An Algebra for TSQL2*

Michael D. Soo Christian S. Jensen Richard T. Snodgrass

1 Introduction

TSQL2 [SAA+94, SA+94b] is a declarative query language, and as such, requires a procedural (algebraic) equivalent for implementation. In this document, we describe such an algebraic language. We undertook this design in order to show that TSQL2 can be implemented efficiently, with minimal extension of existing techniques.

As TSQL2 provides a consistent extension of SQL-92 [ISO92], we had a parallel goal in the construction of this algebra. Namely, whenever possible, we extend, rather than modify, the snapshot relational algebra to accommodate the TSQL2 data model. This extension is performed to allow the use of established optimization strategies and evaluation algorithms. In addition, we have the somewhat conflicting goal of completeness, i.e., any query expressible in TSQL2 should be implementable as an algebraic expression. We informally demonstrate how TSQL2 language clauses are supported in the algebra.

We first describe an algebra for the conceptual data model underlying TSQL2 [JSS94B]. As TSQL2 supports six types of relations, snapshot, valid-time state, valid-time event, transaction time, bitemporal state, and bitemporal event, we describe six corresponding operator sets. This algebra is minimal in that each defined operator is used by some construct in the language. We also show how language constructs map to the algebra, thereby providing an informal demonstration that the algebra has sufficient expressive power to implement the language.

With this (conceptual) algebra in hand, we continue by describing a *representational* algebra that supports the conceptual semantics, but is better suited for efficient query processing [JMR91]. We assume a 1NF tuple-timestamping data model, thereby allowing the adaptation of well-understood storage organization, query optimization, and query evaluation techniques. The correspondence between the conceptual and representational algebras is described.

The algebra represents a single, but important, portion of a database management system (DBMS) supporting TSQL2. We briefly discuss how such a DBMS can be realized. As with the algebra, we start with a conventional DBMS architecture and minimally extend it with temporal support, thereby using established technologies and algorithms whenever possible.

The remainder of the paper is organized as follows. In Section 2, we define the conceptual algebra using the tuple relational calculus. The description of the algebra is organized around the specific data models supported by the language. A brief discussion of additional operators that can be defined in terms of the core operators in provided in Section 3. In Section 4, we examine the expressive power of the defined algebra, and informally argue that the algebra has sufficient power to express most TSQL2 queries. We also argue that the algebra is minimal in the sense

*The authors have the following addresses and affiliations. M. D. Soo and R. T. Snodgrass are with Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, {soo,rts}@cs.arizona.edu. C. S. Jensen is with Aalborg University, Datalogi, Fredrik Bajers Vej 7E, DK–9220 Aalborg Ø, Denmark, csj@iesd.auc.dk.

that each algebraic operator is necessary to implement some language construct. In Section 5, we discuss implementation. We show how the semantics of the conceptual algebra can be supported in a 1NF data model that is well-suited for query evaluation, and argue that efficient support for the semantics of the conceptual model can be achieved. We then address the larger question of how to construct a DBMS architecture supporting TSQL2. The paper is concluded with a summary and a description of limitations and future improvements of the algebra.

2 Conceptual Algebra

In this section, we describe an extended relational algebra that supports TSQL2. This extended algebra operates on conceptual relations as described in the TSQL2 data model commentary [JSS94B].

As stated above, a goal of this design is completeness. We describe the support for each of the six TSQL2 relation types, snapshot, valid-time state and valid-time event, transaction time, and bitemporal state and bitemporal event, beginning with the snapshot model.

Throughout the paper we use the following conventions. Operators superscripted with $^{\text{VS}}$ or $^{\text{VE}}$ denote valid-time state or valid-time event operators, respectively; similarly, the superscript $^{\text{T}}$ indicates a transaction-time operator, and the superscripts $^{\text{BS}}$ and $^{\text{BE}}$ indicate bitemporal state and bitemporal event operators, respectively.

2.1 Snapshot Support

The snapshot algebra described here serves two purposes. First, it is used to implement queries on snapshot databases, as supported by the base SQL-92 language. Second, it provides a set of base operators that we later extend, and augment, with temporal support.

2.1.1 Formalization

Following Maier [Mai85], a relation schema R is a set of attribute names $\{A_1, A_2, \ldots, A_n\}$. Corresponding to each attribute name A_i, $1 \leq i \leq n$, is a set D_i called the domain of A_i. We define $\mathbf{D} = D_1 \times D_2 \times \ldots \times D_n$. A relation r on schema R, sometimes denoted as $r(R)$, is a finite set of mappings x_1, x_2, $\ldots$, x_k from R to $\mathbf{D}$. Hence $r = \{x_1, x_2, \ldots, x_k\}$. Note that r is a set, hence $\forall i, j, 1 \leq i, j \leq k,\ i \neq j \Rightarrow x_i \neq x_j$.

2.1.2 Snapshot Operators

Figure 1 lists the operators on snapshot relations. In the figure, X represents a set of attribute names in R, $\mathcal{E}(X)$ is a set of expressions involving X, P is a predicate on expressions over the attributes of R, and N is a relation name. The operators are named *projection*, *selection*, *Cartesian product*, *left-outer join*, *union*, *difference*, and *rename*, respectively. Tuple calculus definitions of these operators are found elsewhere [Mai85].

2.2 Valid-Time Support

In this section, we modify the snapshot algebra just defined to support valid time, thereby producing a valid-time algebra. As TSQL2 supports two varieties of valid-time relations, namely, valid-time state and valid-time event relations, we define a set of operators for each. We begin by extending the formalization of snapshot relations to incorporate valid time.

$\pi_{\mathcal{E}(X)} : s \rightarrow s$
$\sigma_P : s \rightarrow s$
$\times : s \times s \rightarrow s$
$=\!\!\!\bowtie\!: s \times s \rightarrow s$
$\cup : s \times s \rightarrow s$
$- : s \times s \rightarrow s$
$\rho_N : s \rightarrow s$

Figure 1: Snapshot Operators

2.2.1 Formalization

A valid-time relation schema $R = \{A_1, A_2, \ldots, A_n \mid \mathrm{V}\}$ is a finite set of explicit attribute names $\{A_1, A_2, \ldots, A_n\}$ and a distinguished timestamp attribute V. Corresponding to each attribute name A_i, $1 \leq i \leq n$, is a set D_i called the domain of A_i. We define $\mathbf{D} = D_1 \times D_2 \times \ldots \times D_n$. We use T_v to represent the set of all valid times, and denote the power set of T_v by $\mathcal{P}(T_v)$. The domain of V is $\mathcal{P}(T_v)$.

We use R^e to represent the explicit attributes of R, i.e., $R^e = \{A_1, A_2, \ldots, A_n\}$. A relation r on schema R, denoted as $r(R)$, is a finite set of mappings x_1, x_2, ..., x_k from R^e to $\mathbf{D}$, where associated with each x_i, $1 \leq i \leq k$, is a non-empty timestamp attribute $t_i \in \mathcal{P}(T_v)$. Hence $r = \{x_1 | \, t_1, x_2 | \, t_2, \ldots, x_k | \, t_k\}$. As in the snapshot model, $\forall i, j, 1 \leq i, j \leq k$, $i \neq j \Rightarrow x_i \neq x_j$, i.e., tuples with identical explicit attribute values, so-called *value-equivalent* tuples, are disallowed.

EXAMPLE: Consider a valid-time relation schema Emp = {Name, Dept | V}. Let the domain of the Name attribute be the set {Al, Bill} and the domain of the Dept attribute be the set {Ship, Load}. Then Emp^e = (Name, Dept), and the domain of Emp^e, $\mathbf{D}$, is the Cartesian product {Al, Bill} $\times$ {Ship, Load}.

For the valid-time attribute, let the domain of valid-times be isomorphic to the natural numbers {0, 1, ...}. Then the domain of V = $\mathcal{P}(\{0, 1, \ldots\})$.

Figure 2 shows a relation Employee(Emp).

Name	Dept	V
Al	Ship	$\{10, \ldots, 15, 20, \ldots, 25\}$
Al	Load	$\{21, \ldots, 24\}$
Bill	Load	$\{35, \ldots, 45\}$

Figure 2: Employee Relation

Each tuple in the relation represents a single fact and when that fact was true in the modeled reality. For example, the first tuple records the fact that Al worked for the Shipping department from times 10 through 15 and again from times 20 to 25. Notice that the timestamp associated with this tuple is truly a set of chronons (representing a union of maximal time periods), rather than a contiguous period of time.

Notice also that the relation obeys the restriction that value-equivalent tuples are disallowed. None of the three tuples share the same values on both their Name and Dept attributes. □

Notationally, we use the subscript $_P$ to represent a predicate on R, and the subscript $_F$ to represent a function on the distinguished timestamp attribute V. All of the attribute-dependent

functions of TSQL2 (and, consequently SQL-92), including those on user-defined time, are permitted within the operator subscripts $_P$ and $_F$. Capital letters A, B, and C (possibly subscripted) represent individual attributes of R^e. Similarly, capital letters X, Y, and Z represent subsets of R^e. Lower case letters x, y, and z (possibly subscripted) represent tuples in r.

2.2.2 Valid-Time State Operators

Figure 3 lists the valid-time state operators. In the figure, vs represents a valid-time state relation, bs represents a bitemporal state relation (defined in Section 2.4.1), and s represents a snapshot relation.

$\pi^{\mathrm{VS}}_{\mathcal{E}(X),F} : vs \rightarrow vs$
$\sigma^{\mathrm{VS}}_{P} : vs \rightarrow vs$
$\bowtie^{\mathrm{VS},n}_{P,F}$: $vs \times \cdots \times vs \rightarrow vs$
$=\!\!\bowtie^{\mathrm{VS}}_{P,F,F'}$: $vs \times vs \rightarrow vs$
$\cup^{\mathrm{VS}} : vs \times vs \rightarrow vs$
$-^{\mathrm{VS}} : vs \times vs \rightarrow vs$
$\rho^{\mathrm{VS}}_{N} : vs \rightarrow vs$
$\mathsf{AT}^{\mathrm{VS,BS}} : vs \rightarrow bs$
$\mathsf{SN}^{\mathrm{VS}} : vs \rightarrow s$
$\mathsf{SL_P}^{\mathrm{VS}} : vs \rightarrow vs$

Figure 3: Valid-Time State Operators

In comparison to the snapshot operators in Figure 1, we note that seven of the ten operators are generalizations of their snapshot counterparts; only three additional operators not having snapshot analogs are introduced. (In the operator set, the valid-time theta-join replaces the snapshot Cartesian product as a base operator.)

Let $X \subseteq R^e$ be a subset of the explicit attributes of relation schema R. The valid-time projection operator has two subscripts: a set of expressions $\mathcal{E}(X)$ to project, and an expression F evaluating to a valid-time element, which produces the timestamp of the result tuple. The expressions $\mathcal{E}(X)$ correspond to the expressions present in the `SELECT` clause of a TSQL2 query. In the following definition, we use $\mathcal{E}(x[X])$ to mean the expressions $\mathcal{E}(X)$ evaluated using the attribute values of tuple x.

$$\begin{aligned}\pi^{\mathrm{VS}}_{\mathcal{E}(X),F}(r) = \{z^{(|\mathcal{E}(X)|+1)} \mid\ & \exists x \in r\ (z[\mathcal{E}(X)] = \mathcal{E}(x[X]) \wedge\ F(x[\mathrm{V}]) \subseteq z[\mathrm{V}]) \wedge \\ & \forall x_1 \in r\ (\mathcal{E}(x_1[X]) = z[\mathcal{E}(X)] \Rightarrow F(x_1[\mathrm{V}]) \subseteq z[\mathrm{V}]) \wedge \\ & \forall t \in z[\mathrm{V}]\ \exists x_2 \in r\ (\mathcal{E}(x_2[X]) = z[\mathcal{E}(X)] \wedge t \in F(x_2[\mathrm{V}])) \wedge \\ & z[\mathrm{V}] \neq \emptyset\}\end{aligned}$$

As the projection may produce value-equivalent tuples on $\mathcal{E}(X)$, the second line collapses each set of value-equivalent tuples into a single result tuple. We term this process *coalescing*. The timestamp of the result tuple is produced by the applying the function F to the timestamps of each of the value-equivalent tuples, and then unioning the results. The last line ensures that no spurious chronons are introduced.

In general, other operators, such as the slice operator $\mathsf{SL_P}^{\mathrm{VS}}$ (defined below), may produce non-coalesced results. Technically, this violates the restriction that value-equivalent tuples are not allowed in the data model. However, the presence of value-equivalent tuples is generally restricted to intermediate query results. The projection operator can always be applied when a coalesced result is needed.

EXAMPLE: Let F compute the intersection of a tuple's valid time with the set $\{14, \ldots, 22\}$. Using the Employee relation of Figure 2, the result of $\pi^{\mathrm{VS}}_{\{Name\},\mathtt{VALID}(Employee)\cap\{14,\ldots,22\}}(Employee)$ is shown below.

Name	V
Al	$\{14, 15, 20, 21, 22\}$

The first two tuples in Figure 2 contribute to the single result tuple. The result is coalesced by the projection. The last tuple in Figure 2 does not produce an output tuple since the resulting timestamp is empty. □

Let P be a predicate on R. Then the selection of P on r, σ^{VS}_P, is defined as follows.

$$\sigma^{\mathrm{VS}}_P(r) = \{z \mid z \in r \ \wedge P(z)\}$$

The valid-time selection operator is identical to its snapshot counterpart.

The valid-time theta-join, $\bowtie^{\mathrm{VS},n}_{P,F}$, is an n-way join of the n input relations r_1, r_2, ..., r_n. A result tuple representing the concatenation of the input tuples x_i, $1 \leq i \leq n$, is produced if the predicate P is satisfied. The timestamp of the result tuple is computed by the function F.

$$\begin{aligned}\bowtie^{\mathrm{VS},n}_{P,F}(r_1, r_2, \ldots, r_n) = \{z^{(m+1)} \mid \exists x_1 \in r_1 \exists x_2 \in r_2 \ \ldots \ \exists x_n \in r_n \\ (P(x_1|t_1, x_2|t_2, \ldots, x_n|t_n) \wedge \\ z[R^e_1] = x_1[R^e_1] \wedge \ldots \wedge z[R^e_n] = x_n[R^e_n] \wedge \\ z[\mathrm{V}] = F(x_1[\mathrm{V}], x_2[\mathrm{V}], \ldots, x_n[\mathrm{V}]) \wedge z[\mathrm{V}] \neq \emptyset)\}\end{aligned}$$

where $m = \Sigma^n_{i=1}|R^e_i|$

The valid-time natural join is defined in terms of this operator by specifying the identity function for P and set intersection for F, and using valid-time projection.

EXAMPLE: To illustrate the valid-time theta-join we introduce a new relation Manages with schema (Dept, MgrName | V). The contents of the Manages relation is shown below.

Dept	MgrName	V
Ship	George	$\{11, \ldots, 22\}$
Load	Dan	$\{24, \ldots, 36\}$

The expression $\bowtie^{\mathrm{VS},2}_{Employee.Dept=Manages.Dept,\mathtt{VALID}(Employee)\cap\mathtt{VALID}(Manages)}(Employee, Manages)$ produces a relation showing employees and their managers, by linking the relations through their common Dept attributes. The result of this expression is as follows. Using intersection for the timestamp computation finds precisely those time periods when an employee worked for a department managed by some manager.

Name	Employee.Dept	Manages.Dept	MgrName	V
Al	Ship	Ship	George	$\{11, \ldots, 15, 20, 21, 22\}$
Al	Load	Load	Dan	$\{24\}$
Bill	Load	Load	Dan	$\{35, 36\}$

□

For the family of outer-join operators, we only discuss the valid-time left outer-join, $r_1 \overset{\text{VS}}{\Rightarrow\!\bowtie}_{P,F,F'} r_2$. The right and full variants can be defined in a similar manner. Two tuples $x_1 \in r_1$ and $x_2 \in r_2$ produce one or two output tuples, if they satisfy the predicate P. If P evaluates to $TRUE$ then a result tuple is generated which is the concatenation of x_1 and x_2, with the result of $F(x_1[\text{V}], x_2[\text{V}])$ as the timestamp value. A second result tuple is also produced. The explicit attribute values of this tuple are set to the attribute values of x_1, however, null values replace the attribute values of x_2. The timestamp of the resulting tuple is set to $F'(x_1[\text{V}], x_2[\text{V}])$. (In most cases, F' will be the difference function on the argument timestamps.)

$$\begin{aligned} r_1 \overset{\text{VS}}{\Rightarrow\!\bowtie}_{P,F,F'} r_2 = \{z^{(|R_1^e|+|R_2^e|+1)} \mid \exists x_1 \in r_1 \exists x_2 \in r_2\ (P(x_1|t_1, x_2|t_2) \wedge \\ ((z[R_1^e] = x_1[R_1^e] \wedge z[R_2^e] = x_2[R_2^e] \wedge z[\text{V}] = F(x_1[\text{V}], x_2[\text{V}])) \vee \\ (z[R_1^e] = x_1[R_1^e] \wedge z[R_2^e] = \perp \wedge z[\text{V}] = F'(x_1[\text{V}], x_2[\text{V}]))) \wedge \\ z[\text{V}] \neq \emptyset)\} \end{aligned}$$

To define the union operator, $\cup^{\text{VS}}$, let both r_1 and r_2 be instances of R.

$$\begin{aligned} r_1 \cup^{\text{VS}} r_2 = \{z^{(n+1)} \mid (\exists x \in r_1\ \exists y \in r_2\ (x[R^e] = y[R^e] \wedge z[R^e] = x[R^e] \wedge z[\text{V}] = x[\text{V}] \cup y[\text{V}])) \vee \\ (\exists x \in r_1\ (z[R^e] = x[R^e] \wedge (\neg \exists y \in r_2 (y[R^e] = x[R^e])) \wedge z[\text{V}] = x[\text{V}])) \vee \\ (\exists y \in r_2\ (z[R^e] = y[R^e] \wedge (\neg \exists x \in r_1 (x[R^e] = y[R^e])) \wedge z[\text{V}] = y[\text{V}]))\} \end{aligned}$$

The first line coalesces value-equivalent tuples in r_1 and r_2. The second line accounts for tuples in r_1 that have no value-equivalent tuples in r_2. The third line handles the symmetric case.

With r_1 and r_2 defined as above, valid-time state difference is defined as follows.

$$\begin{aligned} r_1 -^{\text{VS}} r_2 = \{z^{(n+1)} \mid \exists x \in r_1\ ((z[R^e] = x[R^e]) \wedge \\ ((\exists y \in r_2\ (z[R^e] = y[R^e] \wedge z[\text{V}] = x[\text{V}] - y[\text{V}]) \wedge z[\text{V}] \neq \emptyset) \vee \\ (\neg \exists y \in r_2\ (z[R^e] = y[R^e]) \wedge z[\text{V}] = x[\text{V}])))\} \end{aligned}$$

The last two lines compute the valid-time element, depending on whether a value-equivalent tuple may be found in r_2.

The operator ρ_N^{VS} accepts a valid-time state relation as an argument and returns the same relation renamed to the subscript N. It is used when the same relation is referenced through different correlation names.

The $\mathsf{AT}^{\text{VS,BS}}$ operator transforms a valid-time state relation into a bitemporal state relation. (Bitemporal relation schemas are defined in Section 2.4.1.) Each tuple in the input relation produces exactly one output tuple, whose timestamp is constructed as the cross product of the current transaction time and the valid time of the input tuple.

In the following definitions, the function *bi_chr* computes the set of bitemporal chronons from the set of argument transaction times and the set of argument valid times. The symbol c_t denotes the current transaction time.

$$bi\_chr(T, V) = \{(t, v) \mid t \in T \wedge v \in V\}$$

$$\mathsf{AT}^{\text{VS,BS}}(r) = \{z^{(n+1)} \mid \exists x \in r (z[R^e] = x[R^e] \wedge z[\text{T}] = bi\_chr(\{c_t\}, x[\text{V}])\}$$

The SN^{VS} operator transforms a valid-time state relation into a snapshot relation, by simply removing the timestamp associated with each input tuple.

$$\mathsf{SN}^{\text{VS}}(r) = \{z^{(n)} \mid \exists x \in r (z[R^e] = x[R^e])\}$$

For each tuple $x \in r$, the slice operator $\mathsf{SL}_{\mathsf{P}}^{\mathrm{VS}}$ generates possibly many result tuples, each value-equivalent to x, and timestamped with a maximal period contained in $x[\mathrm{V}]$. (As noted above, this operator violates the restriction against value-equivalent tuples. The projection operator may be subsequently applied to coalesce the result.) In this operator, the subscript P specifies that the operator performs partitioning; it is not a predicate as for the selection and join operators.

Prior to defining the slice operator, we first derive the maximal periods from $x[\mathrm{V}]$. The predicate *isContiguous* determines if the valid-time element v is a set of contiguous chronons contained in the valid-time element V.

$$isContiguous(v, V) = \begin{cases} TRUE & \text{if } \forall t \in V(min(v) \leq t \leq max(v) \Rightarrow t \in v) \\ FALSE & \text{otherwise} \end{cases}$$

The function *maxPeriods* produces the maximal periods in the argument valid-time element V using *isContiguous* to determine the corresponding contiguous valid-time elements contained in V. In the following definition, the functions *min* and *max* return the smallest and largest chronons, respectively, in their argument sets.

$$maxPeriods(V) = \{[min(v), max(v)] \mid isContiguous(v, V) \wedge \neg\exists v'(isContiguous(v', V) \wedge v \subset v')\}$$

with the restriction that $\forall t \in V\ \exists v \in maxPeriods(V)(min(v) \leq t \leq max(v))$

The first conjunct ensures that generated periods correspond to contiguous chronon sets in V. The second conjunct ensures that the periods are maximal. The restriction ensures that no information is lost.

The slice operator simply replicates the explicit attribute values of the tuples in the argument relation and attaches a timestamp from the set of maximal periods.

$$\mathsf{SL}_{\mathsf{P}}^{\mathrm{VS}}(r) = \{z^{(n+1)} \mid \exists x \in r\ \exists v \in maxPeriods(x[\mathrm{V}])(z[R^e] = x[R^e] \wedge z[\mathrm{V}] = v)\}$$

EXAMPLE: Using the Employee relation of Figure 2, the result of $\mathsf{SL}_{\mathsf{P}}^{\mathrm{VS}}(Employee)$ is shown below.

Name	Dept	V
Al	Ship	[10,15]
Al	Ship	[20,25]
Al	Load	[21,24]
Bill	Load	[35,45]

Notice that the tuple (Al, Ship, $\{10, \ldots, 15, 20, \ldots, 25\}$) produces two tuples in the sliced relation corresponding to the two maximal periods [10,15] and [20,25] contained in its timestamp. The remaining tuples each contribute a single tuple to the result since only a single maximal period is contained in their timestamps. □

2.2.3 Valid-Time Event Operators

In the valid-time event model, the timestamp V associated with each tuple is a set of valid-time instants, rather than a union of periods as in the valid-time state model. Hence, instants play the analogous role to periods in the valid-time state model. In particular, the slicing operations on valid-time event relations create a group of value-equivalent tuples each stamped with a single instant from the timestamp of the input tuple.

With this slight distinction, the valid-time state and valid-time event operators are identical. Figure 4 summarizes the valid-time event operators. In the figure, ve represents a valid-time event relation, be represents a bitemporal event relation (defined in Section 2.4.1), and s represents a snapshot relation.

$\pi^{\mathrm{VE}}_{\mathcal{E}(X),F} : ve \rightarrow ve$
$\sigma^{\mathrm{VE}}_{P} : ve \rightarrow ve$
$\bowtie^{\mathrm{VE},n}_{P,F} : ve \times \cdots \times ve \rightarrow ve$
$\Rightarrow\!\!\bowtie^{\mathrm{VE}}_{P,F,F'} : ve \times ve \rightarrow ve$
$\cup^{\mathrm{VE}} : ve \times ve \rightarrow ve$
$-^{\mathrm{VE}} : ve \times ve \rightarrow ve$
$\rho^{\mathrm{VE}}_{N} : ve \rightarrow ve$
$\mathsf{AT}^{\mathrm{VE,BE}} : ve \rightarrow be$
$\mathsf{SN}^{\mathrm{VE}} : ve \rightarrow s$
$\mathsf{SL_P}^{\mathrm{VE}} : ve \rightarrow ve$

Figure 4: Valid-Time Event Operators

Comparing Figure 3 and Figure 4, we note that for each valid-time event operator, there is a corresponding valid-time state operator. As the definitions of the valid-time event operators are nearly identical to those of their valid-time state counterparts (modulo the appropriate superscripts, i.e., $^{\mathrm{VE}}$ rather than $^{\mathrm{VS}}$, and $^{\mathrm{BE}}$ rather than $^{\mathrm{BS}}$), we omit their definitions.

2.3 Transaction-Time Support

The algebra defined in the previous section supports valid-time which models changes in the real-world. We now address the orthogonal issue of supporting transaction-time, which models the update activity of the database. Unlike valid-time, there is no notion of event associated with transaction-time. Hence, we define a single algebra for transaction-time relations.

As before, we begin by extending the snapshot formalization of Section 2.1.1, and continue by discussing the semantics of the operators.

2.3.1 Formalization

A transaction-time relation schema $R = \{A_1, A_2, \ldots, A_n | \mathrm{T}\}$ is a finite set of explicit attribute names $\{A_1, A_2, \ldots, A_n\}$ and a distinguished timestamp attribute T. Corresponding to each attribute name A_i, $1 \leq i \leq n$, is a set D_i called the domain of A_i. We define $\mathbf{D} = D_1 \times D_2 \times \ldots \times D_n$. We use T_t to represent the set of all transaction times, and denote the power set of T_t by $\mathcal{P}(T_t)$. The domain of T is $\mathcal{P}(T_t)$.

We use R^e to represent the explicit attributes of R, i.e., $R^e = \{A_1, A_2, \ldots, A_n\}$. A relation r on schema R, sometimes denoted as $r(R)$, is a finite set of mappings x_1, x_2, ..., x_k from R^e to $\mathbf{D}$, where associated with each x_i, $1 \leq i \leq k$, is a non-empty timestamp attribute $t_i \in \mathcal{P}(T_t)$. As for the valid-time models, no value-equivalent tuples may be present in the relation, i.e., $\forall i, j, 1 \leq i, j \leq k$, $i \neq j \Rightarrow x_i \neq x_j$. Hence $r = \{x_1 | \, t_1, x_2 | \, t_2, \ldots, x_k | \, t_k\}$.

All operators on transaction-time relations are superscripted with $^{\mathrm{T}}$. The subscript $_P$ represents a predicate on R; the subscript $_F$ represents a function on the distinguished timestamp attribute T.

2.3.2 Operators

Figure 5 shows the transaction-time operators. In the figure, t represents a transaction-time relation, and s represents a snapshot relation.

$\pi^{\mathrm{T}}_{\mathcal{E}(X),F} : t \rightarrow t$
$\sigma^{\mathrm{T}}_{P} : t \rightarrow t$
$\bowtie^{\mathrm{T},n}_{P,F}: t \times \cdots \times t \rightarrow t$
$=\!\!\bowtie^{\mathrm{T}}_{P,F,F'}: t \times t \rightarrow t$
$\cup^{\mathrm{T}} : t \times t \rightarrow t$
$-^{\mathrm{T}} : t \times t \rightarrow t$
$\rho^{\mathrm{T}}_{N} : t \rightarrow t$
$\mathsf{SN}^{\mathrm{T}} : t \rightarrow s$
$\mathsf{SL}^{\mathrm{T}} : t \rightarrow t$

Figure 5: Transaction Time Operators

In comparison with the snapshot operators in Figure 1, we note that all transaction-time operators except one are generalizations of some corresponding snapshot operator. The additional operator, SN^{T}, transforms a transaction-time relation to a snapshot relation. Furthermore, we note that the set of transaction time operators is a "subset" of the set of valid-time state or valid-time event operators, i.e., there is an analogous valid-time state operator for each transaction-time operator. As the semantics of the transaction-time operators are nearly identical to the valid-time operators, modulo the timestamp attribute name and the proper superscripts, we omit their definitions.

2.4 Bitemporal Support

Having defined operators for both valid-time and transaction-time relations, we now synthesize these operators into operators that accept bitemporal relations as input.

As before, we begin by formalizing bitemporal relations. We then define operators on bitemporal relations.

2.4.1 Formalization

A bitemporal relation schema $R = \{A_1, A_2, \ldots, A_n | \ \mathrm{T}\}$ is a finite set of explicit attribute names $\{A_1, A_2, \ldots, A_n\}$ and a distinguished timestamp attribute T. Corresponding to each attribute name A_i, $1 \leq i \leq n$, is a set D_i called the domain of A_i. We define $\mathbf{D} = D_1 \times D_2 \times \ldots \times D_n$. Let T_t be the set of all transaction times, and T_v be the set of all valid times. We use $\mathcal{P}(T_t \times T_v)$ to denote the power set of the set of bitemporal chronons. The domain of T is $\mathcal{P}(T_t \times T_v)$.

We use R^e to represent the explicit attributes of R, i.e., $R^e = \{A_1, A_2, \ldots, A_n\}$. A relation r on schema R, sometimes denoted as $r(R)$, is a finite set of mappings $x_1, x_2, \ldots, x_k$ from R^e to $\mathbf{D}$, where associated with each x_i, $1 \leq i \leq k$, is a non-empty timestamp attribute $t_i \in \mathcal{P}(T_t \times T_v)$. We add the explicit restriction that $\forall i, j, 1 \leq i, j \leq k$, $i \neq j \Rightarrow x_i \neq x_j$. Hence $r = \{x_1 | \, t_1, x_2 | \, t_2, \ldots, x_k | \, t_k\}$.

In the following, the subscript $_P$ represents a predicate on R, and the subscript $_F$ represents a function on the valid-time component of the distinguished timestamp attribute T.

2.4.2 Bitemporal Operators

Figure 6 shows the bitemporal state and bitemporal event operators. In the figure, bs represents a bitemporal state relation, be represents a bitemporal event relation, vs represents a valid-time state relation, ve represents a valid-time event relation, and t represents a transaction-time relation.

Bitemporal State	**Bitemporal Event**
$\pi^{\mathrm{BS}}_{\mathcal{E}(X),F} : bs \rightarrow bs$	$\pi^{\mathrm{BE}}_{\mathcal{E}(X),F} : be \rightarrow be$
$\sigma^{\mathrm{BS}}_{P} : bs \rightarrow bs$	$\sigma^{\mathrm{BE}}_{P} : be \rightarrow be$
$\bowtie^{\mathrm{BS},n}_{P,F}$: $bs \times \cdots \times bs \rightarrow bs$	$\bowtie^{\mathrm{BE},n}_{P,F}$: $be \times \cdots \times be \rightarrow be$
$=\!\!\bowtie^{\mathrm{BS}}_{P,F,F'}$: $bs \times bs \rightarrow bs$	$=\!\!\bowtie^{\mathrm{BE}}_{P,F,F'}$: $be \times be \rightarrow be$
$\cup^{\mathrm{BS}} : bs \times bs \rightarrow bs$	$\cup^{\mathrm{BE}} : be \times be \rightarrow be$
$-^{\mathrm{BS}} : bs \times bs \rightarrow bs$	$-^{\mathrm{BE}} : be \times be \rightarrow be$
$\rho^{\mathrm{BS}}_{N} : bs \rightarrow bs$	$\rho^{\mathrm{BE}}_{N} : be \rightarrow be$
$\mathsf{SN}^{\mathrm{BS,VS}} : bs \rightarrow vs$ $\mathsf{SN}^{\mathrm{BS,T}} : bs \rightarrow t$	$\mathsf{SN}^{\mathrm{BE,VE}} : be \rightarrow ve$ $\mathsf{SN}^{\mathrm{BE,T}} : be \rightarrow t$
$\mathsf{SL}^{\mathrm{BS}} : bs \rightarrow bs$ $\mathsf{SL}_{\mathsf{P}}^{\mathrm{BS}} : bs \rightarrow bs$	$\mathsf{SL}^{\mathrm{BE}} : be \rightarrow be$ $\mathsf{SL}_{\mathsf{P}}^{\mathrm{BE}} : be \rightarrow be$

Figure 6: Bitemporal Operators

Comparing the operator set in Figure 6 to the previously defined operator sets shows that all of the bitemporal operators are either direct generalizations of their snapshot counterparts, or generalizations of the few additional operators defined for valid-time or transaction-time relations.

In the following, we define the operators that differ from the valid-time state operators of Section 2.2.2. The definitions of the remaining operators are easily generalized from the valid-time operators. As the definitions of the bitemporal state and bitemporal event operators are identical, modulo the appropriate superscript, we omit the definitions of the bitemporal event operators.

The bitemporal theta-join is an n-way join. A result tuple representing the concatenation of the input tuples x_i, $1 \leq i \leq n$, is produced if the predicate P is satisfied. The valid time of a result tuple is produced using the function F. If the intersection of the transaction time components of the input tuples is non-empty then the timestamp of the result tuple is produced by the cross-product of this intersection with the result of F. Otherwise, the transaction time of the result tuple defaults to the current transaction time.

The function *computeTrans* returns the argument set of transaction times, if it is non-empty. Otherwise, the singleton set containing only the current transaction time is returned.

$$computeTrans(T) = \begin{cases} T & \text{if } T \neq \emptyset \\ \{c_t\} & \text{otherwise} \end{cases}$$

The timestamp of a result tuple is produced by the Cartesian product of the result of *computeTrans* on the intersection of the transaction times of the input tuples, and the result of F.

$$\begin{aligned} \bowtie^{\mathrm{BS},n}_{P,F}(r_1, r_2, \ldots, r_n) = \{ z^{(\Sigma_{i=1}^{n}|R_i^e|)+1} \mid \exists x_1 \in r_1 \exists x_2 \in r_2 \ \ldots \ \exists x_n \in r_n \\ (P(x_1|t_1, x_2|t_2, \ldots, x_n|t_n) \wedge \\ z[R_1^e] = x_1[R_1^e] \wedge z[R_2^e] = x_2[R_2^e] \wedge \ldots \wedge z[R_n^e] = x_n[R_n^e] \wedge \\ z[\mathrm{T}] = bi\_chr(T', V')) \} \end{aligned}$$

where

$$T' = computeTrans(\texttt{TRANSACTION}(x_1[\mathrm{T}]) \cap \texttt{TRANSACTION}(x_2[\mathrm{T}]) \cap \ldots \cap \texttt{TRANSACTION}(x_n[\mathrm{T}]))$$

and
$V' = F(\texttt{VALID}(x_1[\mathrm{T}]), \texttt{VALID}(x_2[\mathrm{T}]), \ldots, \texttt{VALID}(x_n[\mathrm{T}]))$

Using bitemporal state projection and set intersection for F, it is possible to define the bitemporal state natural join in terms of this operator.

For the family of outer-join operators, we only discuss the bitemporal state left outer-join. The right and full variants can be defined in a similar manner. Two tuples $x_1 \in r_1$ and $x_2 \in r_2$ produce one or two output tuples, if they satisfy the predicate P. If P evaluates to *TRUE* then a result tuple is generated which is the concatenation of x_1 and x_2, with the result of $F(\texttt{VALID}(x_1[\mathrm{T}]), \texttt{VALID}(x_2[\mathrm{T}]))$ as its valid time. A second result tuple is also produced if $x_1[\mathrm{T}] - x_2[\mathrm{T}] \neq \emptyset$. The explicit attribute values of this tuple are set to the attribute values of x_1, however, null values replace the attribute values of x_2. The valid time of the resulting tuple is set to $F'(\texttt{VALID}(x_1[\mathrm{T}]), \texttt{VALID}(x_2[\mathrm{T}]))$. Normally, F' is the difference function on timestamps.

$$r_1 \Rightarrow\!\!\bowtie^{\mathrm{BS}}_{P,F,F'} r_2 = \{z^{(|R^e_1|+|R^e_2|+1)} \mid \exists x_1 \in r_1 \exists x_2 \in r_2 (P(x_1|t_1, x_2|t_2) \wedge ((z[R^e_1] = x_1[R^e_1] \wedge z[R^e_2] = x_2[R^e_2] \wedge z[\mathrm{T}] = bi\_chr(T', V')) \vee (z[R^e_1] = x_1[R^e_1] \wedge z[R^e_2] = \perp \wedge z[\mathrm{T}] = bi\_chr(T', V'')) \wedge z[\mathrm{T}] \neq \emptyset)\}$$

where T' and V' are as for the bitemporal theta-join, and $V'' = F'(\texttt{VALID}(x_1[\mathrm{T}]), \texttt{VALID}(x_2[\mathrm{T}]))$.

The $\mathsf{SN}^{\mathrm{BS,VS}}$ operator transforms a bitemporal state relation into a valid-time state relation, by simply removing the transaction times associated with each input tuple.

$$\mathsf{SN}^{\mathrm{BS,VS}}(r) = \{z^{(n+1)} \mid \exists x \in r (z[R^e] = x[R^e] \wedge z[\mathrm{V}] = \{v \mid \exists (t, v) \in x[\mathrm{T}]\})\}$$

The $\mathsf{SN}^{\mathrm{BS,T}}$ operator transforms a bitemporal state relation into a transaction-time relation, by simply removing the valid times associated with each input tuple.

$$\mathsf{SN}^{\mathrm{BS,T}}(r) = \{z^{(n+1)} \mid \exists x \in r (z[R^e] = x[R^e] \wedge z[\mathrm{T}] = \{t \mid \exists (t, v) \in x[\mathrm{T}]\})\}$$

The slice operators $\mathsf{SL}^{\mathrm{BS}}$ and $\mathsf{SL}^{\mathrm{BS}}_{\mathsf{P}}$ generate possibly many result tuples from a single input tuple. Each result tuple is value-equivalent to the input tuple. In the case of $\mathsf{SL}^{\mathrm{BS}}$, the result tuples are timestamped with a maximal transaction-time period and the valid-time element corresponding to that transaction time. For $\mathsf{SL}^{\mathrm{BS}}_{\mathsf{P}}$, each result tuple is timestamped with a maximal non-overlapping rectangle in bitemporal space.
In the following we use the *meets* and *overlaps* operators as defined in the language definition commentary [SA+94b].

$$\mathsf{SL}^{\mathrm{BS}}(r) = \{z^{(n+2)} \mid \exists x \in r (z[R^e] = x[R^e] \wedge bi\_chr(z[\mathrm{T}], z[\mathrm{V}]) \subseteq x[\mathrm{T}])\}$$

with the added restrictions that

$$\forall z_1, z_2 \in \mathsf{SL}^{\mathrm{BS}}(r) (z_1[R^e] = z_2[R^e] \Rightarrow \neg(z_1[\mathrm{T}]\ meets\ z_2[\mathrm{T}] \wedge z_1[\mathrm{V}] = z_2[\mathrm{V}]) \wedge \neg(z_1[\mathrm{T}]\ overlaps\ z_2[\mathrm{T}]))$$
$$\forall x \in r\, \forall t \in x[\mathrm{T}]\, \exists z \in \mathsf{SL}^{\mathrm{BS}}(r) (x[R^e] = z[R^e] \wedge t \in bi\_chr(z[\mathrm{T}], z[\mathrm{V}]))$$

The first restriction ensures that the transaction-time periods are maximal, for a constant valid-time element. The second restriction ensures that no information is lost by the operation.

$$\mathsf{SL}^{\mathrm{BS}}_{\mathsf{P}}(r) = \{z^{(n+2)} \mid \exists x \in r (z[R^e] = x[R^e] \wedge bi\_chr(z[\mathrm{T}], x[\mathrm{V}]) \subseteq x[\mathrm{T}])\}$$

with the added restrictions that

$$\forall z_1, z_2 \in \mathsf{SL}_{\mathsf{P}}^{\mathrm{BS}}(r)(z_1[R^e] = z_2[R^e] \Rightarrow (bi\_chr(z_1[\mathrm{T}], z_2[\mathrm{V}]) \cap bi\_chr(z_2[\mathrm{T}], z_2[\mathrm{V}]) = \emptyset \wedge$$
$$z_1[\mathrm{T}]\ meets\ z_2[\mathrm{T}] \Rightarrow z_1[\mathrm{V}] \neq z_2[\mathrm{V}] \wedge$$
$$z_1[\mathrm{T}]\ overlaps\ z_2[\mathrm{T}] \Rightarrow \neg(z_1[\mathrm{V}]\ meets\ z_2[\mathrm{V}]))$$
$$\forall x \in r\, \forall t \in x[\mathrm{T}]\, \exists z \in \mathsf{SL}_{\mathsf{P}}^{\mathrm{BS}}(r)(x[R^e] = z[R^e] \wedge t \in bi\_chr(z[\mathrm{T}], z[\mathrm{V}]))$$

The first restriction ensures that the resulting bitemporal rectangles do not overlap, that the transaction-time periods are maximal (for a constant valid-time period), and that the valid-time periods are in fact maximal. The second restriction ensures that no information is lost by the operation.

2.5 Summary

We have defined a conceptual algebra for TSQL2 implementation. The algebra consists of six operators sets, one for each of the relation types supported by TSQL2. A salient feature of the algebra is that most of the operators are simple generalizations of existing snapshot operators. Specifically, all but three of the valid-time state and valid-time event operators, one of the transaction time operators, and four of the bitemporal state and event operators are generalizations of corresponding snapshot operators.

A consequence of this design is that implementation of the algebra is able to make use of existing, and well-understood, snapshot query optimization and evaluation techniques. As most operators are snapshot extensions, adaptation of existing techniques is simplified. Moreover, since only a few new operators are added we minimize the additional effort needed to extend query optimization to accommodate these operators, as well as the number of new query evaluation algorithms needed by the query processor.

3 Extended Operators of the Conceptual Algebra

The following are useful valid-time operators that can be expressed in terms of the core operators described in the previous section. Similar definitions are easily constructed for the transaction-time and bitemporal models.

Let r_1 and r_2 be valid-time state or valid-time event relations on schemas R_1 and R_2, respectively, and let $X = R_1^e \cap R_2^e$ be the attributes they have in common.

The valid-time Cartesian product $r_1 \times r_2$ is defined in terms of the 2-way valid-time theta-join as follows.

$$r_1 \times^{\mathrm{V}} r_2 \equiv r_1 \bowtie^{\mathrm{V},2}_{TRUE,\mathtt{VALID}(r_1)\cap\mathtt{VALID}(r_2)} r_2$$

With r_1 and r_2 valid-time relations over schemas R_1 and R_2, respectively, as before, the 2-way valid-time natural join of r_1 and r_2, $r_1 \bowtie^{\mathrm{V},2} r_2$ is defined as follows. (As before, ID denotes the identity function.)

$$r_1 \bowtie^{\mathrm{V},2} r_2 \equiv \pi^{\mathrm{V}}_{R_1^e \cup R_2^e, \mathsf{ID}}(r_1 \bowtie^{\mathrm{V},2}_{r_1.A_1=r_2.A_1 \wedge \ldots \wedge r_1.A_m=r_2.A_m,\mathtt{VALID}(r_1)\cap\mathtt{VALID}(r_2)} r_2)$$

Valid-time intersection is defined using valid-time relational difference.

$$r_1 \cap^{\mathrm{V}} r_2 \equiv r_1 -^{\mathrm{V}} (r_1 -^{\mathrm{V}} r_2)$$

Valid-time slice returns the set of tuples valid during a particular chronon. This operator is defined in terms of a selection on a valid-time relation r. We define two variants of this operator. In the first, result tuples retain the valid-time element associated with the original tuples.

$$\tau_t^{\mathrm{V}}(r) \equiv \pi_R^{\mathrm{V}}(\sigma_t^{\mathrm{V}}{}_{\texttt{OVERLAPS VALID}(r)}(r))$$

In the second variant, the timestamp is removed, thereby producing the snapshot relation valid during a particular chronon.

$$\tau_t^{\mathrm{S}}(r) \equiv \mathsf{SN}(\sigma_t^{\mathrm{V}}{}_{\texttt{OVERLAPS VALID}(r)}(r))$$

4 Expressive Power

The defined algebra is able to implement many of the common TSQL2 queries. However, there remain language features, namely data definition [SJG94] and schema modification [RS94] statements and temporal aggregation constructs [KSL94], that are not yet supported by the algebra. We are currently adding support for temporal aggregates.

For this version of the algebra, we now examine its expressive power with respect to TSQL2. Our purpose is to informally demonstrate that the algebra has the following properties.

1. The operator set is minimal, in the sense that each defined operator is necessary for some linguistic construct.
2. Each TSQL2 construct has some corresponding algebraic operator, thereby demonstrating that the algebra is complete.

We limit the discussion to the bitemporal state algebra, one of the two most complex variants of the algebra. Similar arguments are easily constructed for the remaining operator sets.

4.1 Necessity

In this section, we show that each bitemporal state operation has a corresponding linguistic construct in TSQL2, thereby demonstrating that this operator set is minimal. Throughout this section, we use the bitemporal state relation Employee = (Name, Dept | T) to illustrate the discussion. Additional schemas and relations are introduced as necessary.

4.1.1 Projection Operator

The bitemporal state projection operator is used both to implement projection in the `SELECT` clause, and to implement restructuring, if specified, in the `FROM` clause.

For example, consider the query "What are the names of all employees and when did they work for the company?" This query can be expressed in TSQL2 as follows.

```
SELECT Name
FROM Employee
```

Translation of this query into the bitemporal state algebra is straightforward. The corresponding algebraic expression is shown below. The slice operator, $\mathsf{SL}^{\mathrm{BS}}$, produces value-equivalent tuples from each conceptual tuple, where the bitemporal slices associated with the value-equivalent tuples form a cover of the bitemporal region associated with the original tuple. The selection operator is used to apply the default transaction-time restriction of *now*. The projection operator, π^{BS}, projects the name and valid-time associated with each value-equivalent tuple, and coalesces the resulting relation on the restructuring attribute, Name.

$$\pi^{\mathrm{BS}}_{\{Name\},\mathsf{ID}}(\sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\mathrm{BS}}(Employee)))$$

4.1.2 Selection Operator

The selection operator is used to implement restriction in TSQL2. Several language clauses, including the `GROUP BY`, `HAVING`, and `WHERE` are dependent on the selection operator. We show how the selection operator is used to implement the `WHERE` clause.

Consider the query, "What are the name of employees who were employed by the company sometime during 1992 as known by the database on January 1, 1994?" This query restricts the result to those tuples in the Employee relation that were valid during 1992 and current in the database as of January 1, 1994. This query is expressible in TSQL2 as follows.

```
SELECT Name
FROM Employee
WHERE VALID(Employee) OVERLAPS PERIOD '1992' AND
      TRANSACTION(Employee) OVERLAPS DATE '1994-01-01'
```

The above `WHERE` clause is translated into a selection operator whose predicate is the conjunction of conditions as stated in the `WHERE` clause. In particular, the operands of the predicate can involve TSQL2 defined operations such as `VALID` and `TRANSACTION`. The result relation is then projected on the Name attribute.

$$\pi^{\mathrm{BS}}_{\{Name\},\mathsf{ID}}(\sigma^{\mathrm{BS}}_{\mathtt{VALID}(Employee)\ \mathtt{OVERLAPS\ PERIOD\ '1992'}\ \wedge\ \mathtt{TRANSACTION}(Employee)\ \mathtt{OVERLAPS\ DATE\ '1994-01-01'}}(\mathsf{SL}^{\mathrm{BS}}(Employee)))$$

4.1.3 Join Operator

To illustrate the join operator, we introduce an additional bitemporal state relation, Manages. The Manages relation has schema (Dept, MgrName | T).

Consider the query "Who was John's manager when John worked for the Toy department?" This is expressed in TSQL2 as follows. As before, this query uses the default value, *now*, for transaction-time selection.

```
SELECT MgrName
FROM Employee, Manages
WHERE Employee.Dept = 'Toy' AND Employee.Name = 'John' AND
      Manages.Dept = Employee.Dept
```

This query is implemented in the algebra by joining the Employee and Manages relations. The Manages relation is first restricted to only the tuples current in transaction time. Similarly, the Employee relation is restricted to the current tuples recording when John worked for the Toy department. The result is produced by matching tuples from the restructured relations that overlap in valid time.

$$r_1 \bowtie^{\mathrm{BS}}_{Manages.Dept=Employee.Dept,\mathtt{VALID}(Employee)\cap\mathtt{VALID}(Manages)} r_2$$

where

$$r_1 = \sigma^{\mathrm{BS}}_{\mathtt{TRANSACTION}(Manages)\ \mathtt{OVERLAPS\ NOW}}(\mathsf{SL}^{\mathrm{BS}}(Manages))$$

$$r_2 = \sigma^{\mathrm{BS}}_{Name='John'\wedge Dept='Toy'\wedge\mathtt{TRANSACTION}(Manages)\ \mathtt{OVERLAPS\ NOW}}(\mathsf{SL}^{\mathrm{BS}}(Employee))$$

The previous example illustrated a 2-way join. Within a TSQL2 query, the interaction of the `VALID` clause and the `WHERE` clause may require that the full power of the n-way theta-join be used. Consider the following query, which uses the additional bitemporal state relation Salary with schema (Name, Amount | T). (This query does not compute a useful result—we were unable to derive a meaningful query that required a multi-way join. Our only purpose here is to demonstrate the expressive power of the algebra.)

```
SELECT SNAPSHOT E.Name, M.MgrName, S.Amount
FROM Employee(PERIOD) AS E, Manages(PERIOD) AS M, Salary(PERIOD) AS S
WHERE VALID(E) OVERLAPS VALID(S) AND VALID(S) OVERLAPS VALID(M) AND
       VALID(E) MEETS VALID(M)
```

Notice that it is not possible to compute this query as a series of 2-way joins. Suppose we were to first join the Employee and Salary relations (i.e., attempting to satisfy the predicate `VALID(E) OVERLAPS VALID(S)`). The timestamp of the resulting relation must contain both the timestamp of the Employee tuple (to evaluate the conjunct `VALID(E) MEETS VALID(M)`) and the timestamp of the Salary tuple (to evaluate the conjunct `VALID(S) OVERLAPS VALID(M)`). As the join expression is incapable of returning multiple timestamps, this query cannot be written as a series of 2-way join expressions. The remaining cases (first joining Manages and Employee or first joining Manages and Salary) are similar. We therefore use a 3-way join to compute this query.

$$\mathsf{SN}^{\mathrm{VS}}(\mathsf{SN}^{\mathrm{BS,VS}}(\pi^{\mathrm{BS}}_{\{E.Name,M.MgrName,S.Amount\},\mathsf{ID}} \bowtie^{\mathrm{BS}}_{P,F} (r_1, r_2, r_3)))$$

where

$$P \equiv \texttt{VALID}(E)\ \texttt{OVERLAPS}\ \texttt{VALID}(S) \wedge \texttt{VALID}(S)\ \texttt{OVERLAPS}\ \texttt{VALID}(M) \wedge \texttt{VALID}(E)\ \texttt{MEETS}\ \texttt{VALID}(M)$$
$$F \equiv \texttt{VALID}(E) \cap \texttt{VALID}(M) \cap \texttt{VALID}(S)$$
$$r_1 = \sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\mathrm{BS}}(\rho_E(Employee)))$$
$$r_2 = \sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Manages)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\mathrm{BS}}(\rho_M(Manages)))$$
$$r_3 = \sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Salary)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\mathrm{BS}}(\rho_S(Salary)))$$

4.1.4 Outer-Join Operator

Consider the query "Show all employees and their managers, as well as all employees when they did not have managers." We use this query to illustrate the outer-join operator. The query may be posed in TSQL2 as follows.

```
SELECT E.Name, M.MgrName
FROM Employee AS E, Manages AS M
WHERE E.Dept = M.Dept

       UNION

SELECT E.Name, NULL AS M.MgrName
VALID VALID(E) - VALID(M)
FROM Employee AS E, Manages(Dept) AS M
WHERE E.Dept = M.Dept
```

The first query matches an employee and the managers he or she has had. The valid time of resulting tuples defaults to the intersection of the valid times of the matching tuples. The second query determines the times when an employee did not have a manager.

We use the outer-join operator to compute this query. First the relations are restricted to *now*, the default transaction time. The outer-join, producing the union of the two TSQL2 subqueries, is then computed, and the desired attributes are projected.

$\pi^{\text{BS}}_{\{E.Name, M.MgrName\}, \mathsf{ID}}(r_1 \overset{\text{BS}}{\rtimes}_{E.Dept=M.Dept, \texttt{VALID}(E) \cap \texttt{VALID}(M), \texttt{VALID}(E) - \texttt{VALID}(M)} r_2)$

where

$r_1 = \sigma^{\text{BS}}_{\texttt{TRANSACTION}(E)\ \texttt{OVERLAPS NOW}}(\rho^{\text{BS}}_E(\mathsf{SL}^{\text{BS}}(Employee)))$
$r_2 = \sigma^{\text{BS}}_{\texttt{TRANSACTION}(M)\ \texttt{OVERLAPS NOW}}(\rho^{\text{BS}}_M(\mathsf{SL}^{\text{BS}}(Manages)))$

4.1.5 Union Operator

As an example of the union operator, consider the query "What are the names of the employees and managers who worked, sometime, for the Toy department, as best known now?" Assuming that the Name and MgrName attributes of the Employee and Manages relations, respectively, are union-compatible, this query is expressible in TSQL2 as follows.

```
SELECT Name
FROM Employee
WHERE Dept = 'Toy'

       UNION

SELECT MgrName
FROM Manages
WHERE Dept = 'Toy'
```

This query is expressible in the algebra by first restricting the relations, then projecting the desired attributes, and performing the union.

$r_1 \cup^{\text{BS}} r_2$

where

$r_1 = \pi^{\text{BS}}_{\{MgrName\}, \mathsf{ID}}(\sigma^{\text{BS}}_{Dept='Toy' \wedge \texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\text{BS}}(Manages)))$
$r_2 = \pi^{\text{BS}}_{\{Name\}, \mathsf{ID}}(\sigma^{\text{BS}}_{Dept='Toy' \wedge \texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\text{BS}}(Employee)))$

4.1.6 Difference Operator

As a similar example to the previous one, consider the query "What were the names of the employees at any time in their careers when they were not managers?" A TSQL2 formulation of this query is the following.

```
SELECT Name
FROM Employee

       EXCEPT

SELECT MgrName
FROM Manages
```

This query is expressible in the algebra as follows.

$r_1 -^{\text{BS}} r_2$

where

$r_1 = \pi^{\mathrm{BS}}_{\{MgrName\},\mathsf{ID}}(\sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\mathrm{BS}}(Employee)))$
$r_2 = \pi^{\mathrm{BS}}_{\{Name\},\mathsf{ID}}(\sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Manages)\ \texttt{OVERLAPS NOW}}(\mathsf{SL}^{\mathrm{BS}}(Manages)))$

4.1.7 Rename Operator

The rename operator ρ^{BS}_N allows a relation to be referenced by a different name. It is used in the `FROM` clause where a relation, possibly restructured, may be given a correlation name used by the remainder of the query. An example of the ρ^{BS}_N was given in Section 4.1.4.

4.1.8 At Operator

The $\mathsf{AT}^{\mathrm{VS,BS}}$ operator promotes a valid-time state relation to a bitemporal state relation, by creating bitemporal elements from the valid-time elements associated with the input tuples. As such it is not a bitemporal operator, but we illustrate its use here for completeness.

Suppose we have a valid-time state relation Salary = (Name, Amount | V), and we wanted to compute the bitemporal relation showing every employee's salaries and departments. Such a query could arise, for example, as an intermediate result in a larger computation, e.g.,

```
SELECT Employee.Name
FROM Employee, Salary
WHERE TRANSACTION(Employee) OVERLAPS PERIOD(DATE 'Beginning', DATE 'Forever') AND
       Employee.Name = Salary.Name
```

We could compute this query by promoting Salary to a bitemporal state relation, current as of now, and then joining the input relations.

$$\bowtie^{\mathrm{BS},2}_{Employee.Name=Salary.Name,\texttt{VALID}(Employee)\cap\texttt{VALID}(Salary)} (\mathsf{AT}^{\mathrm{VS,BS}}(Salary), \mathsf{SL}^{\mathrm{BS}}(Employee))$$

4.1.9 Snapshot Operator

The snapshot operator is the opposite of the AT operator. It transforms a bitemporal state relation into a valid-time state relation, by removing the transaction-time component associated with each bitemporal chronon.

Suppose we wanted to list all of company's current employees. This is an example of a important class of queries, as it references the current state of the database.

```
SELECT SNAPSHOT Name
FROM Employee
WHERE VALID(Employee) OVERLAPS DATE 'NOW'
```

We could implement this query by selecting the qualifying tuples, and then using the snapshot operator to remove the timestamp attribute.

$$\pi_{\{Name\}}(\mathsf{SN}^{\mathrm{VS}}(\mathsf{SN}^{\mathrm{BS,VS}}(\sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}\ \wedge\ \texttt{VALID}(Employee)\ \texttt{OVERLAPS NOW}} (\mathsf{SL}^{\mathrm{BS}}(Employee)))))$$

4.1.10 Slice Operators

The slice operators implement slicing and partitioning in the `FROM` clause. Specifically, if the `FROM` clause does not specify a partitioning (i.e., either `PERIOD` for state relations or `INSTANT` for event relations) then one of the slice operators $\mathsf{SL}^{\mathrm{BS}}$ or $\mathsf{SL}^{\mathrm{BE}}$ is used. If a partitioning is specified then one of the slice operators $\mathsf{SL}_{\mathsf{P}}^{\mathrm{BS}}$ or $\mathsf{SL}_{\mathsf{P}}^{\mathrm{BE}}$ is used.

Sections 4.1.1 to 4.1.9 showed examples of the $\mathsf{SL}^{\mathrm{BS}}$ operator. As an example of the $\mathsf{SL}_{\mathsf{P}}^{\mathrm{BS}}$ operator, consider the query "Who has worked continuously for the same department for more than one year?"

```
SELECT Name
FROM Employee(PERIOD)
WHERE INTERVAL(VALID(Employee)) > INTERVAL '1-0' YEAR TO MONTH
```

Specifying `PERIOD` in the `FROM` clause slices the Employee relation on transaction time as well as partitioning it on valid time. Graphically, tuples in the input relations are replicated and timestamped with maximal rectangles covering their respective bitemporal elements. This is implemented by the following algebraic expression.

$$\pi^{\mathrm{BS}}_{\{Name\},\mathsf{ID}}(\sigma^{\mathrm{BS}}_{\texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW} \wedge \texttt{INTERVAL}(\texttt{VALID}(Employee)) > \texttt{INTERVAL '1-0' YEAR TO MONTH}'}(\mathsf{SL}_{\mathsf{P}}^{\mathrm{BS}}(Employee)))$$

4.2 Sufficiency

In this section, we enumerate each of the major linguistic clauses in TSQL2 and show the corresponding algebraic equivalents. This demonstration provides an informal proof that the algebra has sufficient expressive power to implement TSQL2. As before we limit the discussion to the bitemporal state operator set. Similar arguments are easily constructed for the remaining data models.

Throughout, we use the Employee relation defined in the previous section to illustrate the discussion.

4.2.1 `SELECT` Clause

The TSQL2 `SELECT` clause allows arbitrary expressions over the attributes of a relation to be projected as the result of a query. The projection operators are the algebraic analogs of the `SELECT` clause. The bitemporal state projection operator, π^{BS}_{F}, also allows the projection of arbitrary expressions. For example, assuming that the `FROM` clause partitions the Employee relation by period, the following `SELECT` clause returns the names for each employee and their associated bitemporal rectangles.

```
SELECT Name, VALID(Employee), TRANSACTION(Employee)
```

Assuming that any associated `FROM` clause does not rename the Employee relation, this `SELECT` clause is translated into the projection operator as follows.

$$\pi^{\mathrm{BS}}_{Name,\texttt{VALID}(Employee),\texttt{TRANSACTION}(Employee),\mathsf{ID}}$$

4.2.2 `VALID` Clause

The `VALID` clause, and its variant the `VALID INTERSECT` clause, perform valid-time projection, i.e., it specifies the valid-time associated with result tuples generated by a query [HJS94B]. For example, consider the following `VALID` clause.

```
VALID PERIOD(DATE '1972-01-01', DATE '1972-12-31')
```

This clause sets the valid time of all result tuples to the specified period.

The `VALID` and `VALID INTERSECT` clauses are supported by the bitemporal state projection operator, $\pi^{\mathrm{BS}}_{P,F}$ where the subscript $_F$ denotes a function computing the valid-time associated with result tuples. Assuming that the set X of attributes is named in the select list, the `VALID` clause from above would be implemented as follows.

$\pi^{\mathrm{BS}}_{X,\,\texttt{PERIOD(DATE '1972-01-01',DATE '1972-12-31')}}$

Implementation of the `VALID INTERSECT` clause is similar, the only difference being that the resulting timestamp is the temporal intersection of the timestamp attribute and a temporal expression. For example, consider the following `VALID INTERSECT` clause.

```
VALID INTERSECT PERIOD(DATE '1972-01-01', DATE '1972-12-31')
```

Assuming a single relation r is named in the `FROM` clause, this `VALID` clause is implemented by the following projection operator.

$\pi^{\mathrm{BS}}_{X,\,\texttt{VALID}(r)\,\cap\,\texttt{PERIOD(DATE '1972-01-01',DATE '1972-12-31')}}$

4.2.3 `FROM` Clause

The `FROM` clause specifies the relations, the form of those relations, and the names used to refer to relations, in a query [SJ94]. The `FROM` clause may specify that a bitemporal relation either be sliced on transaction time, or sliced both on transaction time and valid time, and then partitioned on valid time. In addition, by using the `AS` modifier a correlation name may be applied by which the relation will be referred to in the remainder of the query.

These two operations, slicing and slicing with partitioning, and correlation name assignment, are implemented using the slicing operators $\mathsf{SL}^{\mathrm{BS}}$ and $\mathsf{SL}^{\mathrm{BS}}_{\mathrm{P}}$, and the rename operator ρ^{BS}, respectively. For example, consider the following `FROM` clause which specifies that the Employee relation is to be sliced on transaction time and referred to by the correlation name E.

```
FROM Employee AS E
```

This clause is supported by first slicing the relation and then renaming it to the correlation name.

$\rho_E(\mathsf{SL}^{\mathrm{BS}}(Employee))$

Similarly, consider the `FROM` clause, `FROM Employee(PERIOD) AS E`, where both slicing and partitioning are specified. In the algebra, first the slice operator with partitioning is used. The relation is then renamed to the specified correlation name.

$\rho_E(\mathsf{SL}^{\mathrm{BS}}_{\mathrm{P}}(Employee))$

4.2.4 WHERE Clause

Syntactically, the WHERE clause [HJS94A] is a series of boolean expressions connected by the boolean operators AND, OR, and NOT. The effect of this clause is to restrict the query result to those tuples that match the given condition.

For queries or subexpressions involving a single relation, the selection operator, σ_P^{BS}, is the algebraic analog of the WHERE clause. As the predicate P in the selection operator is purposefully left unspecified, it is straightforward to see that this operator is powerful enough to implement the WHERE clause.

For example, consider the following WHERE clause restricting tuples from the Employee relation.

```
WHERE Employee.Name = 'John' AND VALID(Employee) OVERLAPS DATE '1992-01-01'
```

This WHERE clause can be implemented using the selection operator as follows.

$$\sigma_{Employee.Name='John' \wedge \texttt{VALID}(Employee)\ \texttt{OVERLAPS DATE '1992-01-01'} \wedge \texttt{TRANSACTION}(Employee)\ \texttt{OVERLAPS NOW}}(Employee)$$

If the FROM clause names more than one input relation, then the WHERE clause is implemented by a combination of selections and joins. As the bitemporal join operator is an n-way join, it may handle an arbitrary number of relations. We showed, in Section 4.1.3, an example of a WHERE clause involving three relations which required a 3-way theta-join.

4.2.5 GROUP BY Clause

The GROUP BY clause creates groups of tuples sharing some attribute value, either explicit or temporal. The GROUP BY clause is simply implemented by generating a series of selection expressions, one per group, so that a tuple belongs to a group if it qualifies according to the corresponding selection expression. For example, consider the following GROUP BY clause.

```
GROUP BY VALID(Employee) USING 1 month LEADING 1 month TRAILING 1 month
```

This clause creates a group for each month where a tuple belongs to a group if it falls within the period from one month prior to one month after a given month. Let the months be month_1, month_2, ..., month_n. Then we can determine if a tuple belongs to a group with the following series of n selection operators, $1 \leq i \leq n$.

$$\sigma^{\mathrm{BS}}_{\texttt{VALID}(r)\ \texttt{OVERLAPS PERIOD(BEGIN(}\mathrm{month}_i\texttt{)-INTERVAL '0:1' YEAR TO MONTH,END(}\mathrm{month}_i\texttt{)+INTERVAL '0:1' YEAR TO MONTH)}}(r)$$

Omitting the LEADING or TRAILING clauses would simplify the generated predicate. Also, grouping on explicit attributes is simpler since LEADING and TRAILING specifications are not allowed for non-time attributes. Grouping on user-defined time can be implemented identically to the above.

4.2.6 HAVING Clause

The HAVING clause eliminates groups produced by the GROUP BY clause from further consideration in the query. As the form of the predicate in the HAVING clause is identical to that of the WHERE clause, the discussion in Section 4.2.4 suffices.

4.3 Summary

In this section, we demonstrated two qualities of the algebra. First, we showed that the algebra has no superfluous operators by demonstrating that each bitemporal state operator has a linguistic counterpart. This quality is important since a small operator set simplifies query optimization and evaluation, making implementation of the algebra practical. Second, we showed, for each bitemporal linguistic construct, a corresponding algebraic expression, thereby demonstrating that the algebra has sufficient expressive power to implement TSQL2. Sufficient expressive power is an obvious requirement of any algebra supporting TSQL2.

Together, these qualities imply that the defined conceptual algebra is powerful enough to handle the rich semantics of TSQL2, without undue additional complexity during query optimization. In the next section, we discuss in more detail how the algebra may be efficiently implemented.

5 Implementation

The conceptual model and algebra are not meant for physical implementation due to the N1NF nature of the model. We therefore show how the semantics of the conceptual algebra can be supported with a 1NF representational model and accompanying algebra.

In the bitemporal state representational model, tuples have associated four distinguished timestamp attributes, T_s, T_e, V_s, and V_e, denoting when the fact was current in the database, as well as when the fact was valid in the real-world, respectively. (The simpler bitemporal event model has three timestamps, while valid-time state, valid-time event and transaction-time models have two, one, and two timestamps, respectively.) Collectively, the four timestamps represent a rectangle in bitemporal space. A single conceptual tuple is represented by possibly many value-equivalent representational tuples which collectively have the same information content. The 1NF nature of this representation allows the use, or adaptation of, many well-established query optimization and evaluation techniques. We use snapshot equivalence [JSS94B] to comparing the semantics of conceptual and representational instances.

We define a polymorphic representational algebra that supports all variants of temporal relations. A single algebra is desirable since a small number of operators simplifies query optimization, the cost modeling associated with query plan generation, as well as the expense of programming query evaluation algorithms. Moreover, our goal is to define a representational algebra that not only supports the full semantics of the conceptual algebra, and hence TSQL2, but also is efficiently implementable with minimal extension of conventional query evaluation techniques. Therefore, in the following, when we discuss implementation of the representational operators, we concentrate on the extensions that need to be made to the well-understood snapshot evaluation algorithms in order to support the temporal representational operators.

For a given conceptual algebra expression, correctness requires that some combination of representational operators returns a snapshot equivalent result, given snapshot equivalent operands. As notation, we distinguish representational operators and instances with a hat, e.g., $\hat{\sigma}$ and $\hat{r}$, and snapshot equivalence is denoted as $\overset{S}{\equiv}$.

The representational operators are shown in Figure 7. As can be seen, only a single operator, the coalescing operator $\hat{\mathsf{C}}$, does not have a conceptual analog. This operator is required due to the presence of value-equivalent tuples in the representation. Also, we note that the polymorphism of the operators are supported in large part by parameterizing the timestamp computation functions, F, associated with the projection, theta-join, and outer-join operators.

In many instances, correctness does not require that that the representational tables be coalesced. For bitemporal tables, value-equivalent tuples may have overlapping rectangles; for valid-

$\hat{\pi}_{\mathcal{E}(X),F} : r \rightarrow r$
$\hat{\sigma}_P : r \rightarrow r$
$\hat{\bowtie}^n_{P,F} : r \times \ldots \times r \rightarrow r$
$\hat{\Rightarrow\!\bowtie}_{P,F,F'} : r \times r \rightarrow r$
$\hat{\cup} : r \times r \rightarrow r$
$\hat{-} : r \times r \rightarrow r$
$\hat{\rho}_N : r \rightarrow r$
$\hat{\mathsf{C}} : r \rightarrow r$
$\hat{\mathsf{SL}} : r \rightarrow r$
$\hat{\mathsf{SL}}_\mathsf{P} : r \rightarrow r$

Figure 7: Representational Algebra

time and transaction-time tables, value-equivalent tuples may have overlapping periods. Some of the operators, however, require that their input(s) be coalesced, and some are more efficient if their input(s) are also clustered on the explicit attributes. We note such circumstances in the following analysis.

In the remainder of this section, we enumerate the representational operators, show how they support the semantics of the conceptual operators, and briefly discuss some evaluation trade-offs.

5.1 Projection Operator

The representational projection operators are nearly identical to their snapshot counterpart. They differ only in the addition of the timestamp computation function F, which supports the TSQL2 `VALID` clause and computes the valid-time associated with a result tuple.

Some of the functions F and the expressions in $\mathcal{E}(X)$ may require that the representation be coalesced. Consider the valid-time state projection operator. It may be the case that an expression in $\mathcal{E}(X)$ or the function F requires the entire timestamp of the conceptual tuple, e.g., for a `INTERVAL` operation, present in either the `SELECT` list or the `WHERE` clause. To support the equivalent semantics in the representation, it is necessary to materialize the conceptual timestamp from the possibly many value-equivalent representational tuples. We use the coalescing operator $\hat{\mathsf{C}}$ shown in Figure 7 to do this. Specifically, let r and $\hat{r}$ both be relations of the appropriate type, i.e., valid-time event, valid-time state, transaction time, bitemporal event, or bitemporal state, and let $r \overset{\mathrm{S}}{\equiv} \hat{r}$. Furthermore let $\pi_{\mathcal{E}(X),F}$ be the matching conceptual projection operator. Then the semantics of the conceptual projection operators are implemented in the appropriate representational models as $\pi_{\mathcal{E}(X),F}(r) \overset{\mathrm{S}}{\equiv} \hat{\pi}_{\mathcal{E}(X),F}(\hat{\mathsf{C}}(\hat{r}))$.

We note that for many common queries prior coalescing of the input relation is not required. For example, for projection operations that do not reference the timestamps of input tuples, i.e., only explicit attributes appear in the select list, F is the identity function. Clearly, in such circumstances the coalescing operation can be omitted.

Furthermore, even if the timestamp of input tuples is referenced, it is often the case that the conceptual timestamp need not be materialized. For example, if the `VALID INTERSECT` clause is used with an period literal, e.g., `VALID INTERSECT PERIOD '1993'`, then the representational tuples can be processed one at a time, without first coalescing.

In addition to the conceptual projection operators, a simple variation of the representational projection operators support the conceptual AT operators, which promote valid-time relations to bitemporal relations. Consider the valid-time state at operator, $\mathsf{AT}^{\mathrm{VS,BS}}$. Here the representational relation is projected on all explicit attributes, and, in addition the timestamp of the resulting tuple

is computed using the current transaction time. Similar remarks apply to the family of conceptual snapshot operators, SN.

5.2 Selection Operator

Implementation of the representational selection operator $\hat{\sigma}_P$ is essentially the same as that of its snapshot counterpart. The distinction is that, depending on the form of the predicate P, the input relation may require prior coalescing. For example if P contains the `INTERVAL` operation, then the timestamps of the conceptual tuple must be materialized from the representation. Again, we use the coalescing operator to accomplish this. Formally, with r and $\hat{r}$ defined as above, we implement the appropriate conceptual selection operator σ_P as $\sigma_P(r) \stackrel{S}{\equiv} \hat{\sigma}_P(\hat{\mathsf{C}}(\hat{r}))$.

We note that for many predicates prior coalescing is not required. For example, if P references only the explicit attributes of R then the coalescing operation can be eliminated.

5.3 Join Operator

As with the previous operators, representational theta-join supports predicates over explicit as well as timestamp attributes, and its semantics are essentially the same as the snapshot theta-join, with the addition of the timestamp computation function F.

The comments for the selection operator also apply to the processing of the predicate in the join operator. Consider the valid-time state theta-join, $\bowtie^{\mathrm{VS,n}}_{P,F}$. The computation of the function F may require the entire valid-time element associated with a conceptual tuple. This valid-time element must be materialized from the multiple timestamps associated with the value-equivalent tuples representing the conceptual tuple. Coalescing of the input relation may be required, prior to join evaluation. However, for many common operations, such as temporal intersection, $\cap$, it is possible to iterate through the timestamps in succession, generating the resulting periods.

Semantically, the temporal outer-join operators are implemented using the representational outer-join and coalescing operators, i.e., $\bowtie^n_{P,F}(r_1, \ldots, r_2) \stackrel{S}{\equiv} \bowtie^n_{P,F}(\hat{\mathsf{C}}(\hat{r_1}), \ldots, \hat{\mathsf{C}}(\hat{r_n}))$.

We note that efficient implementation of temporal joins is challenging for two reasons. First, the predicates associated with temporal join operations are usually inequality predicates, rather than the equality predicates prevalent in snapshot databases. Second, as temporal relations may be many times larger than snapshot relations, efficient evaluation is especially important, as the cost of naively computing a Cartesian product is prohibitive.

Several approaches have been proposed for implementing temporal joins. Several exploit ordering of the input tables to achieve higher efficiency. Most approaches should be applicable to the semantics of the operators defined here and to this representational data model. If the underlying tables are ordered, coalescing can be handled in a manner similar to that for projection. We believe that the multiway joins will rarely be required. As evidence, all 152 queries defined in the TSQL2 evaluation commentary [Sno94] (representing the TSQL2 implementation of the consensus temporal query test suite [Jen93]) can be evaluated using 2-way join algorithms.

5.4 Outer-Join Operator

As with the previous operators, representational outer-join, $=\!\!\bowtie_{P,F,F'}$, supports predicates over explicit as well as timestamp attributes, and its semantics are essentially the same as the snapshot outer-join, with the addition of the timestamp computation functions F and F'. Semantically, the temporal outer-join operators are implemented using the representational outer-join and coalescing operators, i.e., $r_1 =\!\!\bowtie_{P,F,F'} r_2 \stackrel{S}{\equiv} \hat{\mathsf{C}}(\hat{r_1}) =\!\!\bowtie_{P,F,F'} \hat{\mathsf{C}}(\hat{r_2})$.

Many of the same comments on the theta-join implementation apply to the representational outer-join. However, the semantics of the outer-join requires that the input relations be coalesced. To see this, consider the query "Who, if anyone, was Ed's manager for the departments in which he was employed?" We must determine not only the periods during which Ed had a manager, but also those times when Ed did not have a manager. This is most easily accomplished if the input tuples are coalesced into, or at least clustered, on their explicit attribute values. Moreover, the timestamp computation function F may require the entire valid-time element associated with a conceptual tuple.

5.5 Union Operator

The representational union operator is identical to its snapshot counterpart. The result of a union of temporal relations is simply the set union of the input relations. Note that the result is not coalesced, as overlapping value-equivalent tuples may be produced. However, the uncoalesced result is clearly snapshot-equivalent to the corresponding conceptual result. Moreover, the representational coalescing operator may be applied to the the uncoalesced result to produce a coalesced representation.

5.6 Difference Operator

Semantically, the conceptual difference operator is implemented using the representational difference operator, i.e., $r - s \overset{S}{\equiv} \hat{r} \hat{-} \hat{s}$.

Unlike the union operator, simple set difference is insufficient for computing the representational difference operator. This is because two value-equivalent conceptual tuples in the input relations produce a result tuple timestamped with the set difference of the input timestamps. Computation of this difference timestamp is most easily accomplished if the conceptual timestamps are first materialized.

Evaluation of the representational difference operator is simplified if the input relations are clustered on their explicit attribute values, thereby allowing the timestamp for a conceptual tuple to be easily materialized. The difference is then computed by performing a single pass over both input relations, in effect performing a merge-join.

However, if the input relations are not clustered on the explicit attribute values then either a nested-loop computation can be used, or prior coalescing must be performed.

5.7 Coalescing Operator

Coalescing is an important operation, since value-equivalent tuples may be present in the representation. As mentioned in the discussions of other operators, the semantics of some queries demand that the input relations be coalesced prior to evaluation.

If prior coalescing is required, this is most easily accomplished if the input relation is sorted on the explicit attribute values. The temporal element associated with the conceptual tuple is easily reconstructed during a scan of the relation. If indexes or precomputed results are available then it may be possible to avoid the relation scan.

5.8 Slice Operator

The representational slice operators $\hat{\mathsf{SL}}$ and $\hat{\mathsf{SL}}_{\mathrm{P}}$ implement the corresponding conceptual slice operators. Many of the previous remarks apply to the slice operators. For example, consider the $\hat{\mathsf{SL}}^{\mathrm{BS}}$ operator. This operator must reconstruct the valid-time element for a given period of

transaction time from the value-equivalent tuples in the representation. This is a variant of the coalescing problem, and is most easily accomplished if the input tuples are clustered on their explicit attribute values. Transaction-time indexes [Sno92] and precomputed results [JMR91], if available, may be helpful.

Now consider the $\hat{\mathsf{SL}}_{\mathrm{P}}^{\mathrm{BS}}$ operator. Interestingly, due to the representation, it may be less expensive to compute this operator than the $\hat{\mathsf{SL}}^{\mathrm{BS}}$ operator, even though the former has more complicated semantics. If the implementation enforces coalescing of the input to the operator then the $\hat{\mathsf{SL}}_{\mathrm{P}}^{\mathrm{BS}}$ operator is free since the representation is already sliced and partitioned into maximal rectangles. If coalescing is not enforced then remarks similar to those made for implementing the $\hat{\mathsf{SL}}^{\mathrm{BS}}$ operator can be made.

5.9 Rename Operator

The temporal ρ operators have no associated cost as they are intensional operators, and are not dependent on the contents of the database. The temporal operators are identical to their snapshot counterparts.

5.10 Optimization

For efficient query evaluation, we would like to design special-purpose operators for frequently used combinations of operators, or consider combinations of operators during query optimization.

For example, we believe that most TSQL2 queries will reference the current state of the database, in order to support conventional snapshot queries. Therefore, an efficient algorithm to transaction timeslice the database, as given by the expression $\hat{\sigma}_{\texttt{TRANSACTION}(r)\texttt{OVERLAPS NOW}}(\hat{\mathsf{SL}}(r))$, would be very beneficial.

As another example, which we borrow from traditional query optimization, consider the expression $\hat{\pi}_{\mathcal{E}(X)}(\hat{\sigma}_{P,F}(\hat{r} \bowtie \hat{s}))$. A simple evaluation of this query would perform each operation sequentially, i.e., first computing the join, then the selection, and finally the projection. The associated cost is then cost computing $\hat{r} \bowtie \hat{s}$, plus the cost of selecting from $\hat{r} \bowtie \hat{s}$ (selectivities are used to estimate the size of this result), plus the cost of projecting the result of the selection (again selectivities are used). More efficient expressions may be substituted, e.g., if it is possible to push the projection and/or selection into the join, or by implementing a combined operator projection/selection operator, thereby eliminating an intermediate result.

5.11 Supporting *Now*

The addition of *now* [CDS+94] has minimal impact on the representational algebra. We treat *now* as a variable that is *ground*, i.e., given a value, during query evaluation or view materialization [CDI+94]. To accommodate *now* in the representational algebra, we propose adding a new function, "now variable assignment," which assigns a value to *now* everywhere that it appears in a tuple. From that point on, the tuple is ground and manipulated exactly as other ground tuples.

We considered the option of allowing the user to specify that *now* remain uninstantiated during query evaluation, however, this option has one primary disadvantage that is illustrated by the following example. Consider a selection on a valid-time state relation where the selected tuples are those preceding December 1, 1993 in valid time. Should a tuple with an ending time of *now* appear in the result? If today is November 30, 1993, then perhaps the result should include the tuple. However, if the tuple is in the result, then the result becomes invalid the very next day, since on December 1, 1993 the tuple no longer precedes that date! We could associate a "lifetime" with every result, but this is an (apparently) expensive option and requires further research. In

the interest of making minimal changes to SQL-92 we advocate grounding every tuple prior to its use.

5.12 Summary

For many of the temporal operators, the same algorithms used to evaluate their snapshot counterparts may be used directly, or with small modifications, the changes being additional parameters passed to the evaluation functions. These operators include the projection, selection, theta-join, outer-join, and union operators. For other operators, such as the at and snapshot operators, evaluation algorithms are easily constructed as variants of the projection evaluation algorithm.

In some instances, the representational operators demand prior coalescing of input relations, mainly for timestamp computation. Efficient algorithms for temporal coalescing, as well as a thorough study of query optimization strategies for queries involving coalescing, is needed for the construction of a TSQL2 query processor. We note that the need for coalescing is determined by the form of the predicate or timestamp computation function associated with a conceptual operator. In many cases, such as when predicates reference only explicit attributes, coalescing is not required.

However, efficient implementation of several operators, most notably the temporal join operators, is significantly more complex than their snapshot counterparts, in order to avoid performing a Cartesian product of the input. Coalescing is again important here; however, new techniques for temporal join implementation may result in improved performance, and justify additional implementation complexity.

Lastly, depending on the representation enforced by the implementation, efficient techniques for temporal slicing may be required. The database implementor may desire to trade cost on update for evaluation expense by allowing non-coalesced or repetitive information in the database. Efficient temporal slicing techniques should exist to support this capability.

These three problems, temporal coalescing, temporal join evaluation, and temporal slicing, are central to the efficient evaluation of TSQL2 queries, and further research is required to develop and analyze associated algorithms.

6 Architecture

The algebra just described is one component of a DBMS architecture supporting TSQL2. In this section, we describe such an architecture. Our goal is to enumerate the changes that a conventional DBMS would need to support TSQL2. As with the algebra, we are concerned with modifying a conventional DBMS in a minimal fashion to support TSQL2. While a more elaborate architecture is possible (likely with significant performance gains), our purpose is to describe the "first step" towards the realization of a temporal DBMS.

In the next section, we describe a canonical design for a conventional DBMS. We then describe the minimal changes needed by each component of the architecture to support TSQL2. We conclude with a few observations about the described architecture.

6.1 Conventional Architectures

Figure 8 shows a conventional DBMS architecture supporting SQL-92. In the figure, ovals represent data items, e.g., user submitted queries, boxes represent software components, e.g., the query compiler, and arrows show the flow of data through the DBMS.

Queries may be submitted by four types of users, the database administration (DBA) staff, interactive users, application programmers, and parametric users.

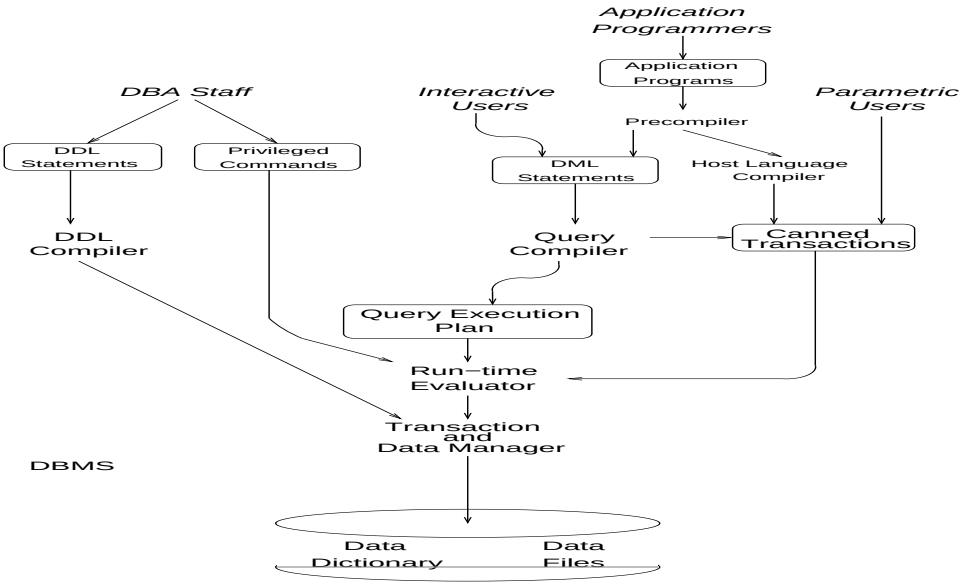

Figure 8: Canonical DBMS Architecture

The DBA staff is responsible for defining and maintaining the database through the execution of data definition language (DDL) statements and privileged commands not available to other users.

Interactive users are sophisticated, database-literate users. They submit SQL-92 queries which are compiled by the query compiler into an procedure-oriented internal representation, the query execution plan. The query execution plan is passed to the run-time evaluator for execution. Actual access to the stored data is performed by the transaction and data manager.

Application programmers submit application programs written in a host programming language, e.g., C, that contain embedded database queries. A precompiler separates the embedded queries from the host application and routes each to an appropriate compiler. The compiled database query and host program are recombined to produce a canned transaction, which may be executed at some later date.

Parametric users are unsophisticated users such as airline reservation agents or customer service representatives. They use the canned transactions produced by applications programmers.

EXAMPLE: Consider snapshot versions of the Employee and Manages schemas defined in Sections 4.1. We illustrate the flow of information through the architecture with the following query, which returns the names of all employees managed by Bob.

```
SELECT Employee.Name, Manages.MgrName
FROM Employee, Manages
WHERE Manages.Dept = Employee.Dept AND Manages.MgrName = 'Bob'
```

Suppose that this query is submitted by an interactive user. (We could assume that the query was submitted via another path, e.g., as an embedded query in an applications program, but the discussion would be more complicated, and no more illustrative.) The query, a DML statement, is first processed by the *query compiler*. The query compiler analyzes the query, first syntactically and then semantically. Syntactic analysis ensures that the lexical and syntactic structure of the query is correct. Semantic analysis performs type checking and verifies that other semantic constraints are

satisfied. Though not shown on the diagram, schema information contained in the data dictionary is used during semantic analysis.

Ultimately, the query compiler produces a procedural expression of the submitted query that is suitable for execution by the run-time evaluator. This procedural expression is based on the relational algebra. As an intermediate step, the query compiler translates the query into a simple algebraic expression which is then optimized. Such an expression for our example might be the following.

$$\pi_{\{Employee.Name, Manages.MgrName\}} (Employee \bowtie_{Employee.Dept=Manages.Dept} (\sigma_{MgrName='Bob'}(Manages)))$$

The final result produced by the query compiler is a query execution plan, which is essentially the optimized algebraic expression with specific algorithms chosen for each algebraic operation. For example, the query compiler might generate the following query execution plan, depending on a estimate of the cost of various algorithms implementing the different operators.

$temp_1 \leftarrow index\_select(Manages, 'Bob')$
$result \leftarrow project_{\{Name, MgrName\}}(nested\_loop\_join(Employee, temp_1, 'Dept'))$

In this query execution plan, the query compiler makes use of an existing index on the MgrName attribute of the Manages relation to quickly find the departments that Bob manages. This intermediate result is stored in the temporary relation $temp_1$. As $temp_1$ is likely to fit in main memory, i.e., Bob only manages a few departments, and hence $temp_1$ will contain only a few tuples, a simple nested-loop join is used to find Bob's employees. In addition, rather than writing another temporary result, the compiler chooses to perform the final projection of the employee name and the manager name attributes "on the fly" with the join.

The query execution plan is sent to the run-time evaluator for execution. For example, for the *index_select* operation in the above query execution plan, the run-time evaluator executes this algorithm, generating a series of index retrievals and subsequent relation page retrievals to materialize the selection. The data retrieval operations are performed by the transaction and data manager which manages the buffer space alloted to the transaction, and ensures the consistency of the database even though multiple transactions may be executing concurrently in the DBMS. □

In the remainder of this section, we describe the minimal changes required to each DBMS component in order to support TSQL2. We note that the precompiler and host language compiler are largely independent of the database query language—they require only small changes to support temporal literal/timestamp conversion. For each of the remaining components, the data dictionary and data files, as well as those within the DBMS proper, we describe the minimal modifications needed by these components to support TSQL2 queries.

6.2 Data Dictionary and Data Files

The data dictionary and data files contain the database, the actual data managed by the DBMS. The data dictionary records schema information such as file structure and format, the number and types of attributes in a relation, integrity constraints, and associated indexes. The data files contain the physical relations and access paths of the database.

For a minimal extension, the data files require no revision. We can store tuple-timestamped temporal relations in conventional relations, where the timestamp attributes are stored as explicit atomic attributes. However, the data dictionary must be extended in a number of ways to

support TSQL2 [SAA+94]. The most significant extensions involve schema versioning, multiple granularities, and vacuuming.

For schema versioning, the data dictionary must record, for each relation, all of its schemas and when they were current. The data files associated with a schema must also be preserved. This is easily accomplished by making a transaction-time relation recording the schemas for a single relation. The transaction time associated with a tuple in this relation indicates the time when the schema was current.

Multiple granularities are associated in a lattice structure specified at system generation time. A simple option is to store the lattice as a data structure in the data dictionary. Alternatively, if the lattice is fixed, i.e., new granularities will not be added after the DBMS is generated, then the lattice can exist as a separate data structure outside of the data dictionary.

Vacuuming specifies what information should be physically deleted from the database. Minimally, this requires a timestamp, the cut-off time, to be stored for each transaction-time or bitemporal relation cataloged by the data dictionary. The cut-off time indicates that all data current in the table before the value of the timestamp has been physically deleted from the relation.

6.3 DDL Compiler

The DDL compiler translates TSQL2 `CREATE`, `ADD`, `REPLACE` and `DROP` statements [SAA+94] into executable transactions. Each of these statements affects both the data dictionary and the data files. The `CREATE` statement adds new definitions, of either relations or indexes, to the data dictionary and creates the data files containing the new relation or index. The `ADD` and `REPLACE` statements change an existing schema by updating the data dictionary, and possibly updating the data file containing the relation. The `ADD` statement is used to add new columns or indexes to a schema, and the `REPLACE` statement is used to change an existing column or index. Lastly, the `DROP` statement is used to remove a table, column, or index definition from a schema, as well as the actual physical data.

Numerous changes are needed by the DDL compiler, but each is straightforward and extend existing functionality in small ways. First, the syntactic analyzer must be extended to accommodate the extended TSQL2 syntax for each of the `CREATE`, `ADD`. `REPLACE`, and `DROP` statements. The semantic analyzer must be extended in a similar manner, e.g., to ensure that an existing relation being transformed into a valid-time state relation via the `ADD VALID STATE` command is not already a valid-time relation.

6.4 Query Compiler

The query compiler translates TSQL2 data manipulation language (DML) statements into an executable, and semantically equivalent, internal form called the query execution plan. As with the DDL compiler, each phase of the query compiler, syntactic analysis, semantic analysis, and query plan generation, must be extended to accommodate TSQL2 queries.

We use the model that the initial phase of the compilation, syntactic analysis, creates a tree-structured query representation which is then referenced and augmented by subsequent phases. Abstractly, the query compiler performs the following steps.

1. Parse the TSQL2 query. The syntactic analyzer, extended to parse the TSQL2 constructs, produces an internal representation of the query, the parse tree.

2. Semantically analyze the constructed parse tree. The parse tree produced by the syntactic analyzer is checked for types and other semantic constraints, and simultaneously augmented with semantic information.

3. Lastly, a query execution plan, essentially an algebraic expression that is semantically equivalent to the original query, is produced from the augmented parse tree by the query plan generator.

The minimal changes required by the query compiler are summarized as follows.

- The syntactic and semantic analyzers must be extended to support TSQL2.
- The query execution plan generator must be extended to support the extended TSQL2 algebra, including the new coalescing, join, and slicing operations. In a minimally extended system, it may be acceptable to use existing algebraic equivalences for optimization, even with the extended operator set. Such an approach preserves the performance of conventional snapshot queries. Later inclusion of optimization rules for the new operators would be beneficial to the performance of temporal queries.
- Support for vacuuming must be included in the compiler. Query modification, which normally occurs after semantic analysis and prior to query optimization, must be extended to include vacuuming support.

The need to extend the syntactic and semantic analyzers is self-evident, and straightforward. (A query compiler has been implemented in conjunction with the MultiCal project that syntactically and semantically analyzes a significant subset of TSQL2.) Extending the query plan generator to use the extended algebra is also straightforward, assuming that temporal aspects of the query are not considered during query optimization. In the worst case, the same performance would be encountered when executing a temporal query on a purely snapshot database. Lastly, in order to support vacuuming, the query compiler, within its semantic analysis phase, must support automated query modification based on vacuuming cut-off times stored in the data dictionary.

6.5 Run-time Evaluator

The run-time evaluator interprets a query plan produced by the query compiler. The run-time evaluator calls the transaction and data manager to retrieve data from the system catalog and data files.

We assume that the run-time evaluator makes no changes to the query plan as received from the query compiler, i.e., the query plan, as generated by the query compiler, is optimized and represents the best possible evaluation plan for the query. As such the changes required for the run-time evaluator are small. Particularly, since evaluation plans for the any new operators have already been selected by the query compiler, the run-time evaluator must merely invoke these operations in the same manner as non-temporal operations. Additionally, as mentioned in Section 5, evaluation algorithms for the new temporal operators (coalescing, n-way joins, and slicing) are similar to well-known algorithms for snapshot operators. For example, coalescing can be implemented with slightly modified duplicate elimination algorithms, which have been well-studied in snapshot databases.

Lastly, changes are needed by the run-time evaluator to support the input and output of temporal literals [Soo93]. Calls to the software components supporting temporal literals must be inserted into the query execution plan by the query compiler and subsequently performed by the run-time evaluator.

6.6 Transaction and Data Manager

The transaction and data manager performs two basic tasks: it manages the transfer of information to and from disk and main memory, and it ensures the consistency of the database in light of concurrent access and transaction failure.

Again, at a minimum little needs to be modified. We assume that the conventional buffer management techniques are employed. Supporting transaction time requires the following small extension to the concurrency control mechanism.

For correctness, transaction times are assigned at commit time, otherwise during an interleaved execution a transaction may see data that is not yet current. This would happen if a transaction reads tuples previously written by a concurrent transaction holding a later transaction time. To avoid this problem, we have an executing transaction write tuples without filling in the transaction timestamp of the tuples. When the transaction later commits, the transaction times of affected tuples are then updated.

This is accomplished by maintaining a (reconstructable) table of tuple IDs written by the transaction. This table is read by an asynchronous background process which performs the physical update of the tuples' transaction timestamp. Correctness only requires that the transaction times for all written tuples be filled in before they are read by a subsequent transaction. While this simple extension suffices, more complex and efficient methods have been proposed [LS93]. Notice also that this algorithm does not affect the recovery mechanism used by the DBMS, assuming that the transaction time of a committed transaction is logged along with the necessary undo/redo information.

6.7 Summary

We have described how a canonical DBMS architecture can be extended to support TSQL2. The changes described are minimal in that they represent the smallest necessary extensions to support the functionality of TSQL2. As the extensions are small, we believe that, as a first-step, TSQL2 can be supported for a relatively low development cost.

We anticipate that the performance of the minimally extended architecture will rival the performance of conventional systems. Snapshot queries on the current database state may suffer a slight performance penalty due to the additional temporal support. However, since we are able to use existing optimization techniques, evaluation algorithms, and storage structures, we expect snapshot queries on the temporal DBMS to approach the performance of identical queries on a conventional DBMS.

Conversely, while there are many opportunities for improvement, we believe that temporal queries on the minimally extended architecture will show reasonable performance. In particular, the architecture can employ new evaluation and optimization techniques for temporal queries currently under investigation. With the addition of temporally optimized storage structures, we expect further performance improvements.

7 Summary and Future Work

We have defined an algebra for TSQL2 implementation. The distinguishing features of this proposal are as follows.

- The algebra supports all six relation types provided by TSQL2.
- The algebra is minimal in the sense that it is an extension of the conventional snapshot algebra, with few additional operators. In addition, we showed that each defined operator

is required by some TSQL2 language construct. Hence, the algebra contains no superfluous operators.

- The algebra is expected to have sufficient expressive power to implement TSQL2 queries, modulo the future work listed below.
- The algebra is minimal in the sense that no superfluous operators were defined.
- The semantics of the algebra can be supported in a 1NF tuple-timestamping representational data model, and hence, efficient implementation of the algebra is possible by exploiting existing query optimization and evaluation techniques.

This algebra is powerful enough to implement most TSQL2 queries. Two language constructs, namely temporal aggregation and the TSQL2 data definition language, are not yet incorporated into the algebra. We expect to augment the algebra with support for these items in the near future.

In addition, we have described an implementation for a TSQL2 database management system. This architecture is notable in that it requires only a few changes to the functionality found in existing conventional DBMS architectures. As such, we believe that TSQL2 can be implemented at relatively low-cost, with the described algebra and architecture providing the implementation framework.

8 Acknowledgements

Curtis Dyreson and Nick Kline gave helpful comments on earlier versions of this document. Additionally, Curtis provided the techniques for supporting *now* in the representational algebra.

This work was supported in part by NSF grants ISI-8902707 and ISI-9302244, IBM contract #1124 and the AT&T Foundation. In addition, support was provided by the Danish Natural Science Research Council through grants 11–9675–1 SE, 11–1089–1 SE, 11–9675–1 SE, and 11–0061.

References

[CDI+94] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "Now" in Temporal Databases. Technical Report R 94–2047 Department of Mathematics and Computer Science, Aalborg University, Denmark, November 1994.

[CDS+94] J. Clifford, C. E. Dyreson, R. T. Snodgrass, T. Isakowitz, and C. S. Jensen. Now in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[HJS94A] S. Hsu, C. S. Jensen, and R. T. Snodgrass. Valid-time Selection in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[HJS94B] S. Hsu, C. S. Jensen, and R. T. Snodgrass. Valid-time Projection in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[ISO92] ISO/IEC 9075: 1992 *International organization for Standardization/International Electrotechnical Commission—Database Language SQL.*

[Jen93] Jensen, C. S. (ed.) A Consensus Test Suite of Temporal Database Queries. Technical Report R 93-2034, Department of Mathematics and Computer Science, Aalborg University, Denmark, November 1993.

[JMR91] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4), pp. 461–473, December 1991.

[JMR91] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7), pp. 513–547, December 1994.

[JSS94B] C. S. Jensen, R. T. Snodgrass, and M. D. Soo. The TSQL2 Data Model. Commentary, TSQL2 Design Committee, September 1994.

[KSL94] N. Kline, R. T. Snodgrass, and T. Y.C. Leung. Aggregates in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[LJS94] T. Y. C. Leung, C. S. Jensen, and R. T. Snodgrass. Update in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[LS93] D. Lomet and B. Salzberg. Transaction-Time Databases. Chapter 16 of *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, pp. 388–417, 1993.

[Mai85] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1985.

[RS94] J. F. Roddick and R. T. Snodgrass. Schema Versioning Support in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[SA+94b] 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. The TSQL2 Language Specification. *The TSQL2 Language Specification*, September 1994.

[SAA+94] 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 (preliminary version). *SIGMOD Record*, 23(1), March 1994.

[SJ94] R. T. Snodgrass and C. S. Jensen. The From Clause in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[SJG94] R. T. Snodgrass, C. S. Jensen, and F. Grandi. Schema Specification in TSQL2. Commentary, TSQL2 Design Committee, September 1994.

[Sno92] R. T. Snodgrass. Temporal Databases. *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, A. U. Frank, I. Campari, and U. Formentini (eds.), Vol. 639 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 22–64, September 1992.

[Sno94] R. T. Snodgrass (ed.) An Evaluation of TSQL2. Commentary, TSQL2 Design Committee, July 1994.

[Soo93] M. D. Soo and R. T. Snodgrass. User-defined Time in TSQL2. Commentary, TSQL2 Design Committee, September 1994.