Valid-time Selection in TSQL2

September 16, 1994

A TSQL2 Commentary

The TSQL2 Language Design Committee

Title	Valid-time Selection in TSQL2
Primary Author(s)	Suchen Hsu, Christian S. Jensen and Richard Snodgrass
Publication History	May 1992. TempIS Document No. 30. September 1994. TSQL2 Commentary.

Contents

List of Tables

Abstract

Temporal databases have now been studied for more than a decade, and numerous temporal query languages have been proposed. One of the essential components of a temporal query language is *valid-time selection*, which allows the user to retrieve tuples based on their underlying valid-times. We have previously surveyed valid-time selection and projection in nine temporal query languages, primarily SQL and Quel extensions. Based on that survey, this document proposes a specific design of the valid-time selection component of the consensus temporal query language TSQL2 that is currently being designed.

1 Introduction

We have previously examined the valid-time selection and projection components of nine different temporal query languages. The examination established a foundation for designing the valid-time selection and projection components of the consensus Temporal SQL2 that is currently being designed. This document is a proposal for the valid-time selection component. It is attempted to base the proposal on the experiences accumulated in previous proposals, and an effort is made to explicitly address important design decisions.

The document is structured as follows. Initially, we describe six general criteria for what is a good language design. The criteria provide guidelines. However, they may conflict at times, and a single general criterion may be applied in several ways in a concrete design. Then, the proposal for valid-time selection is introduced and subsequently discussed in three parts. Mechanisms that involve the referencing and construction/extraction of various kinds of timestamps are covered first. Then, comparison operators are covered. Finally, miscellaneous aspects of the design are discussed. The general design is first presented, and then central design issues are discussed in more detail. The document is ended with a summary.

2 Design Criteria

As a guide for making appropriated design decisions and as and a means of evaluating the proposal, we present some language design criteria. These criteria are *expressive power, consistency, clarity, minimality, orthogonality,* and *independence.* Initially, each criterion is described. Then the interactions among the criteria are exemplified.

Expressive Power This criterion indicates that the language must exhibit a functionality that makes it suitable for its intended applications and does not impose undesirable restrictions on the queries that may be formulated.

This does not mean that providing a lot of operators and functions is necessarily better than a more restricted set. For example, this criterion has implications for our choice of comparison operators that involve valid time.

Consistency For the task at hand, this criterion has at least four implications. First, the design must be consistent with the syntax for user-defined time support in TSQL2. For example, it should use the formats for temporal constants adopted there [Soo & Snodgrass 1992]. Second, the design should be upward compatible with SQL2. This indicates that defaults should be chosen carefully. Third, the proposal should be consistent with the designs of other aspects of TSQL2. Fourth, the syntax should be internally consistent. For instance, mixing postfix and prefix operators is not considered a good design.

Clarity The syntax should clearly reflect the semantics of the language. This aids in formulating and understanding queries. Applications of the principle include the meaningful naming of operators, a proper choice of clauses (to obtain well-structured queries), and a consistent naming style. As a specific example, inclusion of a period comparison operator such as `OVERLAPS` increases readability when compared with an equivalent predicate based on event extraction and event comparison operators.

Minimality The principle of minimality indicates that as few as possible new reserved words and clauses should be introduced and added to those already present in SQL2. It also indicates that new operators should not be included if they duplicate the functionality already provided by existing operators. This is intended to ensure that users will not be unnecessarily burdened by a large set of operators.

Orthogonality it should be possible to freely combine query language constructs that are semantically independent. The Zero–One-Infinity principle may be seen as a more specific design criterion. This criterion states that the only reasonable numbers in a design are zero, one and infinity and that other numbers are unintuitive to users. For example, restricting the number of tuple variables that may be declared in a query to another number (e.g., 15) appears to have no logical explanation and is difficult to remember.

Independence Obeying this criterion ensures that each function is accomplished in only one way. Designing functions to be independent and non-overlapping, orthogonality, minimality, and consistency may be achieved.

Although we would like the design to satisfy all of the criteria, this is not possible because the criteria themselves are conflicting.

An example follows. As will be seen later, timestamp referencing, event extraction, and event time comparison are fundamental to valid-time selection. However, the (functionality-wise unnecessary) use of period comparison operators improve the readability. If we provide event operators only, the design satisfies the minimality and independence criteria, but not that of clarity. Using only event-based operators may result in confusing and erroneous predicates. Conflicts are resolved by retaining duplicating operators if they are used frequently or if their event-based equivalents are complicated. For example, the following two predicates are equivalent.

```
a OVERLAPS b
```

```
(END(a) > BEGIN(b)) and (END(b) > BEGIN(a))
```

The second predicate is hard to understand and `OVERLAPS` is a frequently used operator. In a case like this, the priority of clarity is higher than that of minimality and independence; `OVERLAPS` is included in the proposal.

3 Valid-time Selection in TSQL2

The appendix (Section A) describes the syntax modifications to the SQL2 language necessary to include valid-time selection.

3.1 Overview

The proposal includes four types of valid-time timestamps, namely intervals, instants, periods, and elements. Intervals are directed, unanchored durations of time (e.g., 2 minutes), and while they cannot be uses as valid times, they may be usd in expressions involving valid times. For details on intervals, see [Soo & Snodgrass 1992].

This section discusses three categories of operators related to valid-time selection, namely *extractors*, *constructors*, and *comparison operators*. Operators in the first category, e.g., `BEGIN`, create a new timestamp, e.g., a starting instant, by extraction from an argument timestamp, e.g., a period. To exemplify constructors, `INTERSECT` creates a period as the intersection of two overlapping periods. The operator `OVERLAPS` which tests whether two periods overlap illustrates comparison operators. As indicated, these categories of operators relate to instants as well as periods and elements.

The proposal does not include new clauses for timeslice and temporal predicates, both of which are commonly present in other temporal query languages.

Another characteristic is that valid-time selection and valid-time projection are considered orthogonal and therefore are completely separated in the design. Thus, the timestamps of results of queries are not defined by any valid-time selection operator. Temporal ordering is assumed to be the responsibility of aggregate functions and is thus not addressed here.

Table 1 is a summary of our proposal. In that table, `event` denotes an argument of type `DATE`, —tt TIME, or `TIMESTAMP`, `period` denotes an argument of period type, and `element` denote an argument of element type. The details are discussed in the next three sections.

3.2 Timestamp Referencing and Timestamp Extraction and Constructors

`VALID(correlation name)` is used for indicating timestamps of tuples. The alternative is an explicit reference such as `EMPLOYEE.PERIOD` where `EMPLOYEE` is a tuple variable and the postfix operator `PERIOD` returns the (period-valued) timestamp of an argument tuple. Another alternative is to overload correlation names to assume both a tuple and the timestamp of a tuple, depending on the context.

Several of the operators in this proposal are adapted from Soo's proposal [Soo & Snodgrass 1992]. Whereas TSQL and HSQL use a postfix notation for event extraction, this proposal employs a prefix, function-style notation for extraction. There are three reasons for this decision.

First, the prefix notation avoids confusing extraction with a reference to an attribute of a relation. For example, if `BEGIN` is an operator that extracts the first event of a timestamp, `EMPLOYEE.BEGIN` may be confused with a reference to an attribute `BEGIN` of the `EMPLOYEE` relation. In contrast, `BEGIN(EMPLOYEE)` does not have this problem of disallowing certain attribute names. Second, the prefix notation is more generally applicable than is the postfix notation which may not be used conveniently for operators that

Operation Type	Operators
timestamp referencing	`VALID`(correlation name)

event extraction	`BEGIN(event) BEGIN(period) BEGIN(element)`
	`END(event) END(period) END(element)`

period extraction	`FIRST(period) FIRST(element)`
	`LAST(period) LAST(element)`

event constructors	`FIRST(event, event)`
	`LAST(event, event)`

period constructors	`PERIOD(event, event)`
	`INTERSECT(period, period)`

element constructors	`INTERSECT(element, element)`
	`element + element`
	`element - element`
	May Also be Applied to Periods and Events

element comparison	`element PRECEDES element`
	`element = element`
	`element OVERLAPS element`
	`element CONTAINS element`

period comparison	`period PRECEDES period`
	`period = period`
	`period OVERLAPS period`
	`period MEETS period`
	`period CONTAINS period`

event time comparison	`event PRECEDES event`
	`event = event`
	`event OVERLAPS event`
	`event MEETS event`
	`event CONTAINS event`

mixed comparison	Comparison among elements, periods, and events
time slice clause	*None*
temporal ordering	*Use Aggregate Functions*
separate clause for valid-time selection	*No*
valid-time projection and selection	*Separated*

Table 1: Overview of Valid-time Selection

accept general temporal expressions as arguments. Third, other operators are prefix (or infix); thus, avoiding the postfix notation improves consistency.

The event extractors, `BEGIN` and `END`, return the first and last events, respectively, of an event, an period, or an element. The period extractors, `FIRST` and `LAST`, may be applied to timestamps of period or element type and return the first and last periods of the arguments, respectively.

The event constructors, `FIRST` and `LAST`, are new operators not provided by existing languages. Both of them take two events as arguments and return an event; `FIRST` returns the earlist event, and `LAST` returns the latest event. Following is an example showing the use of these two operators.

PERIOD(FIRST(e_1, e_2), LAST(e_1, e_2))

There is hardly any clearer way to construct a period from two events whose order is not known.

The `PERIOD` function returns a period with two argument events as the delimiters. While the `INTERSECT` operator is not strictly necessary for reasons of functionality, it is still included because it is used frequently. This operator returns the period which is the intersection of two argument periods.

Three operators exist for constructing elements. The set of elements is closed under the binary operations of intersection, union, and difference. Thus, `INTERSECT`, `+`, and `-`, are included for constructing new elements. Since events and periods are special cases of elements, the former two may also appear as arguments.

3.3 Comparison Operators

It is essential that a temporal query language allows for the convenient comparison of timestamps.

A powerful set of comparison operators that satisfies the six design criteria is provided. For element comparison, `PRECEDES`, `=`, `OVERLAPS`, and `CONTAINS` are available; For period comparison `MEETS` is provided. Note that since events may be perceived as special cases of periods, events may occur as arguments where periods are allowed. Similarly, periods may occur where elements may occur. Table 2 gives the definition of these operators with the assumption that E_1 and E_2 are elements and I_1 and I_2 are periods. Operator `MEETS` has no natural generalization for elements. Observe that for events, operators `=`, `OVERLAPS`, and `CONTAINS` are equivalent.

Operator	Definition
E_1 `PRECEDES` E_2	`END(`E_1`)` is earlier than `BEGIN(`E_2`)`
E_1 `=` E_2	E_1 and E_2 are identical
E_1 `OVERLAPS` E_2	the intersection of E_1 and E_2 is not empty
E_1 `CONTAINS` E_2	each event in E_2 is contained in E_1
I_1 `MEETS` I_2	`END(`I_1`) PRECEDES BEGIN(`I_2`)` and there are no events between `END(`I_1`)` and `BEGIN(`I_2`)`

Table 2: Definition of Comparison Operators

This set of operators is complete in the sense that all possible relationships between two periods or two events are covered [Allen 1983, Soo & Snodgrass 1992]. Indeed, a smaller set of operators could have been chosen had completeness, not user-friendliness, been the main concern. Table 3 show the equivalence of Allen's period comparison operators and the proposed operators.

Allen's Operators	Proposed Operators
`a before b`	`a PRECEDES b`
`a equal b`	`a = b`
`a overlaps b`	`a OVERLAPS b AND END(a) PRECEDES END(b)`
`a meets b`	`END(a) = BEGIN(b)`
`a during b`	`BEGIN(b) PRECEDES BEGIN(a)` `AND` `END(a) PRECEDES END(b)`
`a start b`	`BEGIN(a) = BEGIN(b)` `AND` `END(a) PRECEDES END(b)`
`a finish b`	`BEGIN(b) PRECEDES BEGIN(a)` `AND` `END(a) = END(b)`

Table 3: Comparison Expressions in Allen's Definition and the Proposal

According to Table 3, the first two pairs of operators are equivalent, with naming being the only difference. Allen's operator `overlaps` is a asymmetric. We have chosen the more common symmetric counterpart for TSQL2. Operator `meets` requires that the ending event of the first argument is identical to the starting event of the second argument. That is tested easily using event extraction and equality. The `MEETS` included here is harder to express with other operators and is expected to be at least as useful as `meets` in practice. Because the definitions of `during` and `CONTAINS` are slightly different (`a during b` means `a` started later than `b` and ended earlier than `b`), it is necessary to use two subexpressions to accomplish the definition of `during`. However, the operator `CONTAINS` as defined in the proposal is expected to be used more often than `during`. If a a "pure" `CONTAINS`, i.e., a `during`, is needed, the expression in the table provides the functionality.

Note that we do not provide the `ADJACENT` operator (found in TSQL) in our proposal. This choice is made because the predicate, I_1 `ADJACENT` I_2, is equivalent to I_1 `MEETS` I_2 `OR` I_2 `MEETS` I_1, the semantics of which is clear.

The last two operators, `start` and `finish`, are expected to be used only sporadically in queries, so we do not provide equivalent operators for them. In total, we have attempted to strike a balance between minimality and clarity.

Below, two sample queries are expressed using the proposal.

Q1. List all of the employees who worked during all of 1991.

```
SELECT Name
FROM Employee
WHERE VALID(Employee) CONTAINS PERIOD(DATE '01/01/1991', DATE '12/31/1991')
```

Q2. List all the employees who work in the company at some time when Tom is in the Toy department.

```
SELECT E1.Name
FROM Employee E1, Employee E2
WHERE E2.Name = "Tom" AND E2.Dept = "Toy" AND VALID(E1) OVERLAPS VALID(E2)
```

3.4 Additional Aspects

So far, we have accounted for most of Table 1, but a few design decisions still remain to be discussed.

We did not include a special timeslice clause and a separate clause for temporal selection. We also decided to separate temporal ordering from temporal selection. We now discuss the reasons for those decisions.

Most language proposals include a timeslice clause, but after a careful examination of timeslice, we have not included a special timeslice clause, for two reasons. First, a timeslice may be expressed in a clear way without using a special clause. For example, the timeslice in TSQL

```
SELECT Name
FROM Employee
TIME-SLICE [1.1.1990, 12.31.1990]
```

is equivalent to

```
SELECT Name
FROM Employee
WHERE VALID(Employee) OVERLAPS PERIOD '01/01/1990 - 12/31/1990'
```

The meaning of this latter predicate is clear. Omitting a timeslice clause improves minimality without adversely affecting clarity. Second, the independence criterion for temporal predicates is violated if a timeslice clause is included.

Next, we present the considerations that led to not including a new clause for temporal selection. If we provide a new clause, e.g., `WHEN`, possible advantages include a increased syntactically clarity, separation of temporal predicates from non-temporal predicates, and a more structural language. Each of these may lead to increased clarity.

The advantages of not providing a new clause are less reserved words (minimality) and one clause for all predicates (consistency). Also, it is not clear what should be the boundary between a `WHEN` and a `WHERE` clause, i.e., what should go where. Because of user-defined time attributes, we cannot avoid temporal predicates in the `WHERE` clause totally. The advantage of separating temporal predicates from non-temporal predicates is then decreased. Further, it is still possible to write structured queries without a `WHEN` clause.

The language should allow the use of user-defined time attributes and valid times together in the same predicates. With a new clause, this combination of user-defined times with timestamps is not allowed. For example, adding a user-defined time attribute `PROJ-TIME` to the `EMPLOYEE` relation, the predicate

```
WHERE A.PROJ-TIME PRECEDES A
```

cannot be expressed using a combination of the `WHERE` clause and a new clause. Thus, it may be argued that adding a new clause actually decreases the expressive power of the language. Expressive power has the highest priority among the criteria, and it cannot be compromised for any reason.

We consider temporal ordering to be the responsibility of aggregate functions. In temporal ordering, a tuple is selected, not by testing it against a predicate, but by manipulating a group of tuples to get a correct order to obtain a desired tuples. Generally, whether a tuple is selected or not cannot be determined by testing the tuple against a predicate. An aggregate function takes a group of tuples, operates on all of the tuples, and returns a value. Temporal ordering functions fall into this category.

Aggregates are not covered in this document, so temporal ordering is not addressed.

The main reason to separate valid-time projection from temporal selection is added expressive power. When the two are mixed, it may be impossible or difficult to obtain adequate timestamps on result tuples. Separating valid-time selection and projection allows the two to be combined freely.

4 Summary

In a previous paper, we reviewed nine temporal query languages proposed in the past decade. These languages include five extensions to SQL, three extensions to QUEL, and a procedural language. Because their underlying data models were not the same, it is not easy to compare the features in each language. However, all of the features, functionalities, new clauses, and reserved words of these languages were examined carefully when this proposal for valid-time selection in TSQL2 was prepared. It has been a goal to build on the insights gained from the designs of previous temporal query languages.

Initially, six criteria, expressive power, consistency, clarity, minimality, orthogonality, and independence, were defined in order to guide the design.

We attempted to design simple but powerful language constructs with no unnecessary reserved words, clauses, and functions. Thus, many such components present in other language proposals have been eliminated while still supporting valid-time expression and predicates at least as completely as in other languages.

Key features included a clean separation of valid-time selection and valid-time projection. The orthogonality of these constructs is reflected clearly in the design. No new clause was added for valid-time selection, mainly because of a desire to be able to mix valid time and user-defined time attributes in predicates. Finally, Defaults have been chosen carefully.

While important, the notion of temporal ordering was not considered. The functionality of the temporal ordering is close to aggregate functions, and temporal ordering should be designed in that context.

The proposal did not address the transfer of timestamp values to data structures of a programming language program execution via cursors. In SQL2, cursors can return values of explicit attributes; an extension is required to support the transfer of timestamp values.

5 Acknowledgements

We thank Curtis Dyreson, Nick Kline, and Michael Soo for discussions, suggestions, and comments. This work was supported in part by NSF grants ISI-8902707 and ISI-9302244, IBM contract #1124 and the AT&T Foundation. Christian S. Jensen was also supported in part by the Danish Natural Science Research Council, grants no. 11–1089–1 SE and no. 11–0061–1 SE.

6 Bibliography

[Allen 1983] Allen, J.F. "Maintaining Knowledge about Temporal Intervals." *Communications of the Association of Computing Machinery*, 26, No. 11, Nov. 1983, pp. 832–843.

[Ariav 1986] Ariav, G. "A Temporally Oriented Data Model." *ACM Transactions on Database Systems*, 11, No. 4, Dec. 1986, pp. 499–527.

[Ben-Zvi 1982] Ben-Zvi, J. "The Time Relational Model." PhD. Dissertation. Computer Science Department, UCLA, 1982.

[Chamberlain et al. 1976] Chamberlain, D. D. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control." *IBM J.*, 20, No. 6 (1976), pp. 560–575.

[Gadia & Vaishnav 1985] Gadia, S.K. and J.H. Vaishnav. "A Query Language for a Homogeneous Temporal Database," in *Proceedings of the ACM Symposium on Principles of Database Systems*. Mar. 1985, pp. 51–56.

[Gadia 1988] Gadia, S.K. "A Homogeneous Relational Model and Query Languages for Temporal Databases." *ACM Transactions on Database Systems*, 13, No. 4, Dec. 1988, pp. 418–448.

[Gadia 1992] Gadia, Shashi K. "A Seamless generic extension of SQL for querying temporal data." preliminary. Computer Science Department, Iowa State University. Mar. 1992.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. "INGRES–A Relational Data Base Management System," in *Proceedings of the AFIPS National Computer Conference*. Anaheim, CA: AFIPS Press, May 1975, pp. 409–416.

[Jones et al. 1979] Jones, S., P. Mason and R. Stamper. "LEGOL 2.0: A Relational Specification Language for Complex Rules." *Information Systems*, 4, No. 4, Nov. 1979, pp. 293–305.

[Martin et al. 1987] Martin, N.G., S.B. Navathe and R. Ahmed. "Dealing with Temporal Schema Anomalies in History Databases," in *Proceedings of the Conference on Very Large Databases.* Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 177–184.

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

[Navathe & Ahmed 1989] Navathe, S. B. and R. Ahmed. "A Temporal Relational Model and a Query Language." *Information Sciences*, 49 (1989), pp. 147–175.

[Sarda 1990A] Sarda, N. "Extensions to SQL for Historical Databases." *IEEE Transactions on Knowledge and Data Engineering*, 2, No. 2, June 1990, pp. 220–230.

[Sarda 1990B] Sarda, N. "Algebra and Query Language for a Historical Data Model." *The Computer Journal*, 33, No. 1, Feb. 1990, pp. 11–18.

[Sarda 1990C] Sarda, N. "Time-rollback using Logs in Historical Databases." Technical Report. Indian Institute of Technology. June 1990.

[Sarda 1990D] Sarda, N. "Design of a Historical Database Management System," in *Proceedings of the CSI Indore Chapter Conference (invited paper).* Aug. 1990.

[Snodgrass & Ahn 1985] Snodgrass, R. and I. Ahn. "A Taxonomy of Time in Databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data.* Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236–246.

[Snodgrass 1987] Snodgrass, R. "The Temporal Query Language TQuel." *ACM Transactions on Database Systems*, 12, No. 2, June 1987, pp. 247–298.

[Soo & Snodgrass 1992] Soo, M. and R. Snodgrass. "Mixed Calendar Query Language Support for Temporal Constants." TempIS Technical Report 29. Computer Science Department, University of Arizona. October 30, 1991 1992.

[Tansel & Arkun 1986] Tansel, A.U. and M.E. Arkun. "HQuel, A Query Language for Historical Relational Databases," in *Proceedings of the Third International Workshop on Statistical and Scientific Databases.* July 1986.

A Modified Language Syntax

The organization of this section follows that of the SQL2 document. The syntax is listed under corresponding section numbers in the SQL2 document. All new or modified syntax rules are marked with a bullet ("•") on the left side of the production.

Where appropriate, we provide disambiguating rules to describe additional syntactic and semantic restrictions. We assume that the reader is familiar with the SQL2 standard, as well as with the user-defined time proposal, and that a copy of the standard and the proposal is available for reference.

A.1 Section 6.10 <cast specification>

Casting between data types is extended to include the temporal types. No syntactic changes or additions are required.

Additional syntax rules:

1. The table showing allowable data conversions is augmented to add the temporal element (TE) data type.

	<data type> of *TD*												
<data type> of *SD*	EN	AN	VC	FC	VB	FB	D	T	TS	YM	DT	P	TE
EN	Y	Y	Y	Y	N	N	N	N	N	M	M	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N
C	Y	Y	M	M	Y	Y	Y	Y	Y	Y	Y	Y	Y
B	N	N	Y	Y	Y	Y	N	N	N	N	N	N	N
D	N	N	Y	Y	N	N	Y	N	Y	N	N	Y	Y
T	N	N	Y	Y	N	N	N	Y	Y	N	N	Y	Y
TS	N	N	Y	Y	N	N	Y	Y	Y	N	N	Y	Y
YM	M	Y	Y	Y	N	N	N	N	N	Y	N	N	N
DT	M	Y	Y	Y	N	N	N	N	N	N	Y	N	N
P	N	N	Y	Y	N	N	N	N	N	M	M	Y	N
TE	N	N	N	N	N	N	N	N	N	N	N	N	Y

2. If *SD* is C, D, T, or TS and *TD* is TE then the conversion is first done to the *P* type.
3. If *SD* is P and *TD* is TE, then the conversion is into a temporal element containing one period.

A.2 Section 6.11 <value expression>

Value expressions are augmented to include expressions evaluating to temporal elements.

<value expression> ::=
- | <temporal element value expression>

A.3 SECTION 6.?? <period value function>

This is a new section.

<period value function> ::=
- | `VALID` <left paren> { <table name> | <correlation name> } <right paren>
- | `FIRST` <left paren> <temporal element value expression> <right paren>
- | `LAST` <left paren> <temporal element value expression> <right paren>

Additional general rules:

1. Use of `VALID` is allowed only on vaid time state or bitemporal state tables that are partitioned, and denotes a maximal period in the timestamp of the underlying tuple.
2. `FIRST` (`LAST`) extracts the first (last) maximal period from the temporal element.

A.4 Section 6.?? <temporal element value expression>

The following are new nonterminals introduced into the language.

<temporal element value expression> ::=
- <temporal element value term>
- | <temporal element value expression> { <plus sign> | <minus sign> } <temporal element value term>

<temporal element value term> ::=
- <temporal element value factor>

<temporal element value factor> ::=
- <temporal element value primary>

<temporal element value primary> ::=
- <temporal element value function>

Additional general rules:

1. A primary of a <table name> or <correlation name> denotes the datetime or period timestamping of the underlying tuple.
2. '`+`' ('`-`') on temporal elements is set union (difference).

A.5 Section 6.?? <temporal element value function>

A new nonterminal, <temporal element value function>, is added.

<temporal element value function> ::=

- VALID <left paren> { <table name> | <correlation name> } <right paren>
- | INTERSECT <left paren> <temporal element value expression> <comma> <temporal element value expression> <right paren>

Additional general rules:

1. Use of VALID denotes the temporal element timestamping of the underlying tuple, which must be associated with a valid time or bitemporal table that has not been partitioned.
2. Intersection of temporal elements is set intersection.

A.6 Section 8.11 <overlaps predicate>

The syntax is unchanged. However, the applicable types are broadened to include temporal elements.

Operand 1	*Operator*	*Operand 2*
datetime/period/element	=	*datetime/period/element*
datetime/period/element	PRECEDES	*datetime/period/element*
datetime/period/element	OVERLAPS	*datetime/period/element*
datetime/period/element	CONTAINS	*datetime/period/element*
datetime/period/element	MEETS	*datetime/period/element*