shape this requirement.

Few other precise query language requirements have been proposed in the past. For example, while the phrase "upward compatibility" has been used widely and in many contexts, we have found no precise definition of it. Similarly, other upward-compatibility-like requirements, as exemplified next, have been mentioned but never precisely defined.

"The default options are defined such that a query that omits the temporal portion retains the standard meaning of the corresponding SQL SELECT statement." [Ari86, p. 513]

"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." [NA93, p. 99]

"HSQL is a superset of the popular query language SQL." [Sar93, p. 123]

"In fact, the standard clauses of SQL have identical meanings in HSQL." [Sar93, p. 125]

"IXSQL is syntactically and semantically upwards consistent with SQL2." [LM96, p. 1]

In only one place have we encountered an approach that aims at satisfying a requirement that seems similar to temporal upward compatibility. Specifically, the TempSQL language (e.g., [GN93]) introduces a notion of so-called classical and system user types. System users see the full temporal database, while classical users see only the current snapshot of the database. If applications are classical by default, and if individual statements, rather than all statements issued by a user, can be independently made temporal, this would essentially (providing that a number of other design decisions are made correctly) yield a temporal upward compatible SQL extension (see also below).

Now, let us consider the ancestry of ATSQL. In a previous paper, we showed that TSQL2 neither satisfies temporal upward compatibility nor syntactically similar snapshot reducibility

with respect to SQL–92. Indeed, we showed that there are statements in TSQL2 that appear to have no obviously correct semantics (the semantics of TSQL2 were given only informally, in SQL-standard style and in technical commentaries). One problem with TSQL2 is that it is pure in that it does not permit "duplicates" in temporal relations, while SQL–92 does. We felt that there was a need for minimally changing TSQL2 to rectify these deficiencies and thus embarked on designing ATSQL, with the "A" signifying that this would be an "applied" language that permitted duplicates.

At first sight, ATSQL thus looks very much like TSQL2. However, readers familiar with ChronoLog and its SQL cousin, ChronoSQL, will notice that the language is perhaps conceptually closer to these. The explanation is simple: ATSQL was designed to satisfy the sequentiality requirement and the other two languages were designed to satisfy the predecessor of this requirement.

Finally, ATSQL is also related to two ANSI proposals [SBJS96a, SBJS96b] for additions to the (ISO) SQL/Temporal part of SQL3 (SQL/T, for short). Some central aspects of those proposals derive from earlier joint work with Richard Snodgrass on designing ATSQL. As a result, these ANSI-accepted language proposals are quite similar to the language proposed in this paper.

To complete the coverage of related research, Table 7 evaluates all existing temporal SQL proposals that we are aware of, and SQL–92, with respect to our requirements. Recall that the abbreviations UC, TUC, and SR used in the table denote upward and temporal upward compatibility and syntactically similar snapshot reducibility, respectively. The findings reported

Language	Reference	UC	TUC	SR	Comments
TOSQL	[Ari86]	yes	no	no	Extends only a limited subset of SQL.
TSQL	[NA87] [NA89] [NA93]	yes	no	no	Not all snapshot relations can be made temporal. Some SQL views cannot be defined on temporal relations. Automatic coalescing violates TUC.
HSQL	[Sar90] [Sar93]	yes	no	no	SQL queries on temporal relations return temporal relations.
SQL–92	[MS93]	yes	no	no	Adding explicit timestamp columns violates TUC.
TempSQL	[BG93] [GB93] [GN93]	yes	partly compliant	no	Only a subset of SQL is considered. TUC is satisfied only for classical users.
IXSQL	[Lor93] [LM96]	yes	no	no	Extension of SQL with a parameterized period ADT with accompanying query-language facilities is proposed.
ChronoSQL	[Böh94]	yes	no	yes	Designed to fulfill SR. TUC queries default to all states rather than to the current state.
TSQL2	[Sno95]	yes	no	no	Full syntax given. Semantics defined informally in SQL-standard style.
"SQL/T"	[SBJS96a] [SBJS96b]	yes	yes	yes	Designed to satisfy these requirements.

Table 7: Summary of Compatibility and Reducibility Compliances

in the table should be qualified. We report compliance with a requirement if this is claimed in the documentation of a model, or if non-compliance cannot be proved. With the exception of IXSQL and TSQL2, 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

clear. Next, aspects related to the use of regular SQL statements—updates, in particular—on temporal relations or a combination of temporal and non-temporal relations are typically not defined. This makes it hard to verify temporal upward compatibility. Finally, the definition of the syntax, and in particular of the semantics, of several of the models is quite informal and incomplete.

We briefly consider each language in turn and in chronological order of their appearance (a substantially more detailed study may be found elsewhere [BBJS97]).

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, and nor do they satisfy reducibility. SQL–92 is upward compatible with itself—this is trivially true due to the reflexivity of upward compatibility. But it is not temporal upward compatible with itself, and as it has not temporal extensions, the reducibility property is not satisfied.

The next model, TempSQL, introduces the concepts of classical and system user types that may be used to obtain (partial) satisfaction of both compatibilities. Temporal upward compatibility is only satisfied for so-called classical users that see only the current state of all relations. When a classical-user application needs access to past states of a relation and is made a system-user application, the full application must be rewritten, breaking temporal upward compatibility.

IXSQL is different from all the other models in that it does not provide support for implicit time; rather, it adds a parameterized abstract period data type and associated facilities for modification and queries to SQL. For the same reason as for SQL–92, it does not satisfy temporal upward compatibility. As time is explicit, reducibility is also not satisfied.

ChronoSQL was designed to illustrate how to carry over the predecessor of the sequentiality requirement (termed "temporal completeness") from a deductive (ChronoLog) to an SQL-based language. Snapshot reducibility is achieved because it is included in the requirement. Temporal upward compatibility is not achieved because legacy SQL statements consider the entire database rather than just the current state of the database.

The final two models have been documented much more extensively than its predecessors, but their semantics are still given in an informal SQL-standards format. They were discussed earlier in this section.

7 Summary and Research Directions

The paper's topic is how to seamlessly integrate time into SQL. It takes as its outset a number of syntactic and semantic requirements, motivated by real-world concerns, that a temporal data model and query language must satisfy to contend with legacy applications, permit the coexistence of non-temporal and temporal data, and exploit the programmers' expertise with SQL. Care was taken to make the requirements independent of any particular data model, although we explore them in the context of SQL. No existing model or language satisfies all of these requirements.

The next step was to explore how these requirements shape a concrete temporal extension of SQL, termed ATSQL. A black-box approach was adopted in defining ATSQL, leading to a comprehensive temporal query language that covers core as well as advanced language features, e.g., views, integrity constraints, assertions, data definition, aggregation, duplicates, and coalescing. The language supports both point and interval-based semantics.

The paper first defined ATSQL, emphasizing how the requirements shape the design. Second, a guided tour was given that illustrates how it is possible to smoothly migrate from an SQL–92 system managing non-temporal tables to an ATSQL system that gradually turns the tables temporal, offering advanced temporal query language constructs for managing the resulting

tables. The guided tour and the reader's own statements may be executed on a prototype that is available via world-wide-web. Third, the paper precisely defined the semantics of ATSQL in terms of the semantics of SQL and a mapping from SQL to relational algebra. For this purpose, valid-time, transaction-time, and bitemporal counterparts of the standard relational algebra were defined.

The final step was to study the properties of the language. We verified that it satisfies the requirements posed at the outset of the paper, and then continued by studying the properties of ATSQL not strictly dictated by the requirements. Specifically, while snapshot reducibility is point-based in nature, ATSQL is interval-based and is designed to minimally modify the interval timestamps of argument tuples when computing query results.

Several interesting directions for future research may be pointed out. First, the approach may be generalized to other "dimensions," such as those found in spatial databases, leading to spatio-temporal databases. It also appears promising to study how the proposed concepts generalize to databases annotated with other types of multiple orthogonal dimensions, e.g., those found in data warehousing. In doing so, an important challenge is to provide solutions that are general and yet succeed in supporting well the semantics associated with the specific dimensions.

The notion of temporal upward compatibility makes the implicit assumption that the databases of existing DBMS's contain snapshot data and that a temporal dimension is added to data when the DBMS is replaced with a temporal DBMS. However, this scenario is not exhaustive. Rather, it may be observed that a wide variety of existing databases record time-varying data using regular attributes. The ability to use the novel features of the temporal DBMS depend on the time-varying data being recorded using the designated timestamp attributes. How to semi-automatically migrate application code when the transition to using the designated timestamp attributes occurs is an open problem.

Yet another future direction is the study of efficient implementation techniques. The current ATSQL prototype illustrates the feasibility of the language using a layered architecture [TJB97]. This architecture can be used to identify bottlenecks of current DB technology with respect to temporal database applications. The findings may then prompt the development of new DBMS algorithms. This approach has already been pursued for coalescing [BSS96].

Finally, this study reveals a need for further studies of temporal query language properties. For example, the properties of point-based and interval-based temporal languages that affect their utility in real-life applications are not yet well understood (we thus adopted a safe approach, supporting both semantics in ATSQL).

8 Acknowledgements

We greatly appreciate the contributions of Renato Busatto, Robert Marti, Rick Snodgrass, and Andreas Steiner. Renato worked on the formalization of bitemporal negation and the proof of Theorem 5.1. Robert contributed to early work that shaped the black-box idea adopted for ATSQL. Rick made significant contributions at the design level when we jointly designed the early version of ATSQL that is being proposed for inclusion into the SQL standard. Andreas implemented an initial, running prototype for the language proposed to the SQL standardization committee.

The authors were supported in part by the Danish Natural Science Research Council through grant 9400911 and 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.

References

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[BBJS97] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Wirtschaftsinformatik*, 1997.

[BG93] 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.

[BJS95] 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-Verlag, Berlin.

[BM94] M. Böhlen and R. Marti. On the Completeness of Temporal Database Query Languages. *Proceedings of the First International Conference on Temporal Logic*, pages 283–300, July 1994.

[Böh94] M. Böhlen. *Managing Temporal Knowldege in Deductive Databases.* Ph.D. thesis, Departement für Informatik, ETH Zürich, Switzerland, 1994.

[BSS96] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the Twenty-second International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers, Mumbai (Bombay), India, September 1996.

[CCT93] J. Clifford, A. Croker, and A. Tuzhilin. On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 496–533. Benjamin/Cummings Publishing Company, 1993.

[Cel95] J. Celko. *SQL for Smarties: Advanced SQL Programming.* Morgan Kaufmann Publishers, 1995.

[CG85] S. Ceri, G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering* 11(4):324–345, April 1985.

[CK90] C. C. Chang and H. J. Keisler. *Model Theory.* North-Holland, Amsterdam, 3rd edition, 1990.

[Gad88] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 13(4):418–448, December 1988.

[GB93] 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.

[GN93] S. K. Gadia and S. S. Nair. Temporal Databases: A Prelude to Parametric Data. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 28–66. Benjamin/Cummings Publishing Company, 1993.

[GT91] A. Van Gelder and R. W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[Jac83] M. A. Jackson. *System Development*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1983.

[JCE+94] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia, editors. A Concensus Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52–65, March 1994.

[JSS94] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Models via a Conceptual Model. *Information Systems*, 19(7):513–547, December 1994.

[Llo87] J. W. Lloyd. *Logic Programming*. Symbolic Computation, Springer Verlag, Berlin, 2nd edition, 1987.

[LM96] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, to appear.

[Lor93] N. Lorentzos. The Interval-extended Relational Model and Its Application to Valid-time Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Chapter 3, pages 67–91. Benjamin/Cummings Publishing Company, 1993.

[MS91] L. E. McKenzie and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–543, December 1991.

[MS93] J. Melton and A. R. Simon. *Understanding the new SQL: A Complete Guide*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[NA87] 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.

[NA89] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Information Sciences*, 49:147–175, 1989.

[NA93] S. Navathe and R. Ahmed. Temporal Extensions to the Relational Model and SQL. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 92–109. Benjamin/Cummings Publishing Company, 1993.

[NG93] S. Nair and S. Gadia. Algebraic Optimization in a Relational Model for Temporal Databases. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, Texas, June 1993.

[SA85] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In S. Navathe, editor,*Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 236–246. Austin, Texas, May 1985.

[Sar90] N. Sarda. Algebra and Query Language for a Historical Data Model. *IEEE Computer Journal*, 33(1):11–18, February 1990.

[Sar93] N. Sarda. HSQL: A Historical Query Language. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, 1993.

[SBJS96a] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. ANSI Expert's Contribution, ANSI X3H2-96-501r1, ISO/IEC JTC1/SC21/ WG3 DBL MAD–146r2, International Organization for Standardization, November 1996.

[SBJS96b] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. ANSI Expert's Contribution, ANSI X3H2–96–502r2, ISO/IEC JTC1/SC21/WG3 DBL MCI–147r2, International Organization for Standardization, November, 1996.

[Sch77] B. Schueler. Update Reconsidered. In G. M. Nijssen, editor, *Architecture and Models in Data Base Management Systems*. North Holland Publishing Co., 1977.

[SJS95] M. D. Soo, C. J. Jensen, and R. T. Snodgrass. An Algebra for TSQL2. Chapter 27, pages 505–546 in [Sno95].

[Sno87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno90] R. T. Snodgrass. Temporal Databases: Status and Research Directions. *ACM SIGMOD Record*, 19(4):83–89, December 1990.

[Sno93] R. T. Snodgrass. An Overview of TQuel. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, Chapter 6, pages 141–182. Benjamin/Cummings Publishing Company, 1993.

[Sno95] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.

[TJB97] K. Torp, C. S. Jensen, and M. H. Böhlen. Layered Implementation of Temporal DBMSs—Concepts and Techniques. In *Proceedings of the Fifth International Conference On Database Systems For Advanced Applications*, Melbourne, Australia, April 1997.

[TL82] D. C. Tsichritzis and F. H. Lochovsky. Data Models. In *Software Series*. Prentice-Hall, 1982.

[Tom96] D. Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 58–67, Montreal, Canada, June 1996.

[Wie73] G. Wiederhold. How to Write a Schema for a Time Oriented Medical Record Data Bank. Technical report, Standford University, 1973.

[You82] E. Yourdon. *Managing the System Life Cycle*. Yourdon Press, 1982.

Function	Semantics if r is a valid-time relation
$\tau^{vt}_{tp}(r)$	$\{\langle t\rangle \mid \exists VT\ (\langle t \| VT\rangle \in r \wedge VT\ overlaps\ tp)\}$
$\tau^{tt}_{tp}(r)$	$\{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r\}$
$\delta^{vt}_{per}(r)$	$\{\langle t \| VT\rangle \mid \exists VT'\ (\langle t \| VT'\rangle \in r \wedge VT'\ overlaps\ per \wedge VT = intersect(VT', per))\}$
$\delta^{tt}_{per}(r)$	$\{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r\}$
$SN^{vt}(r)$	$\{\langle t, VT\rangle \mid \langle t \| VT\rangle \in r\}$
$SN^{tt}(r)$	$\{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r\}$

Function	Semantics if r is a transaction-time relation
$\tau^{vt}_{tp}(r)$	$\{\langle t \| TT\rangle \mid \langle t \| TT\rangle \in r\}$
$\tau^{tt}_{tp}(r)$	$\{\langle t\rangle \mid \exists TT\ (\langle t \| TT\rangle \in r \wedge TT\ overlaps\ tp)\}$
$\delta^{vt}_{per}(r)$	$\{\langle t \| TT\rangle \mid \langle t \| TT\rangle \in r\}$
$\delta^{tt}_{per}(r)$	$\{\langle t \| TT\rangle \mid \exists TT'\ (\langle t \| TT'\rangle \in r \wedge TT'\ overlaps\ per \wedge TT = intersect(TT', per))\}$
$SN^{vt}(r)$	$\{\langle t \| TT\rangle \mid \langle t \| TT\rangle \in r\}$
$SN^{tt}(r)$	$\{\langle t, TT\rangle \mid \langle t \| TT\rangle \in r\}$

Function	Semantics if r is a bitemporal relation
$\tau^{vt}_{tp}(r)$	$\{\langle t \| TT\rangle \mid \exists VT\ (\langle t \| VT, TT\rangle \in r \wedge VT\ overlaps\ tp)\}$
$\tau^{tt}_{tp}(r)$	$\{\langle t \| VT\rangle \mid \exists TT\ (\langle t \| VT, TT\rangle \in r \wedge TT\ overlaps\ tp)\}$
$\delta^{vt}_{per}(r)$	$\{\langle t \| VT, TT\rangle \mid \exists VT'\ (\langle t \| VT', TT\rangle \in r \wedge VT'\ overlaps\ per \wedge VT = intersect(VT', per))\}$
$\delta^{tt}_{per}(r)$	$\{\langle t \| VT, TT\rangle \mid \exists TT'\ (\langle t \| VT, TT'\rangle \in r \wedge TT\ overlaps\ per \wedge TT = intersect(TT', per))\}$
$SN^{vt}(r)$	$\{\langle t, VT \| TT\rangle \mid \langle t \| VT, TT\rangle \in r\}$
$SN^{tt}(r)$	$\{\langle t, TT \| VT\rangle \mid \langle t \| VT, TT\rangle \in r\}$

Table 8: Timeslice and Snapshot Operators

A Auxiliary Algebraic Operators

Table 8 defines the auxiliary operators that timeslice relations and turn timestamps into regular, explicit attributes. Unlike the other algebraic operators defined in this paper, these operators are overloaded to apply to valid-time, transaction-time, and bitemporal relations, meaning that the type of the argument relation determines the operation to be performed. This property was exploited to concisely define the semantics of core ATSQL statements, in Table 3.

The functions have variants for both valid and transaction time. For example, the valid-time version of the first timeslice operation, τ^{vt}_{tp}, selects all tuples in the argument relation with a timestamp that overlaps time point *tp*. The time dimension used in this selection is not present in the result relation. If valid time is not supported by the relation, the function degenerates to the identity function.

The second timeslice operation, δ_{per}, returns all argument tuples that overlap with period *per*. The timestamp of a result tuple is the intersection of *per* with the tuple's original timestamp. The snapshot operation SN turns a time dimension into an explicit attribute. This operation is not needed at the implementation level where all attributes are explicit.

B The Bitemporal Relational Algebra

As the valid-time algebra was a natural generalization of the relational algebra, so is the bitemporal algebra a natural generalization of the valid-time algebra. It also respects snapshot reducibility, and it only differs from the other algebra in that it deals with bitemporal rectangles rather than with periods. Bitemporal selection, projection, set union, and join (see Figure 15) are straightforward extensions.

Bitemporal difference is substantially more complex. It is defined in terms of three auxiliary predicates, to be defined below. The idea behind the operator's definition is illustrated in Figure 16, where the large rectangle with a thick frame represents the time region of an r_1-tuple, and the black ones are rectangles associated with value-equivalent r_2-tuples. The result

$$
\begin{aligned}
\sigma_c^{bi}(r) &\triangleq \{\langle t \| VT, TT\rangle \mid \langle t \| VT, TT\rangle \in r \wedge c(\langle t \| VT, TT\rangle)\} \\
\pi_f^{bi}(r) &\triangleq \{\langle t_1 \| VT, TT\rangle \mid \exists t_2\ (\langle t_2 \| VT, TT\rangle \in r \wedge t_1 = f(\langle t_2 \| VT, TT\rangle))\} \\
r_1 \cup^{bi} r_2 &\triangleq \{\langle t \| VT, TT\rangle \mid \langle t \| VT, TT\rangle \in r_1 \vee \langle t \| VT, TT\rangle \in r_2\} \\
r_1 \times^{bi} r_2 &\triangleq \{\langle \langle t_1, VT_1, TT_1\rangle \circ \langle t_2, VT_2, TT_2\rangle \| VT, TT\rangle \mid \\
&\qquad \langle t_1 \| VT_1, TT_1\rangle \in r_1 \wedge \langle t_2 \| VT_2, TT_2\rangle \in r_2 \wedge \\
&\qquad VT = \mathit{intersect}(VT_1, VT_2)\ \wedge TT = \mathit{intersect}(TT_1, TT_2) \wedge \\
&\qquad VT_1\ \mathit{overlaps}\ VT_2 \wedge TT_1\ \mathit{overlaps}\ TT_2\} \\
r_1 \setminus^{bi} r_2 &\triangleq \{\langle t \| VT, TT\rangle \mid \exists VT_1, TT_1(\langle t \| VT_1, TT_1\rangle \in r_1 \wedge \\
&\qquad \mathit{candidate\_tuple}(t, VT, TT, VT_1, TT_1, r_2) \wedge \\
&\qquad \mathit{non\_overlapping}(t, VT, TT, r_2) \wedge \\
&\qquad \mathit{unsplittable}(t, VT, TT, VT_1, TT_1, r_2))
\end{aligned}
$$

Figure 15: The Bitemporal Algebra

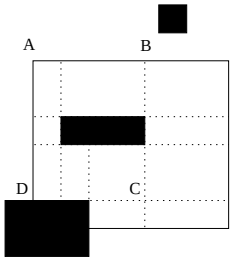


Figure 16: Bitemporal Difference

of the difference $r_1 \setminus^{bi} r_2$ is a set of value-equivalent tuples, one for each of the eleven white rectangles identified by the dashed lines.

The so-called determining time lines associated with r_2-tuples play a crucial role in splitting r_1-tuples and thus in defining the result tuples. Determining time lines start at each vertex of an r_2-tuple, and they extend until they are blocked by a value-equivalent r_2-tuple or until they reach the border of the r_1-tuple. Before explaining the issues in more detail, it is convenient to first introduce some terminology. Each bitemporal tuple has associated a timestamp that encodes a rectangular region in the space spanned by transaction time and valid time. This region, we term the tuple's *time rectangle*, and the rectangle corners are termed *time vertices*. Their coordinates are the tuple's *time coordinates* which thus correspond to the tuple's transaction and valid time. The rectangle sides are *time edges* Finally, a *determining time line* is a vertical or horizontal line segment that originates from some time vertex. We omit the modifier "time" from these terms when no confusion results.

The definition in Figure 15 identifies three requirements to a result tuple X.

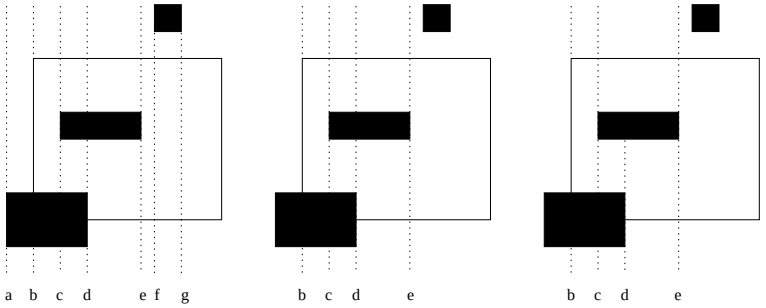
1. The time coordinates of X are derived either from the time coordinates of a (value-equivalent) r_1-tuple, or from (value-equivalent) r_2-tuples that satisfy two restrictions:
   (a) They must temporally overlap with the r_1-tuple whose time rectangle contains the time rectangle of X, and
   (b) the time vertices of X must have direct access to the originating r_2-tuple vertices, meaning that no value-equivalent r_2-tuple lies between originating and resulting vertices.

2. X does not temporally overlap with any value-equivalent r_2-tuple.
3. No determining time lines defined by r_2-tuples that are value-equivalent to X cross its time rectangle.

The first requirement, represented by the predicate *candidate_tuple*, is defined as a conjunction of four subformulas, each of which constrains one of the time vertices of a tuple $\langle t \| VT, TT \rangle$ of $r_1 \setminus^{bi} r_2$.

$$
\begin{aligned}
&\mathit{candidate\_tuple}(t, VT, TT, VT_1, TT_1, r_2) \equiv \\
&\quad TT^- = TT_1^- \vee \\
&\quad \exists VT_2, TT_2(\langle t \| VT_2, TT_2 \rangle \in r_2 \wedge (TT^- = TT_2^- \vee TT^- = TT_2^+) \wedge \\
&\qquad TT_1^- \leq TT^- < TT_1^+ \wedge VT_1^- < VT_2^+ \wedge VT_2^- < VT_1^+ \wedge \\
&\qquad \neg \exists VT_2, TT_2(\langle t \| VT_2, TT_2 \rangle \in r_2 \wedge TT_2^- < TT^- < TT_2^+ \wedge \\
&\qquad\quad (VT_2^+ < VT_2^+ \leq VT^- \vee VT_2^- > VT_2^- \geq VT^+))) \wedge \\
&\\
&\quad TT^+ = TT_1^+ \vee \\
&\quad \exists VT_3, TT_3(\langle t \| VT_3, TT_3 \rangle \in r_2 \wedge (TT^+ = TT_3^- \vee TT^+ = TT_3^+) \wedge \\
&\qquad TT_1^- < TT^+ \leq TT_1^+ \wedge VT_1^- < VT_3^+ \wedge VT_3^- < VT_1^+ \wedge \\
&\qquad \neg \exists VT_3, TT_3(\langle t \| VT_3, TT_3 \rangle \in r_2 \wedge TT_3^- < TT^+ < TT_3^+ \wedge \\
&\qquad\quad (VT_3^+ < VT_3^+ \leq VT^- \vee VT_3^- > VT_3^- \geq VT^+))) \wedge \\
&\\
&\quad VT^- = VT_1^- \vee \\
&\quad \exists VT_4, TT_4(\langle t \| VT_4, TT_4 \rangle \in r_2 \wedge (VT^- = VT_4^- \vee VT^- = VT_4^+) \wedge \\
&\qquad VT_1^- \leq VT^- < VT_1^+ \wedge TT_1^- < TT_4^+ \wedge TT_4^- < TT_1^+ \wedge \\
&\qquad \neg \exists VT_4, TT_4(\langle t \| VT_4, TT_4 \rangle \in r_2 \wedge VT_4^- < VT^- < VT_4^+ \wedge \\
&\qquad\quad (TT_4^+ < TT_4^+ \leq TT^- \vee TT_4^- > TT_4^- \geq TT^+))) \wedge \\
&\\
&\quad VT^+ = VT_1^+ \vee \\
&\quad \exists VT_5, TT_5(\langle t \| VT_5, TT_5 \rangle \in r_2 \wedge (VT^+ = VT_5^- \vee VT^+ = VT_5^+) \wedge \\
&\qquad VT_1^- < VT^+ \leq VT_1^+ \wedge TT_1^- < TT_5^+ \wedge TT_5^- < TT_1^+ \wedge \\
&\qquad \neg \exists VT_5, TT_5(\langle t \| VT_5, TT_5 \rangle \in r_2 \wedge VT_5^- < VT^+ < VT_5^+ \wedge \\
&\qquad\quad (TT_5^+ < TT_5^+ \leq TT^- \vee TT_5^- > TT_5^- \geq TT^+)))
\end{aligned}
$$

The first two lines of the first conjunct have a *generative* purpose, since they identify a collection of candidate transaction-time start values for $\langle t \| VT, TT \rangle$. The left-most diagram below illustrates all such candidate values for the relation depicted in Figure 16:



Not all time lines originating from r_2-tuples provide suitable time coordinates, though. The subsequent three lines of the definition eliminate some of the undesirable ones. Specifically, the third line requires the time edge of an r_2-tuple that generates a candidate transaction time value to overlap the time rectangle of the relevant r_1-tuple. Time lines a, f and g above must then be dropped, as indicated in the middle diagram.

The fourth and fifth lines account for the blocking effect of r_2-tuples on candidate time lines. In the example, the upper part of line d is inadequate. The lines that meet all the restrictions,

indicated in the right-most diagram, correspond to determining time lines for r_1 and r_2, provided they are confined to the time rectangle of the r_1-tuple.

The remaining conjuncts of *candidate_tuple* impose equivalent constraints on each of the other time coordinates of $r_1 \setminus^{bi} r_2$-tuples.

Next, the *non overlapping* of $r_1 \setminus^{bi} r_2$- and r_2-tuples is enforced by the following predicate.

$$\begin{aligned}&\mathit{non\_overlapping}(t, VT, TT, r_2) \equiv \\ &\quad \forall VT_2, TT_2(\langle t \| VT_2, TT_2\rangle \in r_2 \Rightarrow (VT^+ \leq VT_2^- \vee VT_2^+ \leq VT^- \vee TT^+ \leq TT_2^- \vee TT_2^+ \leq TT^-))\end{aligned}$$

In the example, this predicate excludes the (aggregate) time rectangle ABCD in Figure 16, since it contains the rectangle of an r_2-tuple.

Finally, the partition of the time rectangle of an r_1-tuple must be *maximal* according to the previous restrictions, i.e., there should not be any additional determining time lines splitting a time rectangle of a tuple in $r_1 \setminus^{bi} r_2$. This can be ensured by requiring that, whenever the time edge of an r_2-tuple could originate an additional splitting line, there should exist another r_2-tuple blocking its effect:

$$\begin{aligned}&\mathit{unsplittable}(\langle t \| VT, TT\rangle, \langle t \| VT_1, TT_1\rangle, r_2) \equiv \\ &\quad \forall VT_2, TT_2, tt(\langle t \| VT_2, TT_2\rangle \in r_2 \wedge (tt = TT_2^- \vee tt = TT_2^+) \wedge TT^- < tt < TT^+ \wedge \\ &\qquad (VT_1^- < VT_2^- < VT_1^+ \vee VT_1^- < VT_2^+ < VT_1^+) \Rightarrow \\ &\qquad\quad \exists VT_2, TT_2(\langle t \| VT_2, TT_2\rangle \in r_2 \wedge TT_2^- < tt < TT_2^+ \wedge \\ &\qquad\qquad (VT_2^+ < VT_2^+ \leq VT^- \vee VT_2^- > VT_2^- \geq VT^+))) \wedge \\ &\quad \forall VT_3, TT_3, vt(\langle t \| VT_3, TT_3\rangle \in r_2 \wedge (vt = VT_3^- \vee vt = VT_3^+) \wedge VT^- < vt < VT^+ \wedge \\ &\qquad (TT_1^- < TT_3^- < TT_1^+ \vee TT_1^- < TT_3^+ < TT_1^+) \Rightarrow \\ &\qquad\quad \exists VT_3, TT_3(\langle t \| VT_3, TT_3\rangle \in r_2 \wedge VT_3^- < vt < VT_3^+ \wedge \\ &\qquad\qquad (TT_3^+ < TT_3^+ \leq TT^- \vee TT_3^- > TT_3^- \geq TT^+)))\end{aligned}$$

Hence, ABCD in the figure also does not qualify as the time rectangle of an $r_1 \setminus^{bi} r_2$-tuple because various (unblocked) determining time lines split it into seven rectangles.

Finally, we define coalescing of bitemporal relations. Transaction-time coalescing, $coal_{tt}^{bi}$, denoted in ATSQL queries by `(TRANSACTION)`, guarantees maximal transaction-time periods and is illustrated in Figure 17. The five white rectangles illustrate the times of five value-

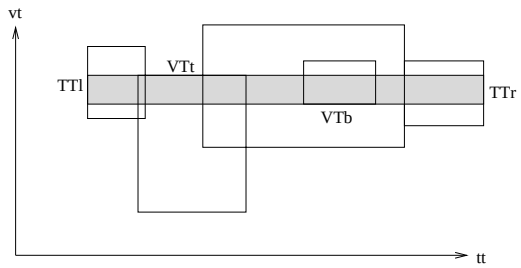


Figure 17: Transaction Time Coalescing of a Bitemporal Relation

equivalent tuples in the uncoalesced relation. The gray tuple is one of the tuples resulting from coalescing. (Transaction-time coalescing ensures maximal expansion in the transaction-time dimension and yields no coalescing in the valid-time dimension.)

Formally, coalescing is defined as follows.

$$
\begin{array}{l}
coal^{bi}_{tt}(r) \triangleq \{\langle t \| VT, TT\rangle \mid \\
\quad \exists VT_1, TT_r(\langle t \| VT_1, TT_r\rangle \in r \wedge TT^+ = TT^+_r) \wedge \\
\quad \exists VT_2, TT_l(\langle t \| VT_2, TT_l\rangle \in r \wedge TT^- = TT^-_l) \wedge \\
\quad \exists VT_t, TT_3(\langle t \| VT_t, TT_3\rangle \in r \wedge (VT^+ = VT^-_t \vee VT^+ = VT^+_t) \wedge TT^-_3 < TT^+ \wedge TT^+_3 > TT^-) \wedge \\
\quad \exists VT_b, TT_4(\langle t \| VT_b, TT_4\rangle \in r \wedge (VT^- = VT^+_b \vee VT^- = VT^-_b) \wedge TT^-_4 < TT^+ \wedge TT^+_4 > TT^-) \wedge \\
\quad \neg\exists VT_5, TT_5(\langle t \| VT_5, TT_5\rangle \in r \wedge (VT^- < VT^+_5 < VT^+ \vee VT^- < VT^-_5 < VT^+) \wedge \\
\qquad\qquad TT^-_5 \leq TT^+ \wedge TT^+_5 \geq TT^-) \wedge \\
\quad \forall VT_6, TT_6(\langle t \| VT_6, TT_6\rangle \in r \ \wedge TT^- \leq TT^-_6 < TT^+ \wedge VT^-_6 \leq VT^- \wedge VT^+_6 \geq VT^+ \Rightarrow \\
\qquad\qquad \exists VT_7, TT_7(\langle t \| VT_7, TT_7\rangle \in r \wedge TT^-_7 < TT^-_6 \leq TT^+_7 \wedge \\
\qquad\qquad\qquad VT^-_7 \leq VT^- \wedge VT^+_7 \geq VT^+)) \wedge \\
\quad \neg\exists VT_8, TT_8(\langle t \| VT_8, TT_8\rangle \in r \wedge (TT^-_8 < TT^- \leq TT^+_8 \vee TT^-_8 \leq TT^+ < TT^+_8) \wedge \\
\qquad\qquad VT^-_8 < VT^+ \wedge VT^+_8 > VT^-)\}
\end{array}
$$

In the first two lines we search for two tuples defining the transaction-time start (TT^-_l) and the transaction-time end (TT^+_r) of a coalesced tuple. In lines 3 and 4, we do the same for valid-time start and end. Lines 5 and 6 ensure that no coalescing in the valid-time dimension is done, i.e., the extension in the valid-time dimension is as small as possible. Lines 7 to 9 ensure that there are no holes, i.e., *all* tuples with a transaction-time start contained in the final maximal transaction time must be covered by another tuple. The last two lines ensure that we get maximal extensions in transaction time, i.e., that no tuple exists that could possibly extend the tuple further.

Valid-time coalescing of a bitemporal relation r, $coal^{bi}_{vt}(r)$, follows the same principle, the only difference being that the roles of valid and transaction time are reversed. The definition is thus omitted.

C Proof of Theorem 5.1

To prove Theorem 5.1, we consider each equivalence in turn. The two sides of the equivalence for selection are defined as follows.

$$
\begin{array}{lcl}
\tau^{vt}_{tp}(\sigma^{vt}_{c}(r)) & = & \{t \mid \langle t \| VT\rangle \in r \wedge c(\langle t, VT\rangle) \wedge VT \textit{ overlaps } tp\} \\
\sigma_c(\tau^{vt}_{tp}(r)) & = & \{t \mid \langle t \| VT\rangle \in r \wedge VT \textit{ overlaps } tp \wedge c(t)\}
\end{array}
$$

To show that these definitions are equivalent, we first exploit the commutativity of conjunction to rewrite "VT *overlaps* $tp \wedge c(t)$" to "$c(t) \wedge VT$ *overlaps* tp." What remains is to prove that $c(\langle t, VT\rangle)$ and $c(t)$ are equivalent. The same predicate c occurs on both sides of the equality, and since the formulation of the theorem disallows the use of VT in predicate c, the equality and thus the first equivalence follows.

The equivalence for projections follows similarly.

$$
\begin{array}{lcl}
\tau^{vt}_{tp}(\pi^{vt}_{f}(r)) & = & \{t_1 \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge t_1 = f(\langle t_2, VT\rangle)) \wedge VT \textit{ overlaps } tp\} \\
\pi_f(\tau^{vt}_{tp}(r)) & = & \{t_1 \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge VT \textit{ overlaps } tp \wedge t_1 = f(t_2))\}
\end{array}
$$

The only difference with respect to selection is that we are dealing with a projection function rather than with a selection predicate. Similarly to before, we first commute two terms and then observe that VT may be omitted as an argument of f because the use of f is disallowed in the theorem, meaning that $(\langle t_2, VT\rangle)$ and $f(t_2)$ are equivalent.

Considering the union operators, we once again apply the definitions of the operators involved to the two sides.

$$
\begin{array}{lcl}
\tau^{vt}_{tp}(r_1 \cup^{vt} r_2) & = & \{t \mid (\langle t \| VT\rangle \in r_1 \vee \langle t \| VT\rangle \in r_2) \wedge VT \textit{ overlaps } tp\} \\
\tau^{vt}_{tp}(r_1) \cup \tau^{vt}_{tp}(r_2) & = & \{t \mid (\langle t \| VT\rangle \in r_1 \wedge VT \textit{ overlaps } tp) \vee (\langle t \| VT\rangle \in r_2 \wedge VT \textit{ overlaps } tp)\}
\end{array}
$$

Transforming the first formula into disjunctive normal form proves the equivalence.

The equivalence involving the Cartesian products is somewhat more complicated to prove.

$$\pi^{-}_{r_1.VT,r_2.VT}(\tau^{vt}_{tp}(r_1 \times^{vt}_{c} r_2)) = \{t_1 \circ t_2 \mid \langle t_1 \| VT_1 \rangle \in r_1 \wedge \langle t_2 \| VT_2 \rangle \in r_2 \wedge VT_1 \; overlaps \; VT_2 \wedge VT = intersect(VT_1, VT_2) \wedge VT \; overlaps \; tp\}$$

$$\tau^{vt}_{tp}(r_1) \times_c \tau^{vt}_{tp}(r_2) = \{t_1 \circ t_2 \mid \langle t_1 \| VT_1 \rangle \in r_1 \wedge VT_1 \; overlaps \; tp \wedge \langle t_2 \| VT_2 \rangle \in r_2 \wedge VT_2 \; overlaps \; tp\}$$

After the usual initial reordering of the terms of the formula, we are left with the proof of the equivalence between "$VT_1 \; overlaps \; VT_2 \wedge VT = intersect(VT_1, VT_2) \wedge VT \; overlaps \; tp$" and "$VT_1 \; overlaps \; tp \wedge VT_2 \; overlaps \; tp$." We consider each formula in turn.

$$VT_1 \; overlaps \; VT_2 \wedge VT = intersect(VT_1, VT_2) \wedge VT \; overlaps \; tp$$
$$\Downarrow$$
(elimination of VT)
$$\Downarrow$$
$$VT_1 \; overlaps \; VT_2 \wedge intersect(VT_1, VT_2) \; overlaps \; tp$$
$$\Downarrow$$
(replace periods with points, cf. Table 2)
$$\Downarrow$$
$$VT_1^+ > VT_2^- \wedge VT_2^+ > VT_1^- \wedge max(VT_1^-, VT_2^-) < tp \wedge min(VT_1^+, VT_2^+) > tp$$
$$\Downarrow$$
$$(max(A, B) < C \equiv A < C \wedge B < C)$$
$$(min(A, B) > C \equiv A > C \wedge B > C)$$
$$\Downarrow$$
$$VT_1^+ > VT_2^- \wedge VT_2^+ > VT_1^- \wedge VT_1^- < tp \wedge VT_2^- < tp \wedge VT_1^+ > tp \wedge VT_2^+ > tp$$

Next we rewrite the second formula.

$$VT_1 \; overlaps \; tp \wedge VT_2 \; overlaps \; tp$$
$$\Downarrow$$
(replace periods with points, cf. Table 2)
$$\Downarrow$$
$$VT_1^- < tp \wedge VT_1^+ > tp \wedge VT_2^- < tp \wedge VT_2^+ > tp$$
$$\Downarrow$$
$$(A < C \wedge B > C \Rightarrow B > A)$$
$$\Downarrow$$
$$VT_1^- < tp \wedge VT_1^+ > tp \wedge VT_2^- < tp \wedge VT_2^+ > tp \wedge VT_1^+ > VT_2^- \wedge VT_2^+ > VT_1^-$$

Apart from the order of the terms, the rewritten formulas are identical.

The final equivalence involves valid-time difference:

$$\begin{aligned} \tau^{vt}_{tp}(r_1 \setminus^{vt} r_2) &= \{t \mid \phi_1\} \\ \tau^{vt}_{tp}(r_1) \setminus \tau^{vt}_{tp}(r_2) &= \{t \mid \phi_2\} \end{aligned}$$

where ϕ_1 is defined as

$$\begin{aligned} &\exists VT, VT_1 \\ &\quad \langle t \| VT_1 \rangle \in r_1 \wedge \\ &\quad (\exists VT_2(\langle t \| VT_2 \rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\ &\quad (\exists VT_3(\langle t \| VT_3 \rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\ &\quad VT^- < VT^+ \wedge \\ &\quad \neg \exists VT_4(\langle t \| VT_4 \rangle \in r_2 \; \wedge VT_4 \; overlaps \; VT) \wedge \\ &\quad VT \; overlaps \; tp\} \end{aligned}$$

and ϕ_2 is defined as

$$\underbrace{\exists VT_1(\langle t \| VT_1\rangle \in r_1 \wedge VT_1 \textit{ overlaps } tp)}_{\psi_1} \wedge \underbrace{\neg\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_2 \textit{ overlaps } tp)}_{\psi_2} .$$

To prove the two sets equivalent, we have to show that the defining formulas are equivalent, i.e., $\phi_1 \equiv \phi_2$. We do so by proving two implications, i.e., $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \Leftarrow \phi_2$.

A) $(\phi_1 \Rightarrow \phi_2)$ With $\phi_2 \equiv \psi_1 \wedge \psi_2$ we can rewrite $\phi_1 \Rightarrow \phi_2$ to $(\phi_1 \Rightarrow \psi_1) \wedge (\phi_1 \Rightarrow \psi_2)$ and prove each of the conjuncts in turn. The proof is based on the following theorems.

T_1	$\sigma_1 \Rightarrow \sigma_2$	$\Rightarrow$	$(\sigma_1 \wedge \sigma_3) \Rightarrow \sigma_2$
T_2	$\sigma_1 \Rightarrow \sigma_2$	$\Rightarrow$	$(\sigma_3 \wedge \sigma_1) \Rightarrow (\sigma_3 \wedge \sigma_2)$
T_3	$\sigma_1 \Rightarrow (\sigma_2 \Rightarrow \sigma_3)$	$\equiv$	$(\sigma_1 \wedge \sigma_2) \Rightarrow \sigma_3$
T_4	$(\sigma_1 \Rightarrow \sigma_2) \wedge (\sigma_3 \Rightarrow \sigma_4)$	$\Rightarrow$	$(\sigma_1 \wedge \sigma_3) \Rightarrow (\sigma_2 \wedge \sigma_4)$
T_5	$\sigma_1 \Rightarrow \forall v\ \sigma_2$	$\equiv$	$\forall v(\sigma_1 \Rightarrow \sigma_2)$ if v does not occur in σ_1
T_6	$X \subseteq Y$	$\equiv$	$\forall z(z \in X \Rightarrow z \in Y)$
T_7	$VT_2 \subseteq VT_1$	$\equiv$	$VT_1^- \leq VT_2^- \wedge VT_2^+ \leq VT_1^+$
T_8	$tp \in VT$	$\equiv$	VT *overlaps* tp
T_9	If $\Delta \vdash (\phi \Rightarrow \psi)$, then $\Delta \vdash ((\exists v\phi) \Rightarrow (\exists v\psi))$		
T_{10}	$\Delta \vdash \forall v\phi$	iff	$\Delta \vdash \phi$

The first sub-proof starts with two formulas that are trivially true.

(1) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \Rightarrow$
$VT_1^- \leq VT^-$

(2) $(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \Rightarrow$
$VT^+ \leq VT_1^+$

We then apply the above theorems until $\phi_1 \Rightarrow \psi_1$ results. (With each intermediate formula, we indicate the formulas and theorems that were used deriving it.)

(3) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \Rightarrow$
$VT \subseteq VT_1$ $\quad (1), (2), T_4, T_7$

(4) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge$
$VT^- < VT^+ \wedge$
$\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4$ *overlaps* $VT) \Rightarrow$
$VT \subseteq VT_1$ $\quad (3), T_1$

(5) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge$
$VT^- < VT^+ \wedge$
$\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4$ *overlaps* $VT) \wedge$
$tp \in VT \Rightarrow$
$tp \in VT_1$ $\quad (4), T_6, T_5, T_{10}, T_3$

(6) $\langle t \| VT_1\rangle \in r_1 \wedge$
$(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge$
$VT^- < VT^+ \wedge$
$\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4$ *overlaps* $VT) \wedge$
VT *overlaps* $tp \Rightarrow$
$\langle t \| VT_1\rangle \in r_1 \wedge VT_1$ *overlaps* tp $\quad (5), T_2, T_8$

The introduction of existential quantifiers (T_9) completes the proof.

To prove $\phi_1 \Rightarrow \psi_2$ we start out with the formula below:

$$\begin{array}{l}\neg\exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_4 \textit{ overlaps } VT) \wedge \\ VT \textit{ overlaps } tp \;\Rightarrow \\ \quad \neg\exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_4 \textit{ overlaps } tp)\end{array}$$

Again, it is easy to see that the formula is trivially true. Next, we apply T_1 followed by T_9 to get

$$\begin{array}{l}\exists VT, VT_1 \\ \quad \langle t\|VT_1\rangle \in r_1 \wedge \\ \quad (\exists VT_2(\langle t\|VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\ \quad (\exists VT_3(\langle t\|VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\ \quad VT^- < VT^+ \wedge \\ \quad \neg\exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_4 \textit{ overlaps } VT) \wedge \\ \quad VT \textit{ overlaps } tp \;\Rightarrow \\ \quad\quad \neg\exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_4 \textit{ overlaps } tp)\end{array}$$

Renaming of a bound variable yields $\phi_1 \Rightarrow \psi_2$.

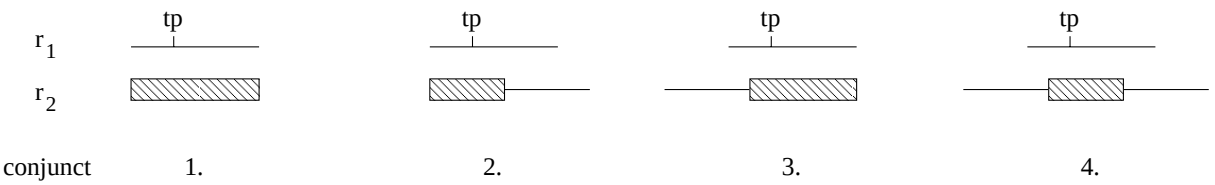
B) $(\phi_1 \Leftarrow \phi_2)$ We prove $\phi_2 \Rightarrow \phi_1$ by reduction to absurdity, i.e., we show that $\neg\phi_1 \wedge \phi_2$ leads to a contradiction. We start with $\neg\phi_1$:

$$\begin{array}{l}\forall VT, VT_1 \\ \quad (\langle t\|VT_1\rangle \in r_1 \wedge \\ \quad (\exists VT_2(\langle t\|VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\ \quad (\exists VT_3(\langle t\|VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\ \quad VT^- < VT^+ \wedge \\ \quad VT \textit{ overlaps } tp \Rightarrow \\ \quad\quad \exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT \textit{ overlaps } VT_4))\end{array}$$

We first apply standard normalization rules [Llo87, p.113] and quantifier elimination [CK90, p.49–58] to get ϕ_3:

$$\begin{array}{l}\forall VT_1(\langle t\|VT_1\rangle \in r_1 \wedge VT_1^- \leq tp < VT_1^+ \Rightarrow \\ \quad \exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_1^+ > VT_4^- \wedge VT_4^+ > VT_1^-)) \wedge \\ \forall VT_1, VT_2(\langle t\|VT_1\rangle \in r_1 \wedge \langle t\|VT_2\rangle \in r_2 \wedge VT_1^- \leq tp < VT_2^- \leq VT_1^+ \Rightarrow \\ \quad \exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_2^- > VT_4^- \wedge VT_4^+ > VT_1^-)) \wedge \\ \forall VT_1, VT_2(\langle t\|VT_1\rangle \in r_1 \wedge \langle t\|VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \leq tp < VT_1^+ \Rightarrow \\ \quad \exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_1^+ > VT_4^- \wedge VT_4^+ > VT_2^+)) \wedge \\ \forall VT_1, VT_2, VT_3(\langle t\|VT_1\rangle \in r_1 \wedge \langle t\|VT_2\rangle \in r_2 \wedge \langle t\|VT_3\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \leq tp < VT_3^- \leq VT_1^+ \Rightarrow \\ \quad \exists VT_4(\langle t\|VT_4\rangle \in r_2 \wedge VT_3^- > VT_4^- \wedge VT_4^+ > VT_2^+))\end{array}$$

Each conjunct of ϕ_3 is represented in the diagram below. The solid lines represent the (times of the) first part of a conjunct, i.e., the part before the implication, whereas the grey rectangles indicate the time range that must be overlapped by yet another r_2-tuple (the second part of the conjunct, i.e., the part that follows the implication).



From ψ_1 and the first conjunct of ϕ_3, it follows that a) there is an r_1-tuple $x_1 = \langle t \| VT_1 \rangle$ such that $tp \in VT_1$ and b) there is an r_2-tuple that temporally overlaps with x_1. Since, according to ψ_2 no r_2-tuple contains tp, the overlap must be of the form depicted in either the second or third diagram. Both cases imply that yet another r_2 tuple exists that overlaps the time period indicated by the gray rectangle. Depending on the time period of r_2, we end up with a situation represented by diagram two, three, or four. (Because of ψ_2, another overlapping is impossible.) Whatever situation it would imply yet another r_2 tuple which is (timely) closer to tp. No finite relation r_2 can fulfill this requirement.