Evaluating and Enhancing the Completeness of TSQL2

Michael H. Böhlen[1]
Christian S. Jensen[2]
Richard T. Snodgrass[1]

TR 95-05

June 28, 1995

Abstract

The question of what is a well-designed temporal data model and query language is a difficult, but also an important one. The consensus temporal query language TSQL2 attempts to take advantage of the accumulated knowledge gained from designing and studying many of the earlier models and languages. In this sense, TSQL2 represents a constructive answer to this question. Others have provided analytical answers by developing criteria, formulated as completeness properties, for what is a good model and language.

This paper applies important existing completeness notions to TSQL2 in order to evaluate the design of TSQL2. It is shown that TSQL2 satisfies only a subset of these completeness notions. In response to this, a minimally modified version of TSQL2, termed *Applied TSQL2*, is proposed; this new language satisfies the notions of temporal semi-completeness and completeness which are not satisfied by TSQL2. An outline of the formal semantics for Applied TSQL2 is given.

[1]Department of Computer Science
University of Arizona
Tucson, AZ 85721
`{boehlen, rts}@cs.arizona.edu`
[2]Department of Mathematics and Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK–9220 Aalborg Ø, DENMARK
`csj@iesd.auc.dk`

Evaluating and Enhancing the Completeness of TSQL2

Contents

1 Introduction

The temporal database community has been prolific in its production of temporal data models and query languages. Over the past fifteen years, more than two dozen temporal relational data models have been proposed, each with one or more associated query languages [ÖS95]. This has left the community with a wide, confusing—but also challenging—variety of alternatives.

As one response to this state of affairs, a committee of eighteen temporal database researchers has recently released the TSQL2 Language Specification [SAA+94a], which defines a temporal extension to the SQL–92 standard [SQL92, MS93]. TSQL2 was created partly in an attempt to consolidate, in a single consensual model and language, the insights and experiences gained from the development of the previous data models and languages.

As a quite different approach, other efforts (e.g., [CCT93, CCT94, MS91, Böh94, BM94]) have put focus on the properties of temporal data models and query languages, as well as on the design alternatives available when developing these. This has led to precise definitions of model and language properties that can be used to characterize and evaluate the many models and languages. In the spirit of Codd's original definition of relational completeness [Cod72], some of these properties have been stated as different kinds of completeness.

It then seems appropriate to use the body of work on model and language properties to study the design of TSQL2—this paper does exactly that. Further, it attempts to rectify any deficiencies found. It is a fundamental assumption of the paper that when evaluating a data model and query language, both the functionality and the syntax for expressing a certain functionality are important. The completeness notions that we adopt in the investigation thus include both functionality-related and syntactical criteria.

Specifically, we formalize the notion of a data model being *upwards compatible* with another data model and show that TSQL2 is upwards compatible with SQL–92. Briefly, this means that a smooth transition from SQL–92 to TSQL2 is possible.

One of the most widely cited distinctions among temporal data models is that between first normal form and non-first normal form models. This distinction has been formally captured by the concepts of *temporally ungrouped* and *grouped* data models [CCT93, CCT94]. We show that TSQL2 is temporally ungrouped and not temporally grouped. As this property is inherent in the model, we do not propose to change it. Rather, we put focus on the implications of a model being ungrouped or grouped.

The last two notions of completeness considered are *temporal semi-completeness* and the more restrictive *temporal completeness* [Böh94, BM94]. The former notion essentially states that a temporal relational data model must contain temporal generalizations of all snapshot relations and queries. Further, the generalized queries must be syntactically similar to the snapshot queries they generalize. Temporal completeness adds further functional and syntactical requirements, accounting for queries that cannot be answered by temporally semi-complete languages. It is shown that TSQL2 does not fully satisfy these completeness notions. This leads to the design of *Applied TSQL2*, a minimally modified version of TSQL2 that satisfies both notions. While the paper will review the relevant concepts and syntactic constructs of TSQL2, the reader is encouraged to consult the extensive commentaries on the language design[1] for a full understanding of the implications of the properties we prove and the modifications we propose.

Related work on completeness has been primarily in the context of non-temporal databases. It is possible to distinguish two basic approaches. The first one takes a particular calculus (usually first order relational calculus) as a metric. Any language having at least the expressive power of

[1]The language specification and the associated commentaries are available via anonymous ftp from `FTP.cs.arizona.edu:tsql/tsql2/tsql2.final`.

the calculus is said to be complete. Original work along these lines was done by Codd for relational databases [Cod72]. There have been generalizations for entity-relationship databases [AC81] and for temporal databases [TC90]. One inherent problem with these approaches is the degree of appropriateness of the calculus that is used as a metric. There is no guarantee that the calculus captures *all* reasonable queries. For example, it has been shown [AU79] that first order relational calculus cannot express the transitive closure of binary relations.

The second approach is to define an appropriately large set of queries and require query languages to express all queries in this set. This kind of completeness was investigated by Bancilhon [Ban78] and Chandra and Harel [CH80]. The definitions of temporal semi-completeness and temporal completeness are somewhat in this spirit. They (in particular temporal semi-completeness) take the set of queries that are expressible by a non-temporal language as a reference and ensure that temporal generalizations of the non-temporal language can express all these queries. Additionally, they establish syntactic restrictions a temporal language must obey.

Briefly, the contributions of the paper are threefold. First, the paper further formalizes some existing definitions of completeness of relevance for temporal data models and query languages, namely the notions of upwards compatibility, temporal semi-completeness and temporal completeness. Second, the paper explores the design of TSQL2 by applying these completeness notions and the notion of temporal (un)groupedness to TSQL2. It is shown that TSQL2 satisfies some of these notions, but does not satisfy all of them. The third contribution is a constructive rectification of the identified deficiencies, leading to a "complete" TSQL2, Applied TSQL2.

The paper is structured as follows. Each of Sections 2–5 first defines a particular type of completeness. They then evaluate the completeness of TSQL2 in the context of each completeness notion. During this investigation, some deficiencies of TSQL2 are uncovered that are subsequently addressed in Section 6. This section proposes constructive solutions to each deficiency, leading to the "applied" version of TSQL2. Finally, Section 7 provides an outline of a core semantics for Applied TSQL2. Section 8 summarizes the paper and points to directions for future research.

2 Upwards Compatibility

Completeness is generally a relative property of a data model or a query language. Thus, a model or language satisfies some notion of "completeness" if it is related to another model or query language in a certain way. In this section, we introduce the first of the three types of completeness. Specifically, we formalize the notion that a data model is upwards compatible with respect to another data model. We subsequently consider the upwards compatibility of SQL–92 with respect to TSQL2.

2.1 Definitions

When a new database management system, with an associated data model, is introduced into an organization, often that system replaces an existing system, also with an associated data model. For software engineering reasons, to be discussed in more detail below, it is an important property that the existing data model be upwards (or, forwards) compatible with the new data model. Put differently, the new data model should be a strict superset of the existing 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. For example, the central data structure of the relational model is the relation, and the central, user-level query language is SQL. 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 expressible by M, and QL is the set of all queries in M that may be formulated on some database in M. We will use db to denote a database and q to denote a query.

Definition 2.1 (upwards compatibility) Let $M_1 = (DS_1, QL_1)$ and $M_2 = (DS_2, QL_2)$ be two data models. Model M_1 is *upwards compatible with* model M_2 if and only if

- DS_1 is a superset of DS_2, and
- for each instance db in $DS_1 \cap DS_2$ (i.e., in DS_2) and for each query expression q in QL_2, q is also a legal query expression in QL_1, and the results of evaluating q on db is the same in M_1 and M_2. ▮

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 in the old system are also legal databases in the new system and thus need not be modified when the new system is adopted. The second condition guarantees that existing queries will remain legal and will 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.

The definition of upwards compatibility is related to the traditional notion of Codd completeness (Codd originally used the term relational completeness) [Cod72], as formulated in the context of the standard relational model. To see the similarity and differences, we review that completeness notion.

Essentially, a relational or extended relational data model is Codd complete if all queries that can be formulated on arbitrary conventional relations expressible in the model are a superset of all relational algebra queries that can be formulated.

Definition 2.2 (Codd completeness) Let $M = (DS, QL)$ be some data model, and let (SR, RA) be the relational model with the relational algebra as its query language. Model M is *Codd complete* if and only if each query in RA has an equivalent counterpart in QL when all db in $DS \cap SR$ are considered. ▮

Two query expressions are equivalent if they always yield mutually identical results when supplied identical arguments. The relational algebra comes in numerous versions[2], and while the definition is dependent on a particular version being chosen, it is not important at this point what particular version is chosen.

The similarity between upwards compatibility and Codd completeness is apparent, but there are also important differences. First, Codd completeness is restricted to use the relational algebra as a yardstick for measuring the expressive power of other query languages. Thus, the relevance of Codd completeness is dependent on how "natural" or well-chosen the relational algebra is. On the other hand, upwards compatibility is not tied to any particular data model.

Second, Codd completeness strictly concerns functionality while upwards compatibility concerns both functionality and the syntax for expressing the functionality. Specifically, Codd completeness is defined in terms of the existence of *equivalent*, but possibly different, query expressions. Upwards compatibility requires query expressions that yield identical results to also be syntactically *identical*. Thus, a model being upwards compatible with the relational model/algebra is a stronger criterion than the model being Codd complete.

[2]The relational algebra used in conjunction with the original definition of Codd completeness [Cod72] included "cartesian [sic] product," "union," "intersection," "difference," "projection," "θ-join," "division," and "restriction" (a special case of selection).

2.2 Upwards Compatibility among SQL–92 and TSQL2

Clearly, it is an important property for a new data model, such as TSQL2, to be a strict superset of the data model it is intended to supersede, i.e., SQL–92. We now consider this issue.

In TSQL2, there are six kinds of relations[3]: snapshot relation, valid-time event relation, valid-time state relation, transaction-time relation, bitemporal event relation, and bitemporal state relation. The first is the kind of relation found in the relational model; the remaining five are temporal relations. As all the schema specification statements of SQL–92 are included in TSQL2, it follows that the data structures of TSQL2 include those in SQL–92.

TSQL2 is also a strict superset of SQL–92 in its query facilities. In particular, if an SQL–92 select statement does not incorporate any of the constructs added in TSQL2 (e.g., the valid clause, the `VALID()` and `TRANSACTION()` expressions, and extensions to the from and group by clauses), and mentions only snapshot relations in its from clause(s), then the language specification states explicitly that the semantics of this statement is identical to its SQL–92 semantics.

It should be noted that the preliminary TSQL2 language specification released in March, 1994 [SAA+94a] did not have that property. In particular, SQL–92 `INTERVAL`s were termed `SPAN`s in the preliminary TSQL2 specification, and TSQL2 `INTERVAL`s were not present at all in SQL–92. The final TSQL2 language specification [SAA+94a] retained SQL–92 `INTERVAL`s and added the `PERIOD` data type, which was previously called `INTERVAL` in preliminary TSQL2 (confusing, isn't it?). Additional changes to the datetime literals were also made to ensure that TSQL2 was a strict superset of SQL–92.

Hence, both conditions are satisfied, demonstrating that TSQL2 is upwards compatible with SQL–92.

As discussed previously, this directly implies that TSQL2 is Codd complete.

Finally note that, while upwards compatibility is a highly desirable property, it says absolutely nothing about constructs added to a data model or query language to support time. This notion of completeness is thus quite limited in scope, as seen from a temporal data-model perspective.

3 Temporal Groupedness

In this section, we first review the previously proposed notions of temporally ungrouped and grouped data models. We then investigate the temporal groupedness of TSQL2. In contrast to upwards compatibility, temporal groupedness speaks directly to the support of time-varying information in the temporal data model.

3.1 Definition

In temporal data modeling, an informal division among temporal relational data models into first normal form (1NF) and non-first normal form (NFNF) models has developed over the years, and each type of model has attracted its followers[4].

With one objective being to clarify this distinction, Clifford et al. [CCT93, CCT94] have recently given a formal definition of two types of relation structures, termed temporally ungrouped and temporally grouped. While it is debatable whether the data model of TSQL2 is strictly a

[3] In this paper, we use the terminology Codd introduced [Cod70]: relation, tuple, and attribute, rather than the more prosaic terminology used in SQL–92 and subsequently in TSQL2: table, row, and column.

[4] In this particular context, the NFNFness is due to *how time is added* to the relational model, so most NFNF temporal data models do not support general NFNF relations, and the distinction is different from the distinction between the 1NF and the various general NFNF relational data models (e.g., [JS82]).

1NF model in the generic sense[5], we will show that the model is temporally ungrouped. To set the stage, we review the definition of a temporally ungrouped data model.

A data model is temporally ungrouped if its data structure component is isomorphic to a particular canonical temporally ungrouped data structure, i.e., an onto and 1–1 mapping must exist between the canonical structure and the structure of the model to be proved temporally ungrouped. The canonical structure is defined next.

Definition 3.1 (canonical temporally ungrouped relation structure) [CCT94, pp. 69–70]
Let $U_D = \{D_1, D_2, \ldots, D_{n_d}\}$ be a set of non-empty value domains, and let $\mathbf{D} = \cup_{i=1}^{n_d} D_i$ be the set of all values. Let $\mathbf{T} = \{t_0, t_1, \ldots, t_i, \ldots\}$ be a non-empty, finite or countably infinite set of times with "$<$" as the total order relation. Finally, let $U_A = \{A_1, A_2, \ldots, A_n\}$ be a set of attributes, and let *TIME* be a distinguished time attribute.

A canonical temporally ungrouped (TU) relation schema is defined as a triple $< \mathbf{A}, \mathbf{K}, \mathbf{DOM} >$ where

(1) $\mathbf{A} \cup \{\mathit{TIME}\}$ ($\mathbf{A} \subseteq U_A$) is the set of attributes of the schema.

(2) The set $\mathbf{K} \cup \{\mathit{TIME}\}$ ($\mathbf{K} \subseteq \mathbf{A}$) is the key of the schema, i.e., $\mathbf{K} \cup \{\mathit{TIME}\} \rightarrow \mathbf{A}$.

(3) $\mathbf{DOM}$ is a function from $\mathbf{A} \cup \{\mathit{TIME}\}$ to $U_D \cup \{\mathbf{T}\}$ that assigns domains in U_D to attributes in $\mathbf{A}$ and *TIME* to $\mathbf{T}$.

A TU database schema is a finite set of TU relation schemas.

A TU tuple $\mathbf{t}$ on schema $< \mathbf{A}, \mathbf{K}, \mathbf{DOM} >$ is a function from $\mathbf{A} \cup \{\mathit{TIME}\}$ to $\mathbf{D} \cup \mathbf{T}$ that assigns a value in $\mathbf{DOM}(A_i)$ to each attribute A_i in $\mathbf{A}$ and a value in *TIME* to $\mathbf{T}$. A TU relation is then a finite set of TU tuples that satisfy the key constraint in **(2)** above. A TU database is a finite set of TU relations. ∎

Example 3.2 The following is a sample TU database with one relation.

A	B	*TIME*
a_1	b_1	1
a_3	b_2	1
a_2	b_1	2
a_3	b_3	2

The relation schema is $< \{A, B\}, \{A, B\}, f >$, where f assigns domains $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ to A and B, respectively, and the natural numbers to *TIME*. ∎

A data model cannot be both temporally ungrouped and temporally grouped, and as we will prove that the TSQL2 data model is isomorphic to TU, we will not give a formal definition of a the canonical temporally grouped relation structure, TG. Rather, we give an example and point to what makes TG grouped.

[5] First normal form (1NF) states that each attribute value is *atomic* [Cod70]. This certainly holds for TSQL2's explicit attributes, which can have as types any of the SQL–92 data types or the new type `PERIOD`. Hence, considering only values of explicit attributes, TSQL2 is a 1NF model. However, the timestamp associated with each tuple in TSQL2 is a *temporal element*, a finite union of periods [JCE+94]. While the timestamp is not an explicit attribute, it can be referenced within a query. We thus feel that timestamps should also satisfy the property. Since the partitioning construct in the from clause of TSQL2 (designated "`(PERIOD)`") effectively iterates over the maximal periods of a temporal element, timestamps are not treated as atomic. Thus, TSQL2 is *not* a 1NF model in this strict sense.

Example 3.3 The schema of a temporally grouped relation consists of the same three components as that of an ungrouped relation, with the exception that the component **DOM** assigns a domain of functions to each attribute in **A**. These functions map times to some value domain. A tuple, then, consists of some specific function for each attribute in **A**. In addition, a tuple has an associated lifespan, a set of times. The functions of a tuple map each time in the tuple's lifespan to some value.

For example, a TG relation schema may have attributes A and B. The key may be the combination of these attributes, and **DOM** may assign functions to A and B that map from the natural numbers to $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$, respectively. A sample tuple may have lifespan $\{1, 2\}$ and may have the mappings $[1 \to a_1, 2 \to a_2]$ as its A-value and $[1 \to b_1, 2 \to b_1]$ as its B-value. A relation instance with this and one more tuple is given next.

A	B	*lifespan*
$1 \to a_1$ $2 \to a_2$	$1 \to b_1$ $2 \to b_1$	$\{1, 2\}$
$1 \to a_3$ $2 \to a_3$	$1 \to b_2$ $2 \to b_3$	$\{1, 2\}$

In comparison with the TU instance given before, this instance adds a grouping of the temporal information. As before, there are four rows. However, these rows may now be combined in several ways to form tuples. Thus, these are also legal TG instance with the same rows.

A	B	*lifespan*
$1 \to a_1$	$1 \to b_1$	$\{1\}$
$2 \to a_2$	$2 \to b_1$	$\{2\}$
$1 \to a_3$ $2 \to a_3$	$1 \to b_2$ $2 \to b_3$	$\{1, 2\}$

A	B	*lifespan*
$1 \to a_1$ $2 \to a_2$	$1 \to b_1$ $2 \to b_1$	$\{1, 2\}$
$1 \to a_3$	$1 \to b_2$	$\{1\}$
$2 \to a_3$	$2 \to b_3$	$\{2\}$

A	B	*lifespan*
$1 \to a_1$	$1 \to b_1$	$\{1\}$
$2 \to a_2$	$2 \to b_1$	$\{2\}$
$1 \to a_3$	$1 \to b_2$	$\{1\}$
$2 \to a_3$	$2 \to b_3$	$\{2\}$

Put informally, the result is that a TG structure contains more elements than a TU structure. This indicates why it is not possible for a relation structure to be both temporally ungrouped and grouped. ∎

3.2 TSQL2 and Temporal Groupedness/Ungroupedness

The canonical ungrouped relation structure TU is a quite simple one. The relation structure of TSQL2 is more elaborate. TSQL2 relations come in several variations. First, relations may support valid time, transaction time, or both. Second, valid-time support may be for either state or event relations. While each of the resulting six types of relations are important in practice, it is advantageous in this context to consider only valid-time state relations. This permits a focus on the important concepts and is consistent with existing work [CCT94][6]. With this restriction, the relation structures of TSQL2 and TU are quite similar.

The central difference is that in TU, tuples are stamped with a single *TIME* value from domain **T** while in TSQL2, tuples are stamped with sets of times, valid-time elements, from domain **T**. As we shall now demonstrate, this difference is not essential.

To show that TSQL2 is temporally ungrouped, we devise an isomorphic mapping between TSQL2 and TU. This mapping takes as argument an arbitrary TSQL2 relation with schema $(A_1, A_2, \ldots, A_n \mid \mathrm{T})$ where the A_i are explicit value attributes and T is the implicit, set-valued

[6] For simplicity, we also assume that the attribute domains of TU and TSQL2 are the same and that all the domains, including the totally ordered time domain, are finite.

time attribute (the vertical bar is used to emphasize that the A_i's are explicit attributes and that T is a distinguished, implicit attribute). It maps each TSQL2 tuple in turn. A TSQL2 tuple

$$(a_1, a_2, \ldots, a_n \mid \{t_{i_1}, t_{i_2}, \ldots, t_{i_k}\})$$

is mapped to the set

$$\{(a_1, a_2, \ldots, a_n \mid t_{i_1}), (a_1, a_2, \ldots, a_n \mid t_{i_2}), \ldots, (a_1, a_2, \ldots, a_n \mid t_{i_k})\}$$

of TU tuples. Note that one TU tuple is generated for each time in the timestamp of the argument TSQL2 tuple. No duplicates are introduced as TSQL2 timestamps are sets of times. Note also that no duplicate tuples are introduced between the sets of tuples generated from individual TSQL2 tuples. This is so because TSQL2 relations do not contain value-equivalent tuples [JS92] (tuples are *value-equivalent* if they agree on all explicit attribute values [JCE+94]).

It should be clear that this mapping is defined for all TSQL2 relations. Next, for any TU relation instance, there exists an TSQL2 instance that maps to it, i.e., the mapping is *onto* the set of all TU relations. To see this, pick an arbitrary TU relation. For each set of value-equivalent tuples, form a single TSQL2 tuple with the same explicit values and with a timestamp that is the union of the timestamps of the value-equivalent tuples. The result is a legal TSQL2 relation, and that relation maps to the initial TU relation. Finally, there is exactly one TSQL2 relation that maps to any TU relation, i.e., the mapping is 1–1. To see this, observe that two different TSQL2 relations map to different TU relations. In conclusion, the mapping is an isomorphism.

It is worth noting that TU and TSQL2 agree regarding duplicates. A TU relation is defined as a *set* of tuples and thus excludes duplicates. TSQL2 relations do not contain value-equivalent tuples, and a timestamp is a *set* of times. A version of TSQL2 changed to allow value-equivalent tuples with overlapping timestamps would contain more instances than the original TSQL2 and would thus not be temporally ungrouped.

It may also be shown that if $\{A_{j_1}, A_{j_2}, \ldots, A_{j_l}\}$ is a temporal key [JSS92] of a TSQL2 relation then $\{A_{j_1}, A_{j_2}, \ldots, A_{j_l}, \mathit{TIME}\}$ is a key of the corresponding TU relation.

We have now seen that TSQL2 is temporally ungrouped and thus not temporally grouped. It is then a natural question to ask whether it would be useful to make TSQL2 grouped. It is our contention that it would not be useful to pursue temporal groupedness in the context of TSQL2. One reason for this is that making TSQL2 relations temporally grouped would change the data structures and thus the query language fundamentally and almost beyond recognition. Another reason is that groupedness does not mix well with the restructuring capabilities of the TSQL2 query language. These capabilities, inspired by, e.g., Gadia [Gad88], have proven very useful [Jen93, SAA+94b]. The following example illustrates in what sense temporal groupedness and restructuring do not mix well.

Example 3.4 In TSQL2 queries, it is possible to declare and use "tuple" variables that range over groups of tuples. To illustrate this, consider the almost identical TSQL2 version, termed `relationAB` (see also below), of the TU instance in Example 3.2. The TSQL2 from clause

```
FROM relationAB ( A ) AS relationA
```

declares a "tuple" variable `relationA` that ranges over groups of `relationAB` tuples with the same `A` value. In the where clause, it is then possible to reference the `A` value (using `relationA.A`) and the timestamp (using `VALID(relationA)`) of this variable. A different TSQL2 from clause

```
FROM relationAB ( B ) AS relationB
```

declares a tuple variable `relationB` that ranges over groups of `relationAB` tuples with the same `B` value. Three groupings are possible, as illustrated by the following relations.

A	B	T
a_1		$\{1\}$
a_2		$\{2\}$
a_3		$\{1,2\}$

`relationA`

A	B	T
	b_1	$\{1,2\}$
	b_2	$\{1\}$
	b_3	$\{2\}$

`relationB`

A	B	T
a_1	b_1	$\{1\}$
a_3	b_2	$\{1\}$
a_2	b_1	$\{2\}$
a_3	b_3	$\{2\}$

`relationAB`

Next, note that these three instances essentially correspond to the last three temporally grouped instances depicted in Example 3.3. However, grouping as in TSQL2 (and, e.g., TempSQL [GB93]) cannot produce the first instance depicted in Example 3.3. This means that temporally groupedness and grouping as in TSQL2 do not coexist harmoniously. ∎

More generally, it is our contention that temporal ungroupedness versus groupedness are fundamental design decisions to be made when designing a data model.

Ungroupedness and groupedness have relative advantages and weaknesses. Clifford et al. [CCT94] lucidly describe the advantages of groupedness. Others have pointed to potential advantages of ungroupedness. To exemplify, assume that attribute A records employee names and attribute B records department names. With grouping, one must decide at schema definition time whether tuples in the relation are intended to represent departments (i.e., one tuple records all employee names for a department which may change its name over time) or employees (i.e., one tuple records all department names for an employee which may change name over time). Without grouping and assuming that departments and employees do not change names over time, this decision may be deferred to when a query is formulated, which is more flexible.

4 Temporal Semi-Completeness

This section first gives refined definitions of *temporal semi-completeness* and *temporal completeness* [BM94]. The definitions presented here add additional syntactical requirements that were intended in the original definitions, but were not stated explicitly.

These notions reflect a belief that both functionality and syntactical requirements are important when evaluating a data model. Both types of requirements are relative to some chosen non-temporal data model. While the definitions are applicable to any pair of a temporal and a non-temporal data model, they are intended to be applied to pairs of a temporal relational data model and the particular version of the snapshot relational model that the temporal model extends.

The section ends with an evaluation of TSQL2 according to each definition. We emphasize that the contribution is not in the definitions per se, but in their application to TSQL2, yielding new insights into this language.

4.1 Definition

To define temporal semi-completeness, we first introduce the auxiliary notion of a snapshot reducible query. We will use r and r^v for denoting a snapshot and a valid-time relation instance, respectively. Similarly, db and db^v are sets of snapshot and valid-time relation instances, respectively.

The definition uses a valid-timeslice operator $\tau_c^{M^v,M}$ (e.g., [Sch77, BM94]) which takes as arguments a valid-time relation r^v (in the data model M^v) and a valid-time instant c and returns a snapshot relation r (in the data model M) containing all tuples valid at time c. In other words, r consists of all tuples of r^v whose valid time includes the time instant c, but without the valid time. We assume that the valid timeslice preserves duplicates, i.e., if r^v contains value-equivalent tuples that are valid at time c then $\tau_c^{M^v,M}(r^v)$ will contain duplicates. This becomes important later, when we consider SQL–92 relations with duplicates.

Definition 4.1 (snapshot reducibility) [Sno87] Let $M = (DS, QL)$ be a snapshot relational data model, and let $M^v = (DS^v, QL^v)$ be a valid-time data model. Also, let db^v be a database instance in DS^v. A valid-time query q^v in QL^v is *snapshot reducible with respect to* a snapshot query q in QL if and only if

$$\forall db^v \; \forall c \; (\tau_c^{M^v,M}(q^v(db^v)) = q(\tau_c^{M^v,M}(db^v))). \qquad \blacksquare$$

Graphically, snapshot reducibility implies that for all db^v and for all c, the commutativity diagram shown in Figure 1 must hold.

$$\begin{tikzcd}
db^v \arrow[rr, "q^v"] \arrow[d, "\text{timeslices at } c"'] & & q^v(db^v) \arrow[d, "\text{timeslice at } c"] \\
\tau_c^{M^v,M}(db^v) \arrow[r, "q"'] & q(\tau_c^{M^v,M}(db^v)) = \tau_c^{M^v,M}(q^v(db^v)) &
\end{tikzcd}$$

Figure 1: Snapshot Reducibility of Query q^v With Respect To Query q

Temporal semi-completeness of a temporal data model with respect to a snapshot data model requires first that all relation instances in the snapshot data model can be produced by taking timeslices of some relation instance in the temporal data model. Further, it is required that each query q in the snapshot model has a counterpart q^v in the temporal model that is snapshot reducible with respect to it. Observe that q^v being snapshot reducible with respect to q poses no syntactical restrictions on q^v. It is thus possible for q^v to be quite different from q, and q^v might be very involved. This is undesirable, as we would like the temporal model to be a straightforward extension of the snapshot model. Consequently, we add to the definition of temporal semi-completeness the syntactical restriction that q^v and q be syntactically similar.

Definition 4.2 (temporal semi-completeness) [BM94] Let $M = (DS, QL)$ be a snapshot data model, and let $M^v = (DS^v, QL^v)$ be a valid-time data model. Data model M^v is *temporally semi-complete with respect to* model M if and only if all three of the following conditions hold.

1. For every relation r in DS, there exists a valid-time relation r^v in DS^v and a time instant c such that $r = \tau_c^{M^v,M}(r^v)$.

2. For every query q in QL, there exists a query q^v in QL^v that is snapshot reducible with respect to q.

3. There exist two (possibly empty) text strings S_1 and S_2 such that for all pairs (q, q^v) of queries, where q^v is snapshot reducible with respect to q, query q^v is syntactically identical to $S_1 q S_2$. $\blacksquare$

Note that the same two strings S_1 and S_2 must apply to all (q, q^v) pairs. The strings represent particular syntactical constructs in the language QL^v.

If the valid-time data model treats valid-time relations as a new type of relation, as does TSQL2, it is possible to use the same syntactical constructs (i.e., q^v and q are identical) for querying snapshot and valid-time relations. In this case, the type of a relation determines the meaning of the syntactical construct.

Temporal semi-completeness of a valid-time data model with respect to a snapshot data model guarantees that the temporal model is a user-friendly (i.e., minor) extension of the snapshot model. Temporal semi-completeness is limited in the sense that it covers only those queries in the temporal data model that are snapshot reducible to a query in the snapshot data model. Most often, a temporal data model allows for the formulation of other queries as well.

4.2 TSQL2 and Temporal Semi-Completeness with Respect to SQL–92

This section identifies where TSQL2 falls short in fulfilling the requirements of temporal semi-completeness. The two related concepts of value-equivalent tuples and duplicates will prove important in this section. The former concept applies only to temporal relations; the latter applies to both valid-time relations and timeslices of valid-time relations. To illustrate the interrelations among these concepts, consider the valid-time relations depicted in Table 1.

r_1

A	T
a_1	$[10-20)$
a_2	$[15-50)$

r_2

A	T
a_1	$[10-17)$
a_1	$[17-20)$
a_2	$[15-50)$

r_3

A	T
a_1	$[10-20)$
a_1	$[15-18)$
a_2	$[15-50)$

r_4

A	T
a_1	$[10-20)$
a_1	$[10-20)$
a_2	$[15-50)$

Table 1: Illustration of Value-equivalent Tuples and Duplicates

Relation r_1 contains no duplicates and no value-equivalent tuples. Thus, no timeslices of r_1 will contain duplicates. Relation r_2 contains no duplicates, but it does contain value-equivalent tuples. However, as the timestamps of the value-equivalent tuples are disjoint, no timeslices will contain duplicates. Relation r_3, like r_2, contains no duplicates, but contains value-equivalent tuples. Unlike in r_2, the timestamps of the value-equivalent tuples and not disjoint and thus there are timeslices of r_3 that contain duplicates. Finally, relation r_4 contains duplicates and thus non-disjoint value-equivalent tuples, leading again to timeslices with duplicates. Note that allowing value-equivalent tuples does not necessarily yield duplicates in timeslices. However, if we want to have duplicates in timeslices, we must allow (non-disjoint) value-equivalent tuples.

4.2.1 Lack of Duplicates in TSQL2

One reason why TSQL2 is not temporally semi-complete with respect to SQL–92 is that SQL–92 relations that contain duplicates have no counterparts in TSQL2 where relations with value-equivalent tuples (and thus duplicates, either in a timeslice, or in the temporal relation itself) are not allowed. Definition 4.2 requires that for every SQL–92 relation r, there must exist a TSQL2 relation r^v and a time instant c such that $\tau_c^{TSQL2,SQL\text{–}92}(r^v) = r$. However, it is not possible to find an r^v in TSQL2 for r's in SQL–92 that contain duplicates. An example illustrates this.

Example 4.3 Let salary relation, *salary_entry*, be given that records (current) monthly incomes of persons. Assume that the person Tom has three incomes because he has three jobs. In two jobs, he makes 1200, and in one he makes 800. This can be represented in SQL–92 as follows.

salary_entry

Name	Amount
Tom	1200
Tom	1200
Tom	800

No timeslice of a TSQL2 relation can yield this relation. The following is a reasonable attempt at adding valid time to the SQL–92 relation to obtain a TSQL2 relation.

salary_entry

Name	Amount	T
Tom	1200	[1994/5 – 1995/3)
Tom	1200	[1994/8 – 1994/12)
Tom	800	[1994/11 – 1995/6)

This relation records that from May 1994 to March 1995, Tom was on one payroll and made a monthly salary of 1200; from August 1994 to December 1994 he was on another payroll where he also made 1200 per month; and from November 1994 to June 1995 he made 800 in a third job. This is not a legal TSQL2 relation because it contains value-equivalent tuples. ∎

The merit of duplicates has already been discussed heatedly (see, e.g., [Dat95, p. 109]). Doubtless SQL–92 would be cleaner in a mathematical sense without duplicates. However, we cannot change SQL–92, so whether we like it or not, it is necessary to deal with duplicates when designing a semi-complete successor to SQL–92. Specifically, for TSQL2 to be snapshot reducible with respect to SQL–92, it must support relations containing value-equivalent tuples with non-disjoint timestamps, permitting duplicates in timeslices.

As a reminder, we note that duplicates may significantly impact the results of queries. For example, the following statement computes a relation that associates with every person that person's total salary.

```
SELECT Name, SUM(Amount)
FROM salary_entry
GROUP BY Name
```

Evaluated over the initial nontemporal *salary_entry* relation, the query computes Tom's salary to be $3200. Without duplicates, the result would have been $2000, which is unintended.

4.2.2 Problems with Subqueries

Temporal semi-completeness requires that for every snapshot query, it is possible to formulate a valid-time query that is snapshot reducible and syntactically similar to it. TSQL2 tries to achieve this goal with a carefully designed default valid clause. This works fine for many simple queries, but it does not work for subqueries.

Ignoring duplicates, the following two SQL–92-statements are equivalent [O'N94, p.117].

```
SELECT r5.a
FROM r5,r6
WHERE r5.a=r6.a
```

```
SELECT r5.a
FROM r5
WHERE EXISTS (SELECT *
              FROM r6
              WHERE r5.a=r6.a)
```

If TSQL2 is to be semi-complete with respect to SQL–92, there must be valid-time queries in TSQL2 that are snapshot reducible with respect to the two queries above and are similar to them. Indeed, the default valid clause of TSQL2 was designed to make those two valid-time queries be identical to the two queries above. The valid-time queries are given below, with the implicit default valid clauses shown.

```
SELECT r5.a                                  SELECT r5.a
VALID INTERSECT(VALID(r5), VALID(r6))        VALID VALID(r5)
FROM r5,r6                                   FROM r5
WHERE r5.a=r6.a                              WHERE EXISTS (SELECT *
                                                           VALID VALID(r6)
                                                           FROM r6
                                                           WHERE r5.a=r6.a)
```

The query to the left behaves as expected. The valid clause states that the valid time of a result tuple is the intersection of the valid times of the argument tuples from r_5 and r_6. This means that the left-hand-side valid-time query is snapshot reducible with respect to the left-hand-side snapshot query. The result ($result_1$) of the query for two sample instances of r_5 and r_6 is shown in Table 2. The situation gets more complicated when we consider the query to the right. The

r_5

A	T
a_1	$[5-9)$

r_6

A	T
a_1	$[7-10)$

$result_1$

A	T
a_1	$[7-9)$

$result_2$

A	T
a_1	$[5-9)$

Table 2: Computing a Valid-time Join Without or With a Subquery in TSQL2 Yields Different Results

outermost valid clause implies that the valid time of a result tuples is equivalent to the valid time of the argument tuple from r_5 (see $result_2$ in Table 2 for an example). This means that the right-hand-side valid-time query is not snapshot reducible with respect to the right-hand-side snapshot query. TSQL2 thus lacks a valid-time query that is snapshot reducible with respect to and is a simple syntactic extension of the right-hand-side snapshot query. Consequently, TSQL2 is not temporally semi-complete with respect to SQL–92.

4.2.3 Summary

We have identified two reasons why TSQL2 is not temporally semi-complete with respect to SQL–92. The first is that, while duplicates are allowed in SQL–92, value-equivalent tuples are not allowed in TSQL2. The second reason is that the valid clause in TSQL2 is not sufficiently powerful to ensure that all SQL–92 queries have similar, snapshot reducible counterparts in TSQL2. We showed this for nested queries. We conjecture that there are also some problems with aggregation, grouping, and ordering.

5 Temporal Completeness

Temporal semi-completeness poses useful restrictions on temporal data models. However, temporal semi-completeness poses restrictions on only a subset of the queries that are generally expressible in temporal data models. For example, it does not cover queries that retrieve information concerning relationships between perceived states of the world at *different* points in time. Furthermore, temporal semi-completeness does not say anything about the format of valid time. Both aspects are accounted for by the notion of a temporally complete data model.

5.1 Definition

Definition 5.1 (temporal completeness) [BM94] A valid-time data model $M^v = (DS^v, QL^v)$ is *temporally complete with respect to* a snapshot data model $M = (DS, QL)$ if and only if all five of the following conditions hold.

1. M^v is temporally semi-complete with respect to M.

2. For every snapshot reducible query q^v in QL^v, it is possible to override snapshot reducibility, either by dropping the syntactic extensions that enforce snapshot reducibility (c.f., Definition 4.2) or by modifying q^v syntactically to $S_1 q S_2$, where S_1 and S_2 are (possibly empty) text strings that depend on QL^v but not on q^v. Overriding snapshot reducibility means to evaluate a query without interpreting valid times.

3. The name of a valid-time relation within a statement can be syntactically substituted (perhaps with other syntactic modifications and additions, such as parentheses) with a query q^v in QL^v that defines the respective valid-time relation without changing the semantics of the statement. The syntactic modifications must depend on QL^v only, not on q^v.

4. Allen's temporal relationships [All83] can be used between (a) temporal attributes of stored valid-time relations (i.e., valid time attributes and explicit temporal attributes), (b) implicitly computed valid times associated with temporally semi-complete (sub)queries, and (c) temporal constants.

5. It is possible to retrieve and constrain (a) maximal continuous valid-time periods and (b) valid times as specified by the user. ∎

First, we require that temporally complete languages are temporally semi-complete. This accounts for queries that can be answered by examining (sequences of) snapshots. Overriding snapshot reducibility accounts for a fundamental principle in databases, namely that a query should treat the elements of a database as uninterpreted objects [CH80, p.158]. Section 5.2.1 provides an example that illustrates this. The third condition ensures that the syntactic construct that is used to enforce snapshot reducibility can be applied not only to whole queries, but also to subqueries. In other words, a temporally complete query may consist of several temporally semi-complete queries. Allen's operators are necessary to state arbitrary temporal relationships. (They were proven to exhaustively describe the relationships between periods [All83]. However, other, equally expressive operators are possible as well.) Note that there are different timestamps that are of interest in a temporal database: temporal attributes of base relations, implicitly computed valid times, and temporal constants. We require that all of them can be used together as operands to Allen's operators. Finally the database system has to support maximal continuous periods and valid times as specified by the user. Both kinds of timestamps have been shown necessary in answering temporal queries [Sri91]. It must be possible to retrieve and constrain (i.e., use as operands of functions and predicates) either kind of timestamp.

We emphasize that the notions of temporal semi-completeness and temporal completeness go beyond approaches that define the completeness in terms of an algebra (i.e., by requiring a temporal language to have the same expressive power as an algebra). For example, temporal semi-completeness (and thus temporal completeness) may, depending on the language it is with respect to, cover aggregates, grouping, null values, ordering, and duplicates.

5.2 TSQL2 and Temporal Completeness with Respect to SQL–92

In order to qualify for temporal completeness, a temporal query language must fulfill the five requirements listed in Definition 5.1. We first must modify TSQL2 to make it temporally semi-complete. To ensure temporal completeness, it must in addition be possible to override snapshot reducibility. The valid clause in TSQL2 is intended for this purpose, but as its scope does not extend to set operations such as `EXCEPT` and `UNION`, the clause cannot override snapshot reducibility for them, either.

The third condition is that a temporal language must allow a valid-time query to appear in a larger query everywhere a valid-time relation name may appear, so that if the valid-time query computes the named relation, the two forms of the larger queries compute the same result. This feature is provided by table expressions, which were introduced in SQL–92 [MS93, p.178] and carried over to TSQL2.

The fourth requirement is satisfied by the where clause which is enhanced with temporal predicates that have the same expressive power as Allen's predicates. Temporal attributes of base relations, implicitly computed valid times (e.g., valid times computed by table expressions), and temporal constants can be used as operands to these predicates.

Finally a temporal language must support maximal continuous valid-time periods and valid times as specified by the user. In the second subsection we will see that TSQL2 ignores the user-specified valid time format.

5.2.1 Overriding Snapshot Reducibility

In TSQL2, the valid clause can be used to override snapshot reducibility—if no valid clause is specified, the semantics defaults to valid-time intersection. However, the scope of the valid clause does not include set operations and, therefore, it is not possible to override valid-time semantics associated with these operations. As an example, suppose the valid-time relations r_5 and r_6 of Table 2. In TSQL2, it is not possible to use `EXCEPT` to retrieve all tuples in r_5 that are not in r_6. Snapshot reducibility is hard-wired into `EXCEPT`, which means that TSQL2 always yields

A	T
a_1	$[5-7)$

rather than the following.

A	T
a_1	$[5-9)$

5.2.2 Beyond Coalescing

The last point of Definition 5.1 requires that a temporal query language be able to retrieve and constrain (a) maximal continuous valid-time periods and (b) valid times as specified by the user. First, TSQL2 falls short in doing this at the outermost level of queries. The results of queries are always coalesced relations, i.e., relations where value-equivalent tuples are eliminated by combining their valid timestamps. This also holds for an individual select statement which may be part of a larger query.

Second, TSQL2 relations are constrained to contain coalesced tuples. To exemplify why this may be a problem, consider relations r_1 and r_2 of Table 1. We may envision that it is significant to a user whether the explicit attribute value a is associated with one single timestamp, $[10-20)$, or is associated with two separate timestamps, $[10-17)$ and $[17-20)$. These two relations may mean different things to a user. However, r_2 is not a legal TSQL2 relation, and if the user inserts tuples $\langle a_1, [10-17)\rangle$ and $\langle a_1, [17-20)\rangle$ into a TSQL2 relation, the tuples will be coalesced, and relation r_1 will be the result. Put differently, TSQL2 does not consider the difference between r_1 and r_2 (and r_3 and r_4) important and thus only admits coalesced relations.

Temporal completeness requires that TSQL2 respects the valid times as provided by the user. If the user provides two intervals for attribute value a, TSQL2 must maintain those two intervals and cannot simply coalesce them. Clearly, this matters for queries. For example, the query "Does there exist an entry with a valid time identical to [10 – 17)" should return "yes" if applied to r_2 (because the user has inserted a tuple with this valid time into r_2) and "no" if applied to r_1 (because the user has not inserted a tuple with this valid time into r_1).

Currently, TSQL2 is a point-based [Cho94], or a snapshot-equivalence preserving [JSS94], temporal query language that uses time intervals at the representational level to achieve a reasonable performance. Changing TSQL2 to respect the valid times as specified by the users represents a substantial conceptual change to TSQL2. It may be argued that admitting uncoalesced relations represents a complication, but it also adds to its expressiveness. With implicit coalescing, users do not have to be concerned with the valid times, but they also cannot associate special semantics with valid times (c.f., Section 4.2.1).

5.2.3 Summary

Apart from not being temporally semi-complete, two aspects prevent TSQL2 from being temporally complete. First, it is not possible to override the temporal semantics of set operations. Second, implicit coalescing prevents TSQL2 from respecting valid times as provided by the users.

6 Ensuring Temporal Completeness in TSQL2

In response to the shortcomings that prevent TSQL2 from being temporally complete, we chose to define a new language, called Applied TSQL2, that will be temporally complete with respect to SQL–92. This section first motivates why we leave TSQL2 unchanged and instead design a new query language. It then reconsiders the problems uncovered in the previous section and proposes appropriate solutions in an informal fashion. A more rigorous mathematical coverage is provided in Section 7.

6.1 An Applied Cousin to TSQL2

In database research, e.g., in database design, relations are generally assumed to be sets of tuples, i.e., without duplicates. While this use of relations as sets is inconsistent with practice, it allows many nice results to be stated and proven much more easily than if duplicates were allowed. In keeping with this tradition, relations in TSQL2 were also defined to be sets of tuples. This has already allowed for relatively straight-forward extension of conventional dependency theory and normal forms to temporal databases [JS92]. Nevertheless, TSQL2 was also envisioned as an extension of SQL–92, and to be relevant to real-world applications. As shown in Section 4, TSQL2 is *not* an extension, for several reasons.

One approach would be to simply modify TSQL2 to address these concerns. As we will see shortly, such modifications jettison other highly useful properties, such as relations being sets. Additionally, supporting duplicates, which is required for temporal semi-completeness with respect to SQL–92, must necessarily abandon temporal ungroupedness.

We propose instead to mirror the relationship between the relational algebra/calculus and SQL–92. TSQL2 should remain as is, so that further work in logical design and query languages can exploit its simple data model, and so that it continues to be temporally ungrouped. A new language should be designed by modifying TSQL2 to reflect the realities of duplicates and the need for temporal completeness with SQL–92. As this new language is preferable to TSQL2 for use by application developers and database administrators, the appellation "Applied TSQL2" seems appropriate. That said, we attempt to ensure that the two languages remain as similar as possible, so that most assertions about TSQL2 also apply to its Applied cousin.

We restrict our discussion to period timestamped valid-time databases. Transaction time, instants (a specialization of periods), and temporal elements (a generalization of periods) are not investigated. Their clean integration into Applied TSQL2 is subject of future research.

6.2 Admitting Value-Equivalent Tuples

We have seen that its lack of value-equivalent tuples prevent TSQL2 from being temporally semi-complete with respect to SQL–92. Therefore, a first step is to allow value-equivalent tuples. This is a significant change, and we will later show how it impacts the use of the language (Section 6.6.1) and its underlying algebra (Section 6.6.2).

The specific change is to timestamp tuples, not with temporal elements, which are finite unions of periods, but rather with multisets of (possibly adjacent, overlapping, or identical) periods. This allows duplicates in timeslices. Also, relations themselves are no longer sets, but are multisets, as in SQL–92.

This change to the data model necessitates changes throughout the language, the full extent of which is beyond the scope of this paper. Coalescing, which occurs automatically in TSQL2, must be accommodated only with explicit directives from the user in Applied TSQL2. Ordering, which is not relevant when relations are sets, becomes an issue in Applied TSQL2. While SQL–92 includes a construct to order a relation in terms of explicit attributes, ordering in terms of time should be considered in Applied TSQL2.

6.3 Achieving Full Snapshot Reducibility

In Section 4.2.2, we saw how the valid clause of TSQL2 was unsuccessful in guaranteeing that all snapshot queries in SQL–92 have similar valid-time counterparts in TSQL2 that are snapshot reducible to them.

It is our contention that too much is expected from the valid clause in TSQL2. It is used for three often related purposes.

1. Defining the default valid times of query result tuples, as well as of tuples that are inserted, modified or deleted.

2. Letting the users control the valid times of result tuples freely (arbitrary temporal expressions can be specified in the valid clause).

3. Overriding snapshot reducibility (see the "superstar" example in [Sno87, p.257]).

As discussed in Section 4.2.2, Item 1 interferes with snapshot reducibility in the presence of subqueries and set operations. Specifically, the default valid clause, which takes the intersection of all of the participating tuple variables, does not ensure snapshot reducibility: there are cases where this default does not work. It is our contention that it is impossible to define a single default valid clause that will work in all situations. Therefore, we propose to abandon the first use of the valid clause while retaining the latter two uses.

Abandoning Item 1 means that we no longer explicitly specify the valid time of result tuples in the default situation. Instead, the valid times of result tuples are computed as defined by snapshot reducibility. The presence of a valid clause in an Applied TSQL2 query then explicitly signals that snapshot reducibility is being overridden, and the absence of a valid clause guarantees by definition snapshot reducibility. In Section 7 we will provide a snapshot reducible semantics for Applied TSQL2 that computes the appropriate tuple timestamps.

6.4 A New Valid Clause

In the previous section, we dropped the use of the valid clause for specifying default timestamps of result tuples. It is still used for explicitly specifying the timestamps of result tuples when snapshot reducibility is not desired. The clause still retains the problem identified in Section 5.2.1, namely that it cannot control the timestamps of tuples that result from set operations because they are outside its scope.

The solution is thus to enlarge the scope of the valid clause to include arbitrary queries. To syntactically reflect this change in semantics, we place the valid clause in front of the select clause, i.e., as the first part of a query. Thus, instead of, e.g.,

```
SELECT r.a
VALID PERIOD(CURRENT_DATE, DATE 'forever')
FROM r
```

we propose to write the following.

```
VALID PERIOD(CURRENT_DATE, DATE 'forever')
  SELECT r.a
  FROM r
```

We emphasize that this syntactical change is intended to reflect that the valid clause now applies to set operations (e.g., `EXCEPT`, `UNION`, etc.) and can thus override snapshot reducibility semantics for these.

In Applied TSQL2, the expression following the `VALID` keyword, takes two different forms.

`VALID SNAPSHOT` <statement>	`VALID` *expression* <statement>

First, note that in both cases <statement> is evaluated with (standard) Codd semantics, i.e., no implicit valid time is computed (c.f., Section 7). The form `VALID SNAPSHOT` returns the result computed by <statement>, i.e., a snapshot relation. The form `VALID` *expression*, where *expression* is a temporal expression possibly containing temporal constants and column names of the relations introduced in <statement>, evaluates *expression* for every result tuple and returns the result of *expression* as the tuple's valid time.

6.5 Supporting User-Specified Valid Times

In Section 5.2.2, we showed that relations in TSQL2 are always coalesced. We saw that this violates temporal completeness and means that TSQL2 does not respect the valid times as entered by the users.

To satisfy temporal completeness, the data model is changed so that relations record the valid timestamps of tuples as they were entered by the users. The changes to relations proposed in Section 6.2 help in achieving this change.

Next, the query language must also provide means of accessing these timestamps. TSQL2 already provides a construct for doing so, the partitioning unit. Specifically, the partitioning unit `PERIOD` may be placed in parentheses after a relation name or a table expression in the from clause. This yields a partitioning of the valid times of tuples into maximal periods (by default, tuples are associated with sets of maximal periods). Not specifying a partitioning unit is equivalent to specifying the default partitioning.

In Applied TSQL2, we eliminate the default partitioning unit. Thus, `Emp` in the clause below denotes `Employee` tuples with the (uncoalesced) timestamps that were provided by the users.

```
FROM Employee AS Emp
```

We also saw that results of TSQL2 queries are always coalesced. This is changed so that there is no automatic/default coalescing at all in queries in Applied TSQL2.

Still, Applied TSQL2 retains the ability to do coalescing, but leaves it to the control of the user. While TSQL2 allowed partitioning in the from clause only, Applied TSQL2 additionally allows a partitioning at the outermost level. This leads to three distinct locations where partitioning ("(PERIOD)") may be specified, as illustrated below.

```
SELECT r.a            SELECT *                   ( SELECT r.a
FROM r(PERIOD)        FROM ( SELECT r.a            FROM r
                              FROM r             )(PERIOD)
                           )(PERIOD)
```

The second and third statements yield the same result, which means that a partitioning at the outermost level is a syntactical shorthand for a table expression. However, because it reduces the syntactical complexity of query expressions and because it makes the language more orthogonal, it is reasonable to include all three possibilities into the language.

6.6 Migrating from TSQL2 to Applied TSQL2

In previous sections we have evaluated TSQL2 against a suite of completeness properties, and found it lacking. Applied TSQL2, a modification of TSQL2, was shown to satisfy temporal semi-completeness and temporal completeness with respect to SQL–92. In this section, we explore further the semantics of Applied TSQL2, and address the impact of these changes on the user and on the algebra.

6.6.1 Effects on the Users

For a user the transition from TSQL2 to Applied TSQL2 is quite subtle because Applied TSQL2 properly subsumes TSQL2. If standard TSQL2 semantics is desired all that is necessary is strictly enforcing a coalesced representation by explicitly specifying a partitioning unit. Apart from that Applied TSQL2 should be more user-friendly to use than TSQL2. First Applied TSQL2 enforces snapshot reducibility by default and, second, it is more orthogonal than TSQL2 because, e.g., snapshot reducibility can always be overridden.

One thing Applied TSQL2 users have to be aware of is that the user-specified format of timestamps, e.g., $[10-20)$ versus $[10-17), [17-20)$, has become relevant if no partitioning unit is specified. We do not expect this to be a problem because usually the user has a good reason for specifying a particular timestamp (see Section 4.2.1) and wants the database system to respect and support it.

6.6.2 Implications for the Underlying Algebra

The algebra for TSQL2 [SJS94] assumes that valid times are stored in an arbitrary format. This means that all valid times have to be coalesced before they are accessed (e.g., before one of Allen's operators is applied or before a temporal set difference is computed). Note that without prior coalescing it is impossible to ensure that answers are independent of the representation of valid time. Also the definitions of some valid-time relational algebra operators (e.g., valid-time set difference) require coalesced input relations.

In Applied TSQL2 it is no longer possible to store timestamps in an arbitrary format and it is also no longer possible to enforce coalescing. Supporting user-specified timestamps requires that all timestamps are stored as specified by the user. Furthermore coalescing is now an operation controlled by the user. Because coalescing usually changes program semantics it can be executed upon a user request only. In particular, this means that the definition of valid-time operators may not assume coalesced input relations.

6.7 Summary

To achieve semi-completeness, we have allowed value-equivalent tuples in Applied TSQL2 relations. This represents a substantial departure from TSQL2 because it involves nontrivial changes to the data model.

We also ensured that queries by default return tuples with timestamps that guarantee snapshot reducibility with respect to SQL–92. This change is a minor departure from TSQL2. It does not change the intended semantics of TSQL2, but repairs TSQL2 by making the syntax properly reflect the intended semantics.

In addition to the two changes above, two more modifications were necessary to make Applied TSQL2 temporally complete. First, we changed the scope of the valid clause so that it can be used for overriding snapshot reducibility of set operations. Syntactically, we reflected this changed semantics by placing the valid clause at the front of a query expression, rather than within, or after, the select statement. Second, by giving the users full control over coalescing, we made Applied TSQL2 respect the timestamps entered by the users. By default, Applied TSQL2 does not coalesce, but the users can still use the partitioning units from TSQL2 for explicit coalescing.

We end by listing the specific changes to TSQL2 that yield Applied TSQL2.

- The data model is changed. Relations in TSQL2 are coalesced sets of tuples timestamped with temporal elements. Relations in Applied TSQL2 are (potentially uncoalesced) multisets of tuples timestamped with multisets of periods.
- By default, the from clause in TSQL2 coalesces value-equivalent tuples. In Applied TSQL2, there is no automatic coalescing. Partitioning on maximal periods is still available using the syntax "`(PERIOD)`".
- The "`VALID` [`SNAPSHOT` | *expression*] <statement>" is new in TSQL2. This statement signals overriding of snapshot semantics for the contained <statement>.
- "`SELECT VALID` *expression* `FROM` ..." in TSQL2 is syntactic sugar for a similar (though more complex) "`VALID` *expression* `SELECT` ..." in Applied TSQL2.
- "`SELECT SNAPSHOT` ..." in TSQL2 is replaced with "`VALID SNAPSHOT SELECT` ..." in Applied TSQL2. "`SELECT` ... `VALID SNAPSHOT`" is syntactic sugar for the latter, effectively moving the keyword `SNAPSHOT` from the `SELECT` portion down to the `VALID` portion, thereby not requiring an enclosing `VALID` statement.
- "`VALID INTERSECT` *expression*" in TSQL2 (as syntactic sugar) is eliminated in Applied TSQL2. This alternative was included in TSQL2 when snapshot reducibility was produced via a default valid clause. As Applied TSQL2 does not use a valid clause to ensure snapshot reducibility, the need for this syntax no longer exists.

As can be seen, the syntactic changes are fairly minimal.

7 Outline of a Formal Semantics

As mentioned earlier, statements without a valid clause are evaluated with temporal semantics, i.e., according to snapshot reducibility (see Figure 1), whereas statements with a leading valid clause are evaluated with nontemporal semantics, i.e., according to Codd semantics. In this section we formalize this imprecise distinction. Note that the semantics described here applies recursively to every table expression. In other words, for every table expression (i.e., in the from clause), we have to specify whether it has to be evaluated with temporal semantics or not. We do not consider here built-in predicates and functions to support (valid) time, as well as other TSQL2 extensions, such as partitioning in the from clause.

For the formal discussion we use some syntactic conventions. First, t_i to denote a tuple, and $t_i \circ t_j$ denotes the concatenation of two tuples. Next, c is a boolean condition on a tuple, and f is a function that maps between tuples. Roughly, condition c represents the where clause, and function f corresponds to the select clause. We assume that temporal operations are implemented as extensions of standard relational database operations. A valid-time tuple is denoted by $\langle t|[S–E)\rangle$ where t represents the explicit attributes and $[S-E)$ is the implicit valid timestamp, consisting of the times in the half-open period with start point S and end point E. As before, r_i^v denotes a valid-time relation, and r_i denotes a snapshot relation.

We use denotational semantics to define Applied TSQL2. Furthermore, we define (temporal) relational algebra operators in terms of tuple relational calculus expressions. We emphasize that, as in SQL–92, duplicates and ordering are preserved, thereby necessitating the use of multisets. Nevertheless, we continue to use standard set notation.

As a first step in defining Applied TSQL2, we define $[\![\text{<statement>}]\!]_{\text{standard}}(r_1, \ldots, r_n)$ to be an algebraic expression utilizing conventional relational algebra operators $\times, \cup, \setminus, \pi, \sigma$. (Here, <statement> is mapped to a relational algebra expression as usual.) The relational algebra expression is evaluated according to the following (standard) definitions.

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

As the next step, we define $[\![\text{<statement>}]\!]_{\text{temporal}}(r_1^v, \ldots, r_n^v)$ to be an algebraic expression which identical to $[\![\text{<statement>}]\!]_{\text{standard}}(r_1, \ldots, r_n)$, except that it uses valid-time relations instead of snapshot relations and uses valid-time relational algebra operators (e.g., $\pi_f^v, \times^v, \cup^v$) instead of snapshot relational algebra operators (e.g., $\pi, \times, \cup$). The valid-time relational algebra operators are defined next.

$$\begin{aligned}
r_1^v \times^v r_2^v &\triangleq \{\langle t_1 \circ t_2|[S-E)\rangle \mid \langle t_1|[A-B)\rangle \in r_1^v \wedge \langle t_2|[C-D)\rangle \in r_2^v \wedge \\
&\qquad S = \max(A, C) \wedge E = \min(B, D) \wedge S < E\} \\
r_1^v \cup^v r_2^v &\triangleq \{\langle t|[S-E)\rangle \mid \langle t|[S-E)\rangle \in r_1^v \vee \langle t|[S-E)\rangle \in r_2^v\} \\
r_1^v \setminus^v r_2^v &\triangleq \{\langle t|[S-E)\rangle \mid \langle t|[A-B)\rangle \in r_1^v \wedge A \leq S \wedge B \geq E \wedge \\
&\qquad \exists C(\langle t|[C-S)\rangle \in r_2^v \vee S = A) \wedge \\
&\qquad \exists D(\langle t|[E-D)\rangle \in r_2^v \vee E = B) \wedge \\
&\qquad \neg\exists U, V(\langle t|[U-V)\rangle \in r_2^v \wedge V > S \wedge U < E)\} \\
\pi_f^v(r) &\triangleq \{\langle t_1|[S-E)\rangle \mid \langle t_2|[S-E)\rangle \in r \wedge t_1 = f(\langle t_2|[A-B)\rangle)\} \\
\sigma_c^v(r) &\triangleq \{\langle t|[S-E)\rangle \mid \langle t|[S-E)\rangle \in r \wedge c(\langle t|[S-E)\rangle)\}
\end{aligned}$$

Among these operators, $\cup^v$, π_f^v, and σ_c^v are straightforward generalizations of their nontemporal counterparts. The valid time of tuples in the output relation is the same as the valid time of respective tuples in the input relation(s). For a valid time Cartesian product $\times^v$, we have to compute the intersection of the input valid times whereas valid-time negation $\backslash^v$ restricts the valid time of tuples in r_1^v to those parts that are not overlapped by the valid time of value-equivalent tuples in r_2^v.

Example 7.1 To exemplify the use of snapshot versus valid-time relational algebra operators, consider the following SQL–92 statement.

```
SELECT r.a
FROM r, s
WHERE r.a=s.a
```

If r and s are snapshot relations, then the semantics of the above statement is $\pi_{r.a}(\sigma_{r.a=s.a}(r \times s))$ (that is, $[\![\text{<statement>}]\!]_{\text{standard}}$), whereas if r and s are valid-time relations, the semantics is $\pi^v_{r.a}(\sigma^v_{r.a=s.a}(r \times^v s))$, that is, $[\![\text{<statement>}]\!]_{\text{temporal}}$. ▮

Given these definitions we need one other operator in order to define the semantics of Applied TSQL2. SN takes a valid-time relation and returns a snapshot relation with an additional (explicit) attribute, holding the valid time of the original valid-time relation.

$$SN(r^v) \triangleq \{\langle t, [S{-}E)\rangle \mid \langle t|[S{-}E)\rangle \in r^v\}$$

Now, the semantics of Applied TSQL2 (ATSQL2) can be defined as follows.

$$[\![\texttt{VALID SNAPSHOT}\ \text{<statement>}]\!]_{\text{ATSQL2}}(r_1^v, \ldots, r_n^v) \triangleq [\![\text{<statement>}]\!]_{\text{standard}}(SN(r_1^v), \ldots, SN(r_n^v))$$

$$[\![\texttt{VALID}\ expr\ \ \text{<statement>}]\!]_{\text{ATSQL2}}(r_1^v, \ldots, r_n^v) \triangleq$$
$$\{\langle t|[S{-}E)\rangle \mid t \in [\![\text{<statement>}]\!]_{\text{standard}}(SN(r_1^v), \ldots, SN(r_n^v)) \wedge [S{-}E) = expr(t)\}$$

$$[\![\text{<statement>}]\!]_{\text{ATSQL2}}(r_1^v, \ldots, r_n^v) \triangleq [\![\text{<statement>}]\!]_{\text{temporal}}(r_1^v, \ldots, r_n^v)$$

This semantics ($[\![\ ]\!]_{\text{ATSQL2}}$) emphasizes that in the presence of the valid statement Codd semantics ($[\![\ ]\!]_{\text{standard}}$) is used, whereas legacy SQL–92 statements utilize temporal semantics ($[\![\ ]\!]_{\text{temporal}}$).

8 Summary and Future Research

This paper has evaluated the consensus temporal query language TSQL2 using existing notions of completeness, some of which were further formalized in the paper.

In consistency with its design goals, TSQL2 was shown to be upwards compatible with SQL–92 and thus to be relationally complete. TSQL2 was also characterized as temporally ungrouped and not temporally grouped. The evaluation of the temporal semi-completeness of TSQL2 with respect to SQL–92 pointed to two important deficiencies: not all SQL–92 relations can be produced taking timeslices of TSQL2 temporal relations, and not all SQL–92 queries have a similar temporal counterpart in TSQL2. Without these deficiencies, TSQL2 would be a "cleaner" extension of SQL–92. The evaluation of temporal completeness of TSQL2 with respect to SQL–92 pointed to two additional problems: with set operations in TSQL2 queries, it is not possible to freely control the valid timestamps of result tuples, and TSQL2 does not respect the valid timestamps of tuples as entered by the users (because value-equivalent tuples are coalesced).

A minimally modified TSQL2, Applied TSQL2, was proposed that rectifies the deficiencies. Among the important differences, relations in Applied TSQL2 are multisets of tuples, and timestamps are multisets of periods. TSQL2 exclusively employs sets. In Applied TSQL2, coalescing is only done when explicitly indicated by the users—there is no automatic coalescing. It is felt that the new facilities for coalescing/partitioning of tuples in Applied TSQL2 are more orthogonal than in TSQL2. The valid clause of Applied TSQL2 differs from that of TSQL2. It does not control the default timestamps of result tuples, and its scope is larger in that it also covers set operations. Also in this respect, we feel that the design of Applied TSQL2 is cleaner than that of TSQL2.

In summary, Applied TSQL2, a temporally semi-complete and complete cousin to TSQL2, employs a more complex and expressive data model than does TSQL2; its query language is similar to that of TSQL2, while including some improvements.

There are several design considerations for Applied TSQL2 that should be investigated in more depth. First, we have only considered the valid-time aspects of temporal data models. It would be appropriate to also take transaction time, by itself and in conjunction with valid time, into consideration. The requires generalization of some of the completeness notions. Next, we have only conducted a preliminary study of the ramifications of moving from sets (of tuples in relations and periods in timestamps) to multisets. A more complete study is warranted. Yet another direction would be to use additional completeness notions in the evaluation of TSQL2. Finally, a comparative study of completeness notions for temporal databases that sheds light on their implications and interrelations, and perhaps leads to new completeness notions, would be worthwhile.

9 Acknowledgments

This work was conducted while the first author visited the University of Arizona. The first author was supported in part by the Swiss NSF. The second author was supported in part by the Danish Natural Science Research Council through grants 11–1089–1, 11–0061–1, and 9400911. The third author was supported in part by NSF grant IRI-9302244.

We thank John Baer, Jan Chomicki, and Charles Kline for interesting discussions and for sharing their knowledge of temporal databases.

References

[AC81] P. Atzeni and P. P. Chen. Completeness of Query Languages for the Entity-Relationship Model. In P. P. Chen, editor, *Proceedings of the Second International Conference on Entity-Relationship Approach*, pages 111–123, October 1981.

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 110–117, January 1979.

[Ban78] F. Bancilhon. On the Completeness of Query Languages for Relational Databases. In *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, Springer Verlag, September 1978.

[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. *The Temporal Deductive Database System ChronoLog*. PhD thesis, Departement Informatik, ETH Zürich, 1994.

[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. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 496–533. Benjamin/Cummings Publishing Company, 1993.

[CCT94] J. Clifford, A. Croker, and A. Tuzhilin. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, 19(1):64–116, March 1994.

[CH80] A. K. Chandra and D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2):156–178, October 1980.

[Cho94] J. Chomicki. Temporal Integrity Constraints in Relational Databases. *Bulletin of the Technical Committee on Data Engineering*, 17(2):33–37, 1994.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod72] E. F. Codd. Relational Completeness of Data Base Sublanguages. *Courant Computer Symposia Series*, 6:65–98, 1972.

[Dat95] C. J. Date. *Relational Database Writings 1991–1994*. Addison-Wesley Publishing Company, 1995.

[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.

[JCE+94] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia. A Glossary of Temporal Database Concepts. *SIGMOD RECORD*, 23(1):52–64, March 1994.

[Jen93] C. S. Jensen. A Consensus Test Suite of Temporal Database Queries. Technical Report R 93-2034, Department of Mathematics and Computer, Institute for Electronic Systems, Fredrik Bajers Vej 7, DK 9220 Aalborg, Denmark, November 1993.

[JS82] G. Jaeschke and H.-J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 124–138, March 1982.

[JS92] C. S. Jensen and R. T. Snodgrass. Proposal of a Data Model for the Temporal Structured Query Language. Technical Report 37, University of Arizona, TempIS Technical Report, July 1992.

[JSS92] C. Jensen, M. Soo, and R. Snodgrass. Extending Normal Forms to Temporal Relations. Technical Report TR 92-17, University of Arizona, July 1992.

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

[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, 1991.

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

[O'N94] P. O'Neil. *Database Principles Programming Performance*. Morgan Kaufmann, San Francisco, 1994.

[ÖS95] G. Özsoyoğlu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4), August 1995.

[SAA+94a] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. TSQL2 Language Specification. *SIGMOD RECORD*, 23(1):65–86, March 1994.

[SAA+94b] R. T. Snodgrass, I. Ahn, G. Ariav, P. Bayer, J. Clifford, C. Dyreson, F. Grandi, L. Hermosilla, C. S. Jensen, W. Käfer, N. Kline, T. Y. C. Leung, N. Lorentzos, Y. Mitsopoulos, J. F. Roddick, M. Soo, and S. M. Sripada. An Evaluation of TSQL2. In *TSQL2 Commentary*. The TSQL2 Language Design Committee, September 1994.

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

[SJS94] M. D. Soo, C. S. Jensen, and R. T. Snodgrass. An Algebra for TSQL2. In *A TSQL2 Commentary*. The TSQL2 Language Design Committee, September 1994.

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

[SQL92] American National Standards Institute. *ANSI X3.135:1992, American National Standard for Information Systems - Database Language SQL*, 1992.

[Sri91] S. M. Sripada. *Temporal Reasoning in Deductive Databases*. PhD thesis, Imperial College of Science and Technology, University of London, 1991.

[TC90] A. Tuzhilin and J. Clifford. A temporal relational algebra as a basis for temporal relational completeness. Technical Report STERN IS-90-18, Information Systems Department, Leonard N. Stern School of Business, New York University, October 1990.