Mixed Calendar Query Language Support for Temporal Constants

Michael Soo, Jie Li and Richard Snodgrass

May 27, 1999

TR-41

A TimeCenter Technical Report

Title	Mixed Calendar Query Language Support for Temporal Constants
Author(s)	Michael Soo, Jie Li and Richard Snodgrass
Publication History	May 1999. A TIMECENTER Technical Report.

TIMECENTER Participants

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

We have exhibited examples of many calendars, and described how a particular calendar can varying depending on its properties. We emphasize that database management systems attempting to support time values must be capable of supporting any notion of time that is of interest to the user population. We address this problem by allowing a calendar to be parameterized by its properties and by supporting multiple calendars within the DBMS.

1 Introduction

Time is an attribute of most real-world phenomena, and, consequently, temporal information has been stored in databases for many years. Recent surveys show that research concerning time in databases has been, and continues to be, very active [Bolour et al. 1982, McKenzie 1986, Snodgrass 1990, Soo 1991, Stam & Snodgrass 1988].

Among database management system vendors, the need to provide time support has also been recognized. We are aware of several commercial products that provide some sort of temporal support [Date & White 1990, Oracle 1987, Tandem 1983]. Unfortunately, most are ad-hoc and widely regarded as poorly designed [Date 1988].

This paper attempts to remedy this situation. We investigate how to augment a conventional relational query language with time values, i.e., *temporal constants*. The language to be modified is the Structured Query Language (SQL) or, to be more specific, SQL2, the next standardization of SQL currently under development [Melton 1990]. This proposal represents the first phase of an on-going project to systematically augment SQL with time support.

The design will focus on the *user level*, i.e., the time abstraction visible to an applications programmer or end user as architectural or representation considerations. We are currently developing an underlying system architecture and physical representation.

Three broad goals motivated the design of this proposal. They are generality, usability, and minimality.

Generality refers to the ability of the system to support a wide variety of time domains. Cultural, societal, legal, and business practices influence both the meaning and usage of time within a group of people. For example, Western cultures generally measure time using the Gregorian calendar while Eastern cultures typically use a lunar calendar. As another example, universities measure time relative to two calendars, academic and non-academic. The query language must provide facilities general enough to support these different views of time.

Usability refers to the ease of use of the system. The constructs provided by the language should be expressive and easy to understand, and their behavior should be intuitive and predictable.

Minimality is a characteristic of the extension relative to the rest of the language. The overall goal of this project is to be influential in future standardizations of SQL. Consequently, the easier the modifications are to incorporate into the language, the more likely that they will be accepted. We have striven for minimality in this respect.

The remainder of the paper presents our proposal. Section 2 discusses previous work related to temporal constants. General concepts about time and the measurement of time are the subject of Section 3. Section 4 describes the temporal modifications and extensions to SQL. Section 5 provides a comprehensive example of how particular temporal semantics can be implemented using this proposal.

Section 6 summarizes the contributions of this paper and compares our approach with other proposals. Future work concerning temporal extensions to SQL is the subject of Section 7. The appendices contain, among other material, a glossary of terms introduced in the paper, full syntax for all modified and additional SQL language constructs, and a summary of how the temporal semantics currently proposed for SQL2 could be implemented in this proposal.

2 Related Work

Several researchers have investigated database time from a conceptual viewpoint. Anderson developed a formal framework to support conceptual time spaces using inheritance hierarchies [Anderson 1982, Anderson 1983]. Like our extension, her model supports multiple conceptual times; this work can be considered a practical extension of the concepts developed by Anderson. However, our work differs in that it is designed as the first step in a general extension of SQL to support time, and as such, forms the basis for exploring temporal semantics beyond those of Anderson's. We will discuss these extensions when we consider future work in Section 7.

Clifford and Rao developed a framework for describing temporal domains using naive set theory and algebra [Clifford & Rao 1987]. This work allows a hierarchy of calendar independent domains to be built and temporal operators to be defined between objects of a single domain and between objects of different domains. The framework is powerful but lacks the ability to describe time domains that are inconsistent with domains of larger units. For example, *weeks* are inconsistent with *months* since a whole number of weeks do not ordinarily correspond to a single month. Our work removes this limitation by making the semantics of any conceptual time unit user-definable. The user is not tied to any predefined notion of time or time domain.

Many researchers, mostly within the artificial intelligence community, have investigated the problems of temporal representation and temporal inferencing. Allen motivated the *interval* as a fundamental temporal entity [Allen 1983]. He formalized the set of possible relationships that could hold between two intervals and developed an inference algorithm to maintain the set of temporal relationships between entities. We use Allen's work on representation and relationships as the basis for defining new temporal comparison operators in SQL.

Improvements to Allen's interval algebra have been proposed by several authors. These extensions are of interest since they improve the theoretical performance of Allen's inference algorithm, and, like our work, they augment the algebra with additional data types. Vilain extended Allen's algebra to include *time points*, that is, instantaneous events [Vilain 1982], and with Kautz investigated the computational complexity of inference algorithms using the interval and point-based representations [Vilain & Kautz 1986]. Space saving improvements to the inference algorithm were proposed by Ligozat and Bestougeff who showed that the set of interval relations can be partitioned into equivalence classes [Ligozat & Bestougeff 1989]. Keretho and Loganantharaj also address efficiency issues and extend the algebra to include quantitative, as opposed to qualitative, comparisons [Keretho & Loganantharaj 1990A, Keretho & Loganantharaj 1990B, Keretho & Loganantharaj 1990C]. While these papers deal mainly with temporal inferencing and are not directly related to our research, we mention them here because they expand on Allen's work.

Other time extensions to SQL have been proposed. Date proposed augmenting SQL with date and time data types [Date 1988]. He extended SQL with facilities to support a single calendar, the Gregorian calendar. Also included were syntax and semantics for arithmetic and boolean expressions involving time. A single unified data type, the *interval*, was defined and used to represent both durations of time and events in time. This unification allows a high degree of orthogonality in temporal expressions but causes semantic difficulties since the distinction between event and duration objects is blurred. Also, the specialization of the solution to a single calendar limits its generality. We will return to this point in Section 6.

Many other researchers have developed sophisticated temporal data models and extended SQL to support these data models [Ariav 1986, Ben-Zvi 1982, Martin et al. 1987, Navathe & Ahmed 1987, Navathe & Ahmed 1989, Sarda 1990A, Sarda 1990B, Sarda 1990C, Sarda 1990D]. Generally, this line of research has ignored the issue of temporal constants or has assumed the use of a single calendar system. Additional papers concerning temporal data models and query languages other than SQL can be found in the collected bibliographies on time in databases [Bolour et al. 1982, McKenzie 1986, Soo 1991, Stam & Snodgrass 1988].

In the commercial arena, as previously mentioned, several systems with support for temporal data types exist [Oracle 1987, Tandem 1983]. These implementations are limited in scope and are, in general, unsystematic in their design. Date provides a thorough critique of one of the systems, DB2 [Date & White 1990, Date 1988].

Lastly, and most importantly, date and time support very similar to that in DB2 is currently being proposed for SQL2 [Melton 1990]. SQL2 corrects some of the inconsistencies in the time support provided by DB2 but inherits its basic design limitations. We will discuss the SQL2 proposal in detail in Section 6.

3 Physical Time, Calendars, and Calendric Systems

In this section, we describe the basic model of time that we propose. We begin by summarizing how time values are represented internally within the database management system and then continue by introducing the concepts of calendars and calendric systems.

3.1 Physical Time

We assume the continuous time-line is quantized into *chronons* of fixed duration, and the granularity of a time value at the query language level is exactly one chronon. The set of chronons form a finite, linear, and totally-ordered set of time values with a defined identity relation.

In a physical relation, time values are represented by *time-stamps*, numeric values representing chronons. Operations on time values are performed by executing analogous operations on time-stamps corresponding to those temporal values. The exact semantics of this internal representation are described elsewhere [Dyreson & Snodgrass 1991], but the details are not relevant here. We do note that time-stamps are of two sizes, *general* and *non-general*. General time-stamps are able to store a time values over an extended range and to a high granularity. Non-general time-stamps are able to represent a narrow range of time values with high granularity or an extended range of time values with low granularity. Full details are available elsewhere [Dyreson & Snodgrass 1991]. Section B.5 shows how a user may specify a temporal data item to be either general or non-general.

We note that time-stamps, while having a precise semantics tied to physical clocks, are independent of an interpretation implied by a user perspective. Such an interpretation, of which there could be several, are provided by calendars, which we now describe.

3.2 Calendars

A *calendar* is a human abstraction of time. Calendars define the time values of interest to a user, usually over a specific segment of the physical time-line. A calendar familiar to many is the Gregorian calendar, based on the rotation of the Earth on its axis and its revolution around the Sun. Some western cultures have used the Gregorian calendar since the late 16th century to measure the passage of time. As another example, Islamics generally use a lunar calendar, based on the amount of time required for the Moon to rotate around the Earth.

The Gregorian and lunar calendars are examples of daily and monthly calendars, but, in general, a calendar can measure time using any well-defined time unit. For example, an employee time card can be regarded as a calendar which measures time in eight hour increments and is only defined for five days of each week. We note that many different calendars exist, and that no calendar is inherently "better" than another; the value of a particular calendar is wholly determined by the population that uses it. Table 1 lists several example calendars.

We emphasize that the usage of a calendar depends on the cultural, legal, and even business orientation of the user. For example, business enterprises generally perform accounting relative to some *fiscal year*.

Calendar	*Description*
UTC2	Revised universal coordinated time
Gregorian	Common western solar with months
Lunar	Common eastern lunar
Julian	Western solar with years and days
Meso-american	260 day cycles
Academic	Year consists of semesters
Common Fiscal	Financial year begins at New Year
Academic Fiscal	Financial year starts in Fall
Federal Fiscal	Financial year starts in October
Time card	8 hour days and 5 day weeks
3-shift Work Day	24 hour day divided into three shifts of 8 hours
Carbon-14	Time based on radioactive decay
Geologic	Time based on geologic processes

Table 1: Common Calendars

However, the definition of fiscal year varies depending on the enterprise. Universities may have their fiscal calendar coincide with the academic year in order to simplify accounting. Other institutions use the more common half-yearly or quarterly definitions of fiscal year.

The defining characteristics of a calendar can be partitioned into two sets, *intrinsic characteristics* which define the universal qualities of the calendar, and *extrinsic characteristics* which define the user-dependent or varying qualities of the calendar.

The intrinsic characteristics of a calendar define the intrinsic semantics of the calendar or components that depend directly on such semantics. For example, the duration of time units (e.g., week, month) and their interrelationships are intrinsic components of a calendar. Functions performing calendar defined computations are also intrinsic. For example, in the Gregorian calendar one could construct a *field extraction function*, `month_name_of`, which returns the name of the month of a given date. Similarly, a function `harvest_moon_date` could be used to compute the date of the harvest moon in a given year.

The extrinsic characteristics, termed *properties*, of a calendar vary depending on the orientation of the user, as discussed above. A typical calendar property is the language in which time values are expressed. For example, in the United States, English is used to express Gregorian dates, while the French language is used to express Gregorian dates in France. Other properties include the format of time constants ("January 1, 1900", and "1 January 1900" denote the same Gregorian date, but in different formats), and local adjustments to time such as daylight savings time in the United States.

We have identified a comprehensive set of calendar properties applicable to most of the calendars that we can imagine. Table 2 lists the properties. Calendars for which a particular property does not apply can ignore the value of the property, if it is defined. Appendix F contains a complete description of the properties in Table 2.

Properties, in conjunction with calendars, are crucial to supporting international use of the DBMS (c.f. [Digital 1991]). Specifically, we support parameterization of calendars by relations, termed *property tables*. Any table used as a property table must have two attributes, *property* and *value*, where *value* defines the named *property*. Both the property and value attributes must have the SQL type for string. For example, to accommodate *timezone* calculations one could specify the location of interest as a property. Using supporting information, such as timezone displacements, subsequent time calculations can be done relative to this location. A default property table is provided by the implementation. The properties contained in the default property table are active until overridden by a user defined property table.

Property	*Default Value*
Beginning	beginning
Default input distribution	uniform
Distribution input format	<value>␣with␣<distribution>␣distribution
Distribution output format	<value>␣with␣<distribution>␣distribution
Forever	forever
Indeterminacy input separator	~
Indeterminacy output separator	~
Instant input format	<month_of_year,english_month_names>␣ <day_of_month,arabic_numeral>,␣ <year,arabic_numeral>
Instant output format	<month_of_year,english_month_names>␣ <day_of_month,arabic_numeral>,␣ <year,arabic_numeral>
Interval input format	<month,arabic_numeral>␣months
Interval output format	<month,arabic_numeral>␣months
Locale	*No default value*
Missing distribution	missing
Now	now
Now separator	␣
Override input epoch	Gregorian
Period input separator	␣-␣
Period output separator	␣-␣
Period output delimiters	[)

Table 2: Property Defaults

We have exhibited examples of many calendars, and described how a particular calendar can varying depending on its properties. We emphasize that database management systems attempting to support time values must be capable of supporting any notion of time that is of interest to the user population. We address this problem by allowing a calendar to be parameterized by its properties and by supporting multiple calendars within the DBMS.

3.3 Calendric Systems

A *calendric system* is the time abstraction available in the query language. Calendric systems are defined as collections of calendars where each calendar covers a contiguous and non-overlapping portion of the physical time-line. We term such segments of the physical time-line *epochs*. It is possible, and likely, that a calendric system has gaps in its time-line which are not covered by any calendar.

Figure 1 illustrates an example calendric system, the Russian calendric system. For the area of the world we now call "Russia," the sequence of calendars shown was used to measure time over the given epochs. In the figure, the physical time-line is not shown to scale.

In prehistoric epochs, the Geologic calendar and Carbon-14 dating (another form of a calendar) are used to measure time. Later, during the Roman empire, the lunar calendar developed by the Roman republic was used. Pope Julius, in the 1st Century B.C., introduced a solar calendar, the Julian calendar. This calendar was in use until the 1917 Bolshevik revolution when the Gregorian calendar, first introduced by Pope Gregory XIII in 1572, was adopted. In 1929, the Soviets introduced a continuous schedule work week

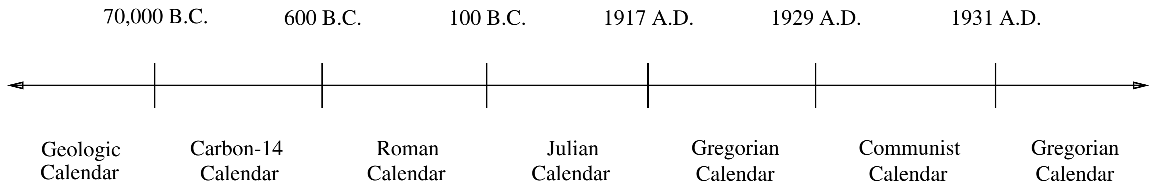

Figure 1: The Russian Calendric System

based on four days of work followed by one day of rest, in an attempt to break tradition with the seven day week. This new calendar, the Communist calendar, had the failing that only eighty percent of the work force was active on any day, and was abandoned after only two years in favor of the Gregorian calendar, which is still in use today.

This definition of calendric system provides a very general notion of time to the query language. Any proposal adding a time attribute domain to a database management system must support some calendric system. To date, all such proposals support calendric systems consisting of a single, pre-defined calendar over a very small epoch [Date 1988, Melton 1990]. For example, the SQL2 proposal supports only the Gregorian calendar and only over the epoch from 1 A. D. to 9999 A. D. Clearly, this is inadequate for applications, such as developing a historical record of ancient Egypt, that fall outside of this epoch. Also, applications that use time values that are within this epoch, but in a different calendar cannot be adequately supported. By allowing multiple calendars to exist within the database management system, and supporting calendric systems with multiple calendars, we provide a general notion of time that is able to express the entire history of an enterprise.

3.4 Summary

Calendars and calendric system define time at the query language level. Multiple calendars and calendric systems allow support for many different notions of time. In conjunction with calendar properties, they provide an important step toward generalization and internationalization of this limited but important component of the DBMS.

4 SQL Language Modifications

This section describes the language constructs we propose for temporal constant support in SQL. We concentrate on new temporal data types, operations, and constructs supporting calendric systems and properties. Appendix B contains Backus-Naur Form (BNF) syntax for all new and modified language constructs. The format of the BNF conforms with that in the SQL2 proposal document [Melton 1990].

4.1 Data Types

Our desire was to develop temporal data types with rich semantics that capture the intuitive and familiar concepts of time while, at the same time, minimizing impact on the language as a whole. We define three new temporal data types, events, intervals, and spans. Example queries involving these data types are shown in Table 3. The semantics and formal definitions of the new data types will be given shortly.

The rationale behind these data types is that they allow one to easily convey temporal ideas, and each has unique semantics distinct from the others. Events are necessary when describing isolated moments, intervals are useful for discussing a specific period of time, and spans are needed to discuss durations of time. While arguments can be made for additional data types or for a smaller set of types, this set represents the best compromise between our conflicting goals of usability and minimality.

Data Type	*Example Query*
event	"When did Ed walk through the door?"
interval	"Did Ed visit Alice between October and December?"
span	"How long was Ed in school?"

Table 3: Examples of Temporal Queries

We now describe the semantics of event, intervals, and spans in detail. The syntax for defining temporal data objects is shown in Appendix B.5.[1]

4.1.1 Events

An *event* is an isolated instant in time. An event, *e*, is said to occur at time *t* if *e* occurs at any time during the chronon (see Section 3.1) represented by *t*. For example, if the implementation fixes the granularity of a chronon as one second, then any two events which occur during a single second are assumed to happen simultaneously.

We assume here that events are known to occur to the precision of a chronon. That is, we assume that all events are *determinate*. In reality, events may not be known to such a fine precision. For example, we may know that Ed and Alice were married in September 1955, but we may not know the precise day (or even second) when the ceremony occurred. Work is currently under way to eliminate this assumption [Dyreson & Snodgrass 1992].

Specification of event values is done with a string-like notation. An `event` constant is syntactically delimited by vertical bars ("`|`").[2] The string of characters contained within the bars is interpreted as an event constant by a calendar.[3] A single vertical bar within an event constant is denoted by two consecutive vertical bars ("`||`").[4] Errors in translation are flagged with an appropriate message.

As examples of event constants, `|10:00 AM Tuesday|` and `|Midnight December 31, 1991|` are valid event constants in the Gregorian calendar, and `|Sunset Ramadan 1, 1872|` is a valid event constant in the Islamic calendar. Conversely, `|December 31, 1991|` is not a valid event constant since it does not fall within a single chronon.

4.1.2 Intervals

An *interval* is a set of contiguous chronons, I, with a minimum element t_{min} and a maximum element t_{max} such that $t_{min} \leq t_{max}$. For all chronons t where $t_{min} \leq t \leq t_{max}$, $t \in I$. If $t_{min} = t_{max}$ then I consists of a single chronon. We term such intervals *degenerate intervals*.

Specification of interval values is done with a string-like notation. An `interval` constant is syntactically delimited by square brackets ("`[]`").[5] The string of characters contained within the square brackets

[1]Time-stamps for events, intervals, and spans can be specified as `general` (see Section 3.1) using the syntax in Appendix B.5.

[2]Two vertical bars "(`||`)" is the SQL symbol for the string concatenation operator. A finite state automaton is able to distinguish the concatenation operator from an event delimiter with a one character lookahead.

[3]An alternative syntax we considered was to prefix the constant with a keyword indicating its type, e.g. `event '10:00:00 AM Tuesday'`. We discarded this idea since we believe that expressions involving temporal constants are more readable without the intrusion of keywords.

[4]The use of two consecutive delimiter characters to represent a single delimiter character within temporal values was chosen to be consistent with strings in SQL2. In strings, two consecutive quote symbols (`''`) represent a single quote (`'`).

[5]Square brackets are also used in Pascal embedded SQL statements to indicate array bounds. Syntatic analysis is able to distinguish this usage from interval bounds.

is interpreted to be an interval constant defined by a calendar. A single closing square bracket ("`]`") in an interval constant is denoted by two consecutive closing square brackets ("`]]`").

For example, in the Gregorian calendar `[1776]`, `[July 1776]`, `[July 4, 1776]`, `[|Noon July 3, 1776|,|Noon July 4, 1776|]` and `[Noon 7/3/1776 to Noon 7/4/1776]` are valid interval constants. However, `[Noon 7/4/1776, Noon 7/3/1776]` is invalid since `|Noon 7/4/1776|` occurs after `|Noon 7/3/1776|`.

In the previous examples, the format of the interval values enclosed within the square brackets varies considerably. We allow interval values to be any arbitrary string of characters. The meaning of that string of characters is determined by a calendar. Since input and output formats are calendar properties, the interpretation and display of arbitrary strings as interval constants can be supported.

Events and intervals are related in the sense that two events can uniquely determine an interval, and an interval's bounding events can always be determined. Built-in functions described in Section 4.3 are provided to accomplish this.

Since intervals are uniquely determined by their bounding events, we provide a constructor function `interval(`e_1`,` e_2`)` on the event expressions e_1 and e_2 which returns an interval `[`e_1`,` e_2`]`. Section 4.3 describes the constructor function.

4.1.3 Spans

A *span* defines an unanchored duration of time, that is, an amount of time with known length but no specific starting or ending chronons. For example, the span one week is known to have a duration of seven days, but one week can refer to any block of seven consecutive days. A span can be either positive, denoting forward motion in time, or negative, denoting backwards motion in time.

Specification of span constants is done with a string-like notation. A span constant is syntactically delimited by percent signs ("`%`"). The characters contained within the percent signs are interpreted as a span value defined by a calendar. A single percent sign within a span constant is denoted by two consecutive percent signs ("`%%`"). Errors in translation are flagged with an appropriate message. For example, `%1 week%`, `%2 years%` and `%-19 seconds%` are valid span constants in the Gregorian calendar. While we realize that percent signs make the syntax somewhat awkward, a better alternative is not available.

The duration of a span is either context dependent or context independent. A *fixed span* represents the same duration independently of its use. Conversely, the duration represented by a *variable span* depends on the context in which it appears. For example, the constant `%April%` represents a fixed span since the month of April always contains thirty days. An example variable span, in the Gregorian calendar, is the constant `%1 month%` which can represent anywhere from twenty-eight to thirty-one days, depending on context.

Variable spans provide convenience but can cause semantic difficulties if not carefully designed. Consider the following expression [Date 1988].

```
|12:00 PM May 31, 1991| + %1 month%
```

The result of the expression is an event representing a date. If, as one might expect, the expression computes the last day of June then the result returned is `|12:00 PM June 30, 1991|` since June has only thirty days. However, consider this expression.

```
(|12:00 PM May 31, 1991| + %1 month%) - %1 month%
```

Assuming that the addition operation still returns the value `|12:00 PM June 30, 1991|`, subtracting `%1 month%` can return either `|12:00 PM May 31, 1991|` or `|12:00 PM May 30, 1991|`

depending on the meaning of `%1 month%`. Both interpretations have valid uses, and neither should be imposed over the other.

The meaning of the variable span `%1 month%` is specific to a calendar. In the previous example, one calendar may ascribe the value `|12:00 PM May 31, 1991|` to the expression; another may ascribe the value `|12:00 PM May 30, 1991|`. We want to support many calendars within a single database management system. It is therefore necessary to provide a mechanism for specifying the meaning of both fixed and variable spans a calendar. By defining the semantics within the calendar, generality and usability are increased since the calendar developer is free to assign whatever meaning he or she desires.

It is important to note that fixed spans and variable spans are not different data types; they possess the exact same semantic properties. They differ only in how their meanings are assigned and computed, and these are calendar specific issues.

4.1.4 Examples Using the Temporal Types

To contrast the differences between the three temporal types, consider the temporal constants `|February 1991|`, `[February 1991]`, and `%February%`. `|February 1991|` is an indeterminate event (an event which is known to have occurred sometime during February 1991);[6] `[February 1991]` is a determinate interval denoting the month of February in the year 1991, and `%February%` is a variable span whose value is either twenty eight or twenty nine days. It is interesting to note that `[February 1991]` is equivalent to `interval(|Midnight 1 February 1991|, |Midnight 1 February 1991| + %February%)` and that, in this case, `%February%` can be replaced by `%1 month%`.

As an example of using the temporal data types, consider an employment database containing personnel information. We would like to record information such as the name and identification number of the employee, his or her birthday and age, and the period of that person's employment. A relation with the schema *employee(name, id, birthday, age, when_employed)* can be used to record the employee records. An SQL statement to create a base relation with this schema is shown in Figure 4.1.4.

```
create table employee (
    name character (20),
    id character (5),
    birthday event,
    age span,
    when_employed interval);
```

Figure 2: Example `create` Statement with Temporal Attributes

A sample instance of the employee relation is shown in Table 4. Note that the calendar being used in the example is the Gregorian calendar. While the temporal values physically stored in the database would be calendar independent, we have shown the time values in Gregorian format for simplicity.

4.2 Calendric System and Property Selection

This section describes how calendric systems are selected and how calendar properties are specified. Calendric systems can be specified globally or locally within a query. Similarly, property tables can be specified as either global defaults or on a per-item basis.

[6] We do not address indeterminacy in this paper, but we include this example since it illustrates the differences between events, intervals and spans. See Section 7.

name	*id*	*birthday*	*age*	*when_employed*
'Ed'	11111	|10:00 AM 1/21/29|	%62 years%	[1/1/50, 1/1/65]
'Alice'	22222	|4:00 AM 3/8/34|	%57 years%	[3/1/60, 3/1/70]
'Noland'	33333	|12:00 PM 12/17/59|	%32 years%	[6/1/80, 6/1/85]
'Dawn'	44444	|8:00 PM 9/15/60|	%31 years%	[8/1/85, 6/1/90]

Table 4: An Example Instance of the Employee Relation

4.2.1 Calendric System Specification

Calendric systems are specified by lexical scope in a sequence of SQL statements. The scope of a globally declared calendric system includes all statements up to the next global declaration of a calendric system. Global calendric systems are declared with the `declare calendric system` command. For a particular item, a local declaration can be used to override the declaration at the global scope. Local calendric systems are declared with the `as` clause.

```
...

declare calendric system as russian;

declare x cursor for
    select name, id, birthday,
        age with property_table_a, when_employed as american
    from employee
    where month_name_of(birthday) = 'Jinvar' and
        birthday < |2 Jinvar 1925| and
        age > %60 years% as american with property_table_b
        when_employed overlaps [1975];

procedure set_x_properties
    sqlcode
    set properties with x_property_table;

declare calendric system as american;

procedure open_x_cursor
    sqlcode
    open x;

...
```

Figure 3: Example of Calendric System and Property Selection

Figure 3 shows an excerpt of an SQL module. This example illustrates all of the ways in which calendric systems and property tables may be specified; it is not intended to be realistic. The `russian` calendric system is declared in the global scope. The scope of this declaration extends to the next global declaration, naming the `american` calendric system. The `russian` calendric system applies to the *birthday* and *age* attributes in the target list of the `select` statement since, unlike the *when_employed* attribute, no calendric system is locally declared for these attributes via an `as` clause. Similarly, the `russian` calendric system is used to resolve the function `month_name_of`, and to interpret the constants `|2 Jinvar 1925|` and `1975` ("Jinvar" is a phonetic translation into the Latin alphabet of the Russian word for "January"), while the `american` calendric system is used to interpret the span `%60 years%`. We note that, in this instance, the function `month_name_of` is defined via the `russian` calendric system and returns Russian month names. We assume that the implementation defines a globally scoped default calendric system which applies if no calendric system is globally declared.

Appendix B.6 contains syntax for calendric system specification.

4.2.2 Property Specification

Properties may be specified on both a global and a per-item basis. The properties contained in a property table are activated by executing a `set properties` command. Those properties remain in effect, and can influence any intervening temporal operations, until explicitly deactivated by another `set properties command`. In addition, a property table can be specified for a single operation, via a `with` clause.

The mechanisms for property selection are dynamic. This contrasts with the mechanisms for calendric system selection which are static. This distinction is consistent with our specification of properties as extensional—since the extension of a database can not be predicted a priori, the value of a property cannot be known at compile-time. Static specification of property tables is therefore not possible.

In Figure 3, the procedure `set_x_properties` contains a command which activates the properties contained in the property table `x_property_table`. Invocation of this procedure causes the properties contained in that table to be activated. These properties remain active until explicitly overridden by another `set properties` command. For example, if an application program calls the procedure `set_x_properties` prior to calling the procedure `open_x_cursor`, then the properties contained in `x_property_table` will override the properties in the default property table when the cursor is opened.

Conversely, naming a property table for an individual data item limits the activation of those properties to the processing of that data item. For example, associated with the attribute `age` in the `select` clause is a property table `property_table_a`. The properties in this table are activated temporarily while timestamps are being converted for the age attribute.

As mentioned in Section 3.2, a default table of property values is provided by the DBMS. The values in the default property table are customized by local site personnel and are expected to be appropriate for most situations at that site.

Appendices B.21 and B.2 contain syntax for property table specification.

4.3 Built-in Functions

Having defined the semantics of the basic data objects, it is convenient to have simple mechanisms for data conversion and manipulation. To accomplish these tasks, we define some built-in functions in Table 5. The set of built-in functions in can be classified into three classes, data destructors, data constructors, and miscellaneous functions.

The destructor functions `begin` and `end`[7] return an interval's beginning and ending events, respectively.

[7] An alternative syntax for selecting the delimiting events of an interval would use a postfix notation as in I.`begin` or I.`end` for an interval I. We chose to use functional prefix notation since the SQL2 proposal defines several temporal functions (`extract` and datetime value functions) using this syntax.

The functions `interval`, `intersect`[8] and `span` construct new data values from their arguments. The `interval` function returns an interval bounded by two events, and the `intersect` function computes the interval representing the overlap of two intervals. Lastly, the `span` function constructs a span representing the duration of an interval.

The four additional functions, `absolute`, `first`, `last`, and `present`, perform miscellaneous operations. Since spans may be of negative duration, the `absolute` function is provided to return the absolute value of a span. The `first` and `last` functions accept two events as arguments. `first` returns the earliest event, and `last` returns the latest event.[9] Finally, `present` is a zero-argument function which returns the present time.

Built-in Function	*Explanation*
event ← `begin(`*interval*`)`	Returns the *event* delimiting the beginning of the *interval*
event ← `end(`*interval*`)`	Returns the *event* delimiting the end of the *interval*
interval ← `interval(`$event_1$`, `$event_2$`)`	Returns the *interval* $[event_1, event_2]$ if valid, `null` otherwise
interval ← `intersect(`$interval_1$`, `$interval_2$`)`	Returns the overlap of $interval_1$ and $interval_2$ if they intersect, `null` otherwise
span ← `span(`*interval*`)`	Returns the duration of the *interval*
span ← `absolute(`*span*`)`	Returns the absolute value of *span*
event ← `first(`$event_1$`, `$event_2$`)`	Returns $event_1$ if $event_1 \leq event_2$, $event_2$ otherwise
event ← `last(`$event_1$`, `$event_2$`)`	Returns $event_1$ if $event_1 \geq event_2$, $event_2$ otherwise
event ← `present`	Returns the present time

Table 5: Built-in Functions for Data Conversions

The functions shown in Table 5 are calendar independent. Additional functions may possibly be defined by a calendar.[10] For example, conversion functions to translate time values between different timezones can be implemented using a calendar defined function.

4.4 Special Time Values

The physical time-line introduced in Section 3.1 is, by necessity, finite. In this section, we discuss data values that are unique to time, and develop mechanisms by which events outside of the physical time-line can be represented.

[8] The keyword `intersect` also denotes the intersection operator on relations. We decided to reuse this keyword to reduce the number of new keywords added to the language and to prevent programmer confusion over similar keywords. Using lookahead symbols an LR parser can distinguish the different uses of the keyword. Similar remarks apply to `begin`, `end`, `interval`, `span`, `absolute`, `first`, and `last`.

[9] The functions `first` and `last`, together with `begin`, `end`, and interval, are able to implement all interesting interval constructors we can imagine, including `intersect`. In this sense, `intersect` is syntactic sugar. It is, however, a commonly used temporal operation, and we include it for usability.

[10] To distinguish calendar defined functions from SQL defined keywords, we adopt the convention that calendar functions are prefixed with an ampersand (“`&`”). The ampersand character is also used to indicate embedded SQL statements, e.g., `&sql`. This does not conflict with our usage.

Table 2 introduced the the concepts of *beginning* and *forever*, that is, special time values representing negatively infinite and positively infinite time, respectively.[11] We discuss, in this section, applications of the special event values *beginning* and *forever*. We assume that all examples use the Gregorian calendar and the English language. That is, the value associated with the *beginning* property is the English word `beginning`, and the value associated with the *forever* property is the English word `forever`.

As a simple example of using forever, consider the following situation. If Beth is extremely content with her position in the company, then she may decide to remain with the company indefinitely. A value of `|forever|` is used to indicate this in her employment interval. A tuple for Beth is appended to the employee relation of Table 4. That tuple is shown in Table 6.

<table>
<tr><th>name</th><th>id</th><th>birthday</th><th>age</th><th>when_employed</th></tr>
<tr><td>Beth</td><td>55555</td><td>|9:30 PM 3/17/28|</td><td>%63 years%</td><td>[|6/1/90|, |forever|]</td></tr>
</table>

Table 6: An Example Using `forever`

As an example of using beginning, suppose that Beth has been an employee of the company since before employee records were accurately kept. This can be represented by the special event value `|beginning|`. The tuple for Beth is amended as shown in Table 7.

<table>
<tr><th>name</th><th>id</th><th>birthday</th><th>age</th><th>when_employed</th></tr>
<tr><td>Beth</td><td>55555</td><td>|9:30 PM 3/17/28|</td><td>%63 years%</td><td>[|beginning|, |forever|]</td></tr>
</table>

Table 7: An Example Using `beginning`

The special event values may be used to define intervals given the restrictions in Section 4.1.2. For example, the interval `interval(|beginning|,|forever|)` denotes the interval from negative infinity to positive infinity. Similarly, `interval(|beginning|,|beginning|)` represents a degenerate interval whose beginning and ending event values precede all other events, and the interval `interval(|forever|,|foreve` represents a degenerate interval whose beginning and ending event values follow all other events. The intervals `interval(`t`,|beginning|)` where $t \neq$ `|beginning|` and `interval(|forever|,`t`)` where $t \neq$ `|forever|` are illegal as they violate the constraint on interval bounds.

Lastly, we define a special zero span value. This value is needed since spans may be of positive or negative duration. The span constant `%0%` represents the span of zero duration.

4.5 Arithmetic Expressions

Arithmetic operations on temporal values are necessary in many computations. For example, one may wish to determine how many shopping days are left until Christmas or the arrival time of a train given its departure time and the duration of its trip.

In extending the basic arithmetic operators to the temporal data types, we have attempted to maximize the usability of the expressions through orthogonality and overloading while providing the temporal semantics that seem most intuitive. We have defined what we believe is the largest set of expressions possible given the semantics of our temporal data types. Expressions that are unintuitive or are not practically useful are disallowed.

Arithmetic involves the event, interval, span, and numeric data types. Arithmetic support for intervals is limited to simple displacement over a span of time. That is, for an interval value `interval(`t_{min}`,`t_{max}`)`

[11] This set of special event values is not complete. For example, the concept of *now*, which represents the constantly changing present, is another example of a special time value. It is not included in this discussion since it exhibits indeterminate semantics, and indeterminacy is beyond the scope of this paper.

and a span s, the result of `interval(`t_{min}`,`t_{max}`) + `s is `interval(`t_{min}` + `s`,`t_{max}` + `s`)`. Subtraction is similarly defined. A calendar may define additional functions that perform arithmetic calculations on interval parameters.

The four basic arithmetic operators (/, +, −, *) are overloaded for the temporal data types. /, *, and + are binary operators implementing the operations of division, multiplication, and addition, respectively. − implements binary subtraction in addition to span value negation, a unary operation. Table 8 shows the legal operations and operands. In the table, *span* represents a span, either fixed or variable, *event* is an event, and *numeric* is a numeric valued expression as defined in the SQL standard proposal.[12]

Operand 1	*Operator*	*Operand 2*	*Yields*
	−	*span*	*span*
span	+	*span*	*span*
span	−	*span*	*span*
event	+	*span*	*event*
event	−	*span*	*event*
span	+	*event*	*event*
event	−	*event*	*span*
span	*	*numeric*	*span*
numeric	*	*span*	*span*
span	/	*numeric*	*span*
span	/	*span*	*numeric*
span	+	*interval*	*interval*
interval	+	*span*	*interval*
interval	−	*span*	*interval*

Table 8: Valid Arithmetic Expressions and Results.

It is important to note that the operations are not orthogonal. For example, the expression *event* * *event* is undefined since reasonable semantics for the expression could not be determined. The semantic analyzer of the query processing system will detect undefined expressions through type checking. Appendix E lists all undefined expressions.

Also, operators may not obey the associative law of arithmetic. As an example of non-associativity, consider the following expression.

event − *span* − *event*

If parenthesized as follows then the expression is meaningful and evaluates to a span.

(*event* − *span*) − *event*

However, it is meaningless if parenthesized with the second subtraction given higher priority since *span* − *event* does not have intuitive semantics.

event − (*span* − *event*)

The programmer is responsible for ensuring that expressions are correctly parenthesized. Expressions that violate the typing rules described in Table 8 will, however, be detected during semantic analysis.

[12]The presentation in this section conforms with the definition of *datetime value expressions* as contained in the SQL2 proposal. Precedence and associativity are embedded implicitly in the syntax for arithmetic expressions (see Appendix B), and typing rules are specified here separately in a tabular format.

4.6 Comparison Expressions

One motivation for incorporating temporal values into the query language is to determine the temporal relationships between objects. For example, in the employee relation of Table 4, one might be interested in who was hired during a particular year, or given two employees, who has more seniority.

Defining temporal comparison operators allows one to determine these relationships. Allen defined a set of interval relations and showed that the set was complete in the sense that all possible relationships between two intervals were covered [Allen 1983].

In the interest of usability, we define an analogous set of operators for the event, interval and span data types. Our interval operator set was derived by examining the comparison operators of other temporal extensions to SQL [Ariav 1986, Ben-Zvi 1982, Martin et al. 1987, Navathe & Ahmed 1987, Navathe & Ahmed 1989, Sarda 1990A, Sarda 1990B, Sarda 1990C, Sarda 1990D]. Without decomposing intervals into events, we are able to directly simulate all of the operators that have been included in these proposals. We formally prove the completeness of the operator set by showing the operators are at least as powerful as Allen's with respect to interval comparisons, and we also show that all possible relations between other temporal types are covered.

Conversely, in the interest of minimality, we have attempted to extend the number of keywords and operators in the language by the smallest number possible. Therefore, whenever possible, we have overloaded the standard comparison operators. Comparison of span values is completely covered by the standard operators. In order to accommodate interval comparison we have introduced mnemonic comparison operators. These operators were then overloaded for interval-event, event-event, and event-interval comparisons.

4.6.1 Allen's Operators

Allen's operators for interval relationships and a pictorial representation of intervals for which the operators hold are shown in Figure 4 [Allen 1983]. (In order to distinguish Allen's operators from ours, we show Allen's operators in italic font throughout this section.)

For each relationship, except *equals*, its *inverse* relationship appears immediately below it. For example, *after* is the inverse of *before* since X *after* Y if and only if Y *before* X. The inverse of *equals* is not shown since equality is symmetric.

4.6.2 New Comparison Operators

Our proposed boolean operators are shown in Table 9. The relationships for span-span, event-event, and event-interval, and interval-event comparison should be clear. Let X and Y be intervals. We define the meanings of the interval-interval operators as follows.

- X `precedes` Y if and only if `end(`X`) < begin(`Y`)`.
- $X = Y$ if and only if `begin(`X`) = begin(`Y`) and end(`X`) = end(`Y`)`.
- X `meets` Y if and only if `end(`X`) = begin(`Y`)`.
- X `overlaps` Y if and only if there exists a chronon t such that $t \in X$ and $t \in Y$.
- X `contains` Y if and only if `(begin(`Y`) > begin(`X`) or begin(`Y`) = begin(`X`)) and (end(`Y`) < end(`X`) or end(`Y`) = end(`X`))`.

To show that the operator set is complete, we begin by noting that spans are comparable only to other spans and are incomparable to either events or intervals. The common arithmetic comparison operators are overloaded for span comparison.

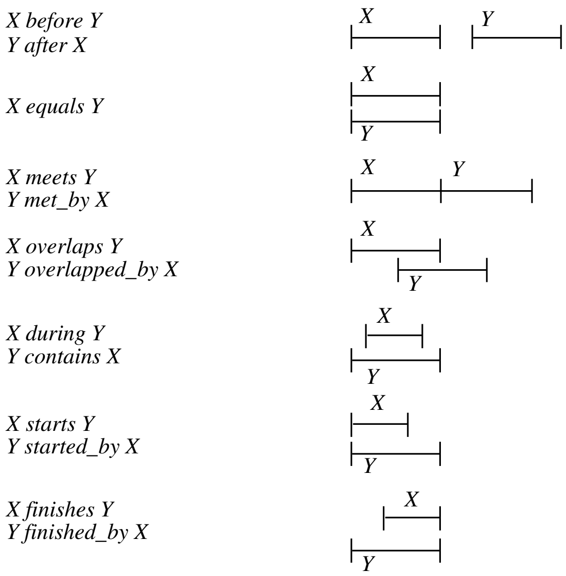

Figure 4: Complete Set of Interval Relations Defined by Allen

As can be seen from Figure 4, if an inverse relationship holds then the non-inverse version of the operator holds with the operands reversed. Therefore, what remains is to show that our operators subsume Allen's non-inverse relations.

Operand 1	*Operator*	*Operand 2*
span	=	*span*
span	<	*span*
span	>	*span*
event	=	*event*
event	`precedes`	*event*
event	`precedes`	*interval*
event	`overlaps`	*interval*
interval	`precedes`	*event*
interval	`overlaps`	*event*
interval	`precedes`	*interval*
interval	=	*interval*
interval	`meets`	*interval*
interval	`overlaps`	*interval*
interval	`contains`	*interval*

Table 9: Proposed Set of Comparison Operators

There are four cases to consider regarding the remaining data types. They are comparing events to events, comparing events to intervals, comparing intervals to events, and comparing intervals to intervals.

Case 1: Comparing events to events. Let e_1 and e_2 be events. Then e_1 can occur before e_2, e_1 can occur at the same time as e_2, or e_1 can occur after e_2. These subcases are handled by e_1 `precedes` e_2, $e_1 = e_2$, and e_2 `precedes` e_1, respectively.

Case 2: Comparing events to intervals. Let e be an event and *I* be an interval. Then e can occur before the beginning of *I*, e can occur sometime during *I*, or e can occur after the end of *I*. These subcases are handled by e `precedes` I, e `overlaps` I, and I `precedes` e, respectively.

Case 3: Comparing intervals to events. Let I be an interval and e be an event. Then the end of I can occur before e, I can occur coincidentally with e, or the beginning of I can occur after e. These subcases are handled by I `precedes` e, I `overlaps` e, and e `precedes` I, respectively.

Case 4: Comparing intervals to intervals. We show that the non-inverse relations defined by Allen are subsumed by our operators. The non-inverse relations are *before*, *equals*, *meets*, *overlaps*, *during*, *starts*, and *finishes*. We consider each in turn.

- *before* is identical to the `precedes` operator.
- *equals* is identical to the = operator.
- *meets* is identical to the `meets` operator.
- *X overlaps Y* can be implemented using `begin(`*X*`) < begin(`*Y*`) and end(`*X*`) < end(`*Y*`) and end(`*X*`) > begin(`*Y*`)` for intervals *X* and *Y*.
- *X during Y* can be implemented using *Y* `contains` *X* `and not` *Y* `=` *X* for intervals *X* and *Y*.
- *X starts Y* can be implemented using `begin(`*X*`) = begin(`*Y*`) and end(`*X*`) < end(`*Y*`)` for intervals *X* and *Y*.
- *X finishes Y* can be implemented using `begin(`*X*`) > begin(`*Y*`) and end(`*X*`) = end(`*Y*`)` for intervals *X* and *Y*.

This exhausts all cases, and so our set of temporal operators is complete.

It is interesting to note that we are able to directly simulate all but three of Allen's operators without decomposing the intervals into their bounding events. Two of these operators, *starts* and *finishes*, are not as commonly used as the others, and, furthermore, are not provided by other temporal SQL extensions. We feel, therefore, that not providing direct analogs is justified. For the third, we provide a generalized definition of interval overlap. That is, *X* `overlaps` *Y* is identical to *X overlaps Y* `or` *Y overlaps X*.

4.7 Aggregate Functions

The five SQL2 aggregate functions that operate on sets of values are shown in Table 10. An additional aggregate function `count(*)`, which determines the number of rows in a table, is not shown since it is not relevant to the discussion.

Aggregate Function	*Result*
`count`	Number of scalars in a column
`sum`	Sum of the scalars in a column
`avg`	Average of the scalars in a column
`max`	Largest scalar in a column
`min`	Smallest scalar in a column

Table 10: SQL2 Aggregate Functions

We extend the semantics of these functions to accommodate the temporal data types. As with arithmetic expressions, we identify all applications of these functions to the temporal types that have clear and intuitive semantics. For example, the sum of a set of span values is clearly defined while the sum of a set of interval

Aggregate Function	`event`	`interval`	`span`
`count`	•	•	•
`sum`			•
`avg`	•		•
`max`	•		•
`min`	•		•

Table 11: Aggregate Functions Applied to Temporal Types

```
select avg(e.age)
    from employee e
```

Figure 5: A Simple Example Using the `avg` Aggregate Function

values is not. Table 11 shows the aggregate functions and the temporal types to which they apply. A "•" indicates that the function is defined for that data type.

In Table 11, we have disallowed the `sum` aggregate to operate on a set of event values. This is consistent with our arithmetic algebra which disallows expressions such as *event* + *event*. However, we do allow the average to be computed over a set of events. The average of a set of events corresponds to a "midpoint" event computed with each event in the set given equal weight. While average is normally defined using addition and division, the aggregate `avg` over an event set has clear semantics and is allowed for event columns.

For a simple example of using a temporal aggregate function, we return to the employee relation of Table 4. Consider the query "What is average age of the employees in the company?" An SQL formulation of this query is shown in Figure 5. The average age is computed by using the `avg` aggregate function on the *age* attribute for each row in the table.

As a slightly more involved example, consider the query "What is the average length of service for people whose age is over the average age of all employees?" An SQL formulation of this query is shown in Figure 6.

In general, the rules for aggregate function use remain unchanged with the exception that they are now defined as given in Table 11 for the temporal data types.

4.8 Default Values

When the schema of a relation is defined in SQL, default values may be specified for columns of the relation. Attributes are set to the default values if no initial value is supplied by the user when a tuple is inserted into the relation. To maintain consistency with the language, we provide default capability for the temporal data types. For the time being, we ignore the closely related issue of null values. That topic will be addressed in the following section.

Default values are given by the `default` clause in a column definition. In general, SQL2 defaults can be one of three items: a literal value, `null`, or `user`. The `user` option returns a string value and so is not

```
select avg(span(e.when_employed))
    from employee e
    where e.age > (select avg(f.age) from employee f)
```

Figure 6: An Involved Example Using Aggregate Functions

```
create table employee (
    name character (20),
    id character (5),
    birthday event default null,
    age span default %0%,
    when_employed interval default interval(present, |forever|));
```

Figure 7: Setting Default Values for Temporal Columns

appropriate for initializing a temporal column. Therefore, we allow temporal default values to be any of the following.

- A temporal constant defined by a calendar.
- A built-in function that returns a value of the appropriate type.
- A `null` value.

For built-in functions supplying default values, we restrict any parameters passed to that function to be constant values.

For example, assuming that a tuple is inserted into the employee relation of Table 4 when an employee is hired, it may be appropriate to initialize the *birthday* attribute to `null` and the *age* attribute to zero if they are not known and the *when_employed* attribute to the interval from the present time to `|forever|`. The modified create statement is shown in Figure 7.

Default values for the non-temporal attributes *name* and *id* could also be specified, but they are not shown since *name* and *id* are non-temporal attributes.

The full syntax for default value specification is shown in Appendix B.19.

4.9 Null Values

An issue related to default values is the concept of *null values*. A null in SQL is used to represent *indeterminacy*, i.e., that an attribute has no applicable value or that a value exists but is unknown [Date 1989A].

In general, temporal data can exhibit both two kinds of indeterminacy, *value indeterminacy* and *historical indeterminacy* [Dyreson & Snodgrass 1992]. Value indeterminacy covers the two types of indeterminacy denoted by `null` and mentioned above. Temporal indeterminacy deals with the uncertainty of when occurrences happened in time. SQL support for temporal indeterminacy is being investigated separately and will not be addressed here. This section concentrates on supporting value indeterminacy for temporal attributes in SQL.

As an example of a value indeterminate temporal attribute, consider the situation where a temporary employee is hired, but the period when he or she was employed is forgotten before the tuple can be entered into the table. Then the *when_employed* attribute for the tuple is value indeterminate. This is denoted by a `null` as in Table 12.

name	*id*	*birthday*	*age*	*when_employed*
`Clarence`	`66666`	`|7:30 AM 3/17/25|`	`%66 years%`	`null`

Table 12: An Example of Unknown Value Indeterminacy

In Table 12, the `null` value is needed because one does not know what the correct value is for the attribute. That is, one *knows* that a correct value exists but is uncertain of its value. This contrasts with the situation where a value for the attribute is known not to exist, that is, that the attribute has no applicable value. As a whimsical example, suppose that employee Clarence was not born, but instead that he was hatched from an egg. Then the *birthday* attribute for Clarence's tuple would reflect this with a `null` as in Table 13. (We assume that Clarence's interval of employment is still unknown, and so the *when_employed* attribute in Clarence's tuple has the value `null`.)

name	*id*	*birthday*	*age*	*when_employed*
`Clarence`	`66666`	`null`	`%66 years%`	`null`

Table 13: An Example of Inapplicable Value Indeterminacy

The fact that both unknown value and inapplicable value indeterminacy are represented by a single type of null value is an unfortunate characteristic of SQL. This example is one of many well-documented shortcomings of SQL null values [Date 1989A]. In extending null values to the temporal data types, we have elected to retain these semantics since tampering with them would require an entire redesign of nulls in SQL, and that would greatly impact the language.

Therefore, we adopt the following rules regarding null values [Date 1989A].

- In temporal arithmetic expressions of the form A *op* B if either A or B is `null` then the result of the expression is `null`.
- In temporal comparison expressions of the form A *op* B if either A or B is `null` then the result of the expression is `null`.
- In `where` and `having` clauses nothing, including a `null`, is equivalent to a `null`.
- In computing `distinct`, `order_by` and `group_by` clauses a `null` is considered equivalent to other `null`s.
- A temporal built-in function receiving a `null` as a parameter returns `null`.
- Scalar aggregate functions, except `count`, over temporal domains return `null` when operating on the empty set. The aggregate function `count` given an empty set returns zero.

4.10 Summary

We have augmented SQL with language features supporting temporal constants. The proposal includes new temporal data types, calendric system and property table selection constructs, temporal built-in functions, special time values, temporal arithmetic and comparison, aggregate functions, and support for value indeterminacy. As an illustration of the material, we will show, in the following section, how the semantics of a particular calendric sytem could be implemented within our proposal.

5 An Example Calendric System

To illustrate the proposal, we will show how a particular calendric system can be defined using our proposal. The example we use is the calendric system defined in the proposed SQL2 standard [Melton 1990]. This calendric system consists of a single calendar, the Gregorian calendar, defined over the epoch 1 A.D. to 9999 A.D. We show how equivalent semantics can be implemented in our proposal using a new calendar definition and the calendar-independent language constructs. This section assumes that the reader is familiar with the date and time support being proposed for SQL2.

```
declare calendric system as gregorian_sql

select e.birthday, e.age
    into (bday, age with event_yearmonth)
    from employee e
    where e.name = 'Dawn'
```

Figure 8: Example of Simulating the SQL2 Calendric System

5.1 Data Types

The two notions of time provided by SQL2, *datetime* and *interval*, roughly correspond to our data types of `event` and `span`, respectively. The three datetime data types, *date*, *time*, and *timestamp* can be simulated with `events` and the proper display formats. Separate output format properties can be constructed to display events as dates, times, and timestamps. One of these output formats can be selected as the session-default. The remainder can be specified on a per-item basis. Similar remarks apply to the two classes of intervals, *year-month* intervals and *day-time* intervals.

For example, in the employee relation of Table 4, the *birthday* attribute could be considered either a timestamp if the time of birth is known (and relevant), or a date if not. Assuming the existence of a calendric system, `gregorian_sql2`, which returns events in SQL2 timestamp format, and a table `event_yearmonth` which contains year-month format properties, the query in Figure 8 displays for employee Dawn her *birthday* and *age* in timestamp and year-month formats, respectively.

The `declare calendric system` statement selects the `gregorian_sql` calendric system as the session-default. Since timestamp format is the default for event retrieval, the *bday* attribute is returned as an SQL2 timestamp. The `event_yearmonth` identifier names a table containing SQL2 yearmonth formats for input and output. The calendric system is directed to this table to find the appropriate output format.

Implementing day-time intervals is similar to the above. However, the issue of why SQL2 provides two incomparable span data types, year-month intervals and day-time intervals, raises interesting questions. We will investigate these questions in Section 6.2.4.

5.2 Time Zones

SQL2 provides the notion of *timezone*. Timezones partition the surface of the earth into regions where the current time is the same everywhere within each region. The time within a region is known as the *local time*.

Two facets of timezones are supported by SQL2. A time or timestamp value may be converted to *Universal Coordinated Time* (UTC) before storage in a database, and relative time calculations may be performed on time values in different timezones.

Within our scheme, conversion between timezones can be accomplished by defining explicit conversion functions in the calendar. For example, to simulate the storage of values in UTC, one could define a function `at(`*time_value*`,` tz_1`,` tz_2`)` which converts *time_value*, assumed to be a time value in timezone tz_1, to the simultaneous time in timezone tz_2. The function `at` would be defined as an intrinsic component of the calendar. It would be explicitly called before a time value is stored in the database.

This mechanism allows the general conversion of time values between any timezones, and the calculation of expressions containing time values relative to several timezones. In such expressions, values in different timezones can be converted to a common timezone, like UTC, before performing any computations. For example, suppose a train leaves Los Angeles at 9:00 AM Pacific Standard Time on Christmas Day

1991 and arrives in Phoenix at 6:00 PM Mountain Standard Time the same day. To compute the amount of time the train spent in transit the following computation could be used. (It is assumed that `'mst'`, `'pst'`, and `'utc'` are string values defined by the calendar which represent the given timezones. This could be accomplished by maintaining a timezone table with the schema (timezone:*string*, displacement:*span*).)

```
at(|6:00 PM 12/25/91|, 'mst', 'utc') - at(|9:00 AM 12/25/91|, 'pst',
                                       'utc')
```

Both the departure and arrival times are translated from their current timezones to UTC. The elapsed time is then calculated using the UTC times.

5.3 Built-in Functions

The built-in functions defined in the SQL2 proposal are shown in Table 14.

Built-in Function	*Explanation*
`current_date`	Return the current date in the local timezone
`current_time`	Return the current time in the local timezone
`current_utc_time`	Return the current time in UTC
`current_timestamp`	Return the current timestamp in the local timezone
`current_utc_timestamp`	Return the current timestamp in UTC

Table 14: SQL2 Built-in Functions for Time

We have not discussed the physical representation of time values other than the general considerations mentioned in Section 3.1. The functions `current_utc_time` and `current_utc_timestamp` can be implemented by defining the mapping from the internal time representation to UTC. This mapping is well-defined since the internal time line and UTC are well-defined. The remaining functions, `current_date`, `current_time`, and `current_timestamp`, can be implemented by translating between UTC and the local timezone using the value returned by the appropriate UTC function. For example, assuming that the `current_utc_timestamp` function is implemented, the function `current_timestamp` can be computed with the following expression.

```
current_utc_timestamp + at(|12:00 AM|, 'utc', 'local')
```

Another type of function defined in the SQL2 proposal allows the extraction of fields from time values. We also allow the definition of such functions in calendric systems. Calendar defined functions may return either string or numeric values. For example, the SQL2 extract function `extract(month from birthday)` could be implemented with a calendar defined function `month_number(`*event*`)` which when invoked returns a numeric value representing the month component of the *event*. Likewise, a function `month_name(`*event*`)` can be defined which returns a string containing the month component of the *event*. Similar remarks apply to the `span` data type.

5.4 Arithmetic Expressions

The legal arithmetic expressions on the SQL2 datetime and interval data types are shown in Table 15. Next to each SQL2 arithmetic expression is an equivalent expression using our data types.

As can be seen from Table 15, any legal arithmetic expression using the SQL2 data types is also a legal arithmetic expression in our proposal when the corresponding data types are used. Therefore, any temporal calculation that can be performed using the SQL2 proposal can also be performed in our proposal.

SQL2 Expression	*Equivalent Expression*
interval ← *datetime* − *datetime*	*span* ← *event* − *event*
datetime ← *datetime* + *interval*	*event* ← *event* + *span*
datetime ← *datetime* − *interval*	*event* ← *event* − *span*
interval← *interval* + *interval*	*span* ← *span* + *span*
interval ← *interval* − *interval*	*span* ← *span* − *span*
interval ← *interval* * *numeric*	*span* ← *span* * *numeric*
interval ← *interval* / *numeric*	*span* ← *span* / *numeric*

Table 15: SQL2 Arithmetic Expressions and Equivalents

5.5 Comparison Expressions

We showed in Section 4.6 that all possible relationships between our temporal data types were expressible using the comparison operators defined in this paper. Since our temporal types are a superset of the temporal types in the SQL2 proposal, it follows that the our comparison operators subsume the SQL2 comparison operators.

5.6 Default Values

The built-in functions shown in Table 14 may be used to supply default values in SQL2. These functions are zero argument functions which return either the current date, time, or timestamp. Only event-like columns may be initialized with built-in functions.

We have generalized this by allowing arbitrary built-in functions to supply default values to event, interval, or span columns. For simplicity, however, we limit the parameters that can be passed to such built-in functions to be constant values.

5.7 Null Values

As mentioned in Section4.9, we have adopted the null value scheme in SQL2 without modification.

5.8 Summary

This section has demonstrated an application of the material in Section 4. We have shown how the temporal support contained in the SQL2 proposal could be implemented with our mechanisms. In doing so, we have shown how specific calendar semantics can be accommodated within a calendric system independent framework. Appendix C contains a concise summary of the material in this section.

6 Summary and Evaluation

To summarize the contributions of this paper, we contrast it with two other proposals for temporal constant support in SQL, that of Date [Date 1988] and that contained in the proposed SQL2 standard [Melton 1990]. We choose these proposals for comparison since our work, in many ways, generalize and refine the concepts they develop.

The discussion is framed in terms of the three design goals which guided this project, generality, usability, and minimality. We will discuss how our proposal addresses each of these issues, and we will examine the extent to which the other two proposals provide similar functionality.

6.1 Generality

The time support provided by both the Date and SQL2 proposals is limited to a single calendar, the Gregorian calendar. We have generalized this by allowing any well-defined calendar to be supported by the database management system.

Separation of calendars from the database management system allows the format of displayed time values to be controlled locally rather than by the designers of the database management system. The special requirements of the time domain, as discussed in Section 3, are now supported at the local site where they can be adapted to local custom, political considerations, language, and so forth.

As a concrete example, consider the implementation of variable spans. In the Gregorian calendar, the variable span `%1 month%` represents a number of days which varies between twenty-eight and thirty-one. In Date's proposal, the semantics of `%1 month%` with respect to arithmetic is defined by a built-in function, *add_month*, which is part of the query language proper. Adding `%1 month%` to the last day of a month always computes the last day of the next month regardless of the number of days in each. While this is a reasonable interpretation, there are many other interpretations that are equally valid. For instance, in a payroll application, a *month* may be the accounting term which defines a fixed pay period. This definition of *month* has no relation to a calendar month, and so Date's proposal, which has a predefined notion of month, falls short. Of course, within Date's proposal, one could define an interval containing the number of days in a *month*, but we feel that time objects such as these are basic to a calendar and should be supported directly by the database management system.

6.2 Usability

To evaluate the usability of our proposal, we will examine all three proposals relative to the functionality described in Section 4. We begin by considering the data types provided by each proposal.

6.2.1 Data Types

Table 16 shows for each of the temporal types we propose, an analogous type, if one exists, that is provided by one of the other two proposals.

Data Type	*Date's Proposal*	*SQL2 Proposal*
`event`	`interval`	`datetime`
`interval`		
`span`	`interval`	`interval`

Table 16: Analogous Types in Date and SQL2 Proposals

As can be seen from Table 16, Date's concept of an interval corresponds to both the `event` and `span` types, and SQL2's datetime and interval concepts are analogous to `events` and `spans`, respectively. No data type provided by either Date or SQL2 is similar to our `interval` data type though SQL2 does provides limited support for interval-like comparison (see Section 6.2.5). Our set of data types can be considered to be a superset of the types provided by Date and SQL2, though with refined semantics.

We feel that intervals, as defined in Section 4.1.2, are a fundamental concept of time and so should be included in the set of temporal types [Allen 1983]. As stated in Section 4.1, our desire was to create a set of types that capture the intuitive and familiar aspects of time, without unduly impacting the language. We believe that adding intervals to the two concepts of time provided by Date and SQL2 provides such a set.

6.2.2 Built-in Functions

Each of the three proposals define built-in functions for the temporal data types. SQL2 defines functions that return the current date, time, and timestamp, in addition to field extraction functions (`extract` (*field* `from` *column*)). Date provides similar facilities while adding functions that convert temporal data to numeric and string data. Our proposal defines built-in functions for returning the present time and constructing and destructing temporal data, and we provide the means for calendars to define extraction functions and, also, conversion functions to and from non-temporal data types.

We have abstracted the best qualities of the Date and the SQL2 proposals and then applied them in a calendar independent context. The result allows more generality than either Date's or SQL2's scheme, and, moreover, the facilities are in place for additional functions to be defined, if so desired, by a calendar.

6.2.3 Special Time Values

Concepts analogous to the special time values *beginning* and *forever* are not found in either Date's proposal or the SQL2 proposal. The utility of these notions was discussed in Section 4.4.

6.2.4 Arithmetic Expressions

The difference between the three proposals becomes most evident when discussing the semantics of arithmetic expressions.

By combining the notions of event and span into a single data type, Date is able to achieve a high degree of orthogonality in arithmetic expressions. Similarly, we attempted to exploit orthogonality whenever possible, but given our wider set of temporal types the amount of orthogonality among our expressions is not as great. Appendix E shows the expressions that are disallowed in our proposal.

While a higher degree of orthogonality would be desirable, we do not necessarily feel that this is a poor aspect of the design. Two reasons support this viewpoint. The first is that many of the expressions shown in Appendix E have unintuitive semantics and so should be disallowed. The second reason is that what is lost in orthogonality is compensated by the semantic richness of the three temporal types. The expressiveness of our data types allow temporal ideas to be clearly conveyed. We strongly believe that separate types should be kept separate.

The arithmetic semantics provided by the SQL2 proposal are unclear, and the cause of this confusion provides more support for our general approach. In the SQL2 proposal, two distinct types of *interval*, which corresponds to our notion of span, are defined. They are *year-month interval*s, which consist of a years part and a months part, and *day-time interval*s, which consist of a days, an hours, a minutes and a seconds parts. Year-month intervals and day-time interval are incomparable and cannot not be combined in either arithmetic or comparison expressions. The reason for this dichotomy is to avoid the fact that there are a variable number of days in both months and years. That is, since identical year-month intervals could possibly represent different time spans depending on the presence of leap days or which months are being represented, SQL2 skirts the issue by not regarding year-month intervals as a span of days at all.

A moment of reflection reveals that a year-month interval is an example of what we have defined as a variable span. By defining the semantics of variable spans in the calendar, we are able to avoid this shortcoming of SQL2. The meaning of a year-month interval in terms of some number of days can be defined unambiguously by the calendar. A consequence of this fact is that our design is more expressive than either the SQL2 proposal or Date's proposal. Both of their calendric schemes can be implemented in our proposal (see Section 5 for a discussion of the SQL2 calendric system), and we can construct expressions with semantics that can not be implemented in theirs.

6.2.5 Boolean Expressions

In analyzing our temporal comparison operators, we are drawn to many of the same conclusions as in the previous analysis of arithmetic operators. The richness of our temporal types and, in particular, the addition of an interval data type allows us to more clearly express temporal relationships than either the SQL2 proposal or the Date proposal.

We have shown that the temporal comparison operators defined in Section 4.6 are able to express the complete set of relationships between any two temporal objects. This is an extension of the operators proposed by Date and SQL2. Both provide comparison between their event-like and span-like objects by overloading the common scalar comparison operators. We have adopted this approach whenever possible. That is, we have defined analogous comparison operators for span-span and event-event comparison.

Regarding interval-interval comparison, both the SQL2 proposal and this paper define support for interval-interval predicates. The SQL2 proposal defines an `overlaps` operator which operates on interval-like objects. This predicate considers adjacent datetime-datetime or datetime-interval column pairs to be interval-like objects. Comparison is made to determine if two rows contain temporally overlapping pairs of columns. The `overlaps` predicate is the only operator defined on interval-like column pairs.

Our approach extends the interval-interval comparison support by directly incorporating the notion of interval into the type system and adding predicates that can completely determine all possible relationships between two intervals. We believe that completeness is necessary for usability and expressive power.

Lastly, interval-event and event-interval comparison is addressed only by our proposal though it would be a small extension of the SQL2 `overlaps` predicate.

6.2.6 Aggregate Functions

Table 17 shows the aggregate functions and the SQL2 data types on which they operate. Table 18 shows the same information for Date's proposal.

Aggregate Function	`datetime`	`interval`
`count`	•	•
`sum`		•
`avg`		•
`max`	•	•
`min`	•	•

Table 17: Aggregate Functions Applied to SQL2's Data Types

Aggregate Function	`interval`
`count`	
`sum`	
`avg`	
`max`	•
`min`	•

Table 18: Aggregate Functions in Date's Proposal

A quick comparison of Table 17 and Table 18 to Table 11 shows that our proposal has defined a superset of Date's and SQL2's aggregate functions. There is little to say other than that the function applications shown in Table 11 have clearly defined semantics, and so, they should be permitted by the query language.

6.2.7 Default Values and Null Values

Both default and null values (for value indeterminacy) were added to the proposal to maintain conformity with the remainder of the language. That is, we have adopted exactly the default column initialization and null value logic of SQL2. While the null value mechanism in SQL has some well-known semantic inconsistencies [Date 1989A], we have elected to incorporate it without modification since any attempt to change it would have tremendous impact on the language overall.

6.3 Minimality

The last design criteria, minimality, is, in general, independent of the other proposals. We have attempted to make the language modifications easily adoptable by having minimal impact on the language as a whole. A number of language design techniques have been used to accomplish this. These include being parsimonious in the number of new keywords introduced, reusing existing operator symbols and overloading new ones whenever possible, and attempting to maximize the orthogonality of constructs in the language.

We believe that we have been reasonably successful at meeting this goal while still producing a language with high expressive power and usability. As evidence, Appendix B.3 shows that by replacing the time support currently proposed for SQL2 with our proposal, the number of keywords in the language actually decreases. One reason for the decrease is that calendar-specific keywords are removed from the language. The complexity of the language has decreased since its size has decreased, but, as shown above, the expressive power of the language has increased.

7 Future Work

This work provides a complete language design for temporal constants in SQL. We have defined a set of temporal data types along with built-in functions, special time values, arithmetic operators, comparison operators, and aggregate functions. The design has been developed in a calendar independent framework, and facilities have been described for supporting multiple calendric systems within the query language.

There are, however, additional user level topics that complement this design, and which are beyond the scope of this paper. We discuss some of these below. In addition, we discuss plans for implementing both this proposal and some extensions.

Further research can be divided into two categories, extensions for temporal constants and support for valid and transaction time semantics. We divide the discussion along these guidelines.

7.1 Temporal Constants

As mentioned in Section 2, this work is the first step in an on-going project to augment SQL with temporal support. On the user level, our next steps is to augment SQL with language constructs supporting historical indeterminacy [Dyreson & Snodgrass 1992]. Allowing historical indeterminacy will increase the usability of the language by extending predicates to events whose precise occurrence may not be known. This relaxes the rather strict assumption in Section 4.1 that events must be known to the granularity of a single chronon. Also, we believe that some special time values have indeterminate qualities. An example would be the value *now* which refers to the present moment in time. Since the present time is ever changing, the value of now is inherently indeterminate. Issues such as these are being investigated in the context of indeterminacy.

Regarding implementation, we have prototyped a calendric system for the Gregorian calendar. We plan to build an SQL query processor that parses the described language constructs and to integrate this query processor with the calendric system prototype [Soo & Snodgrass 1992, Soo et al. 1992]. The interface between the query processor and calendric system must be specified and implemented.

7.2 Valid and Transaction Time

Snodgrass and Ahn have identified three orthogonal aspects of database time, *valid time* (the time in the real-world that is being modeled), *transaction time* (the time that changes are made to the database), and *user-defined time* (any time domain which is uninterpreted by the database management system) [Snodgrass & Ahn 1986]. Temporal constants can be used to express time values in each of these aspects of time. As such, a well-designed user model for temporal constants provides the foundation for further work in valid and transaction time support, as well as being a full and concrete proposal for user-defined time.

A formal semantics for SQL and the language modifications is needed to to ensure a consistent and viable design. A prototype implementation will then be developed to empirically determine the usefulness of the modifications and to explore efficiency issues related to the use of the temporal constructs.

8 Acknowledgements

The authors thank Curtis Dyreson, Suchen Hsu, Christian S. Jensen and Nick Kline for many enlightening discussions and their insightful comments on earlier drafts of this paper. Funding for this work was provided in part by the National Science Foundation through Grant IRI-8902707 and by the IBM Corporation through Contract #1124.

9 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.

[Anderson 1982] Anderson, T. L. "Modeling Time at the Conceptual Level," in *Proceedings of the International Conference on Databases: Improving Usability and Responsiveness*. Ed. P. Scheuermann. Jerusalem, Israel: Academic Press, June 1982, pp. 273–297.

[Anderson 1983] Anderson, T. L. "Modeling Events and Processes at the Conceptual Level," in *Proceedings of the Second International Conference on Databases*. Ed. S.M. Deen and P. Hammersley. The British Computer Society. Cambridge, Great Britain: Wiley Heyden Ltd., 1983.

[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.

[Bolour et al. 1982] Bolour, A., T. L. Anderson, L. J. Dekeyser and H. K. T. Wong. "The Role of Time in Information Processing: A Survey." *SigArt Newsletter*, 80, Apr. 1982, pp. 28–48.

[Clifford & Rao 1987] Clifford, J. and A. Rao. "A Simple, General Structure for Temporal Domains," in *Proceedings of the Conference on Temporal Aspects in Information Systems*. AFCET. France: May 1987, pp. 23–30.

[Digital 1991] DEC "Digital Guide to Developing International Software." Digital Press, 1991.

[Date 1989A] Date, C. J. “A Guide to the SQL Standard (Second Edition).” Addison-Wesley, 1989.

[Date 1989B] Date, C. J. “An Overview of SQL2.” *Info. Database*, 4, No. 1, Spring 1989, pp. 8–12.

[Date & White 1990] Date, C. J. and C. J. White. “A Guide to DB2.” Reading, MA: Addison-Wesley, 1990. Vol. 1, 3rd edition.

[Date 1988] Date, C.J. “A Proposal for Adding Date and Time Support to SQL.” *SIGMOD Record*, 17, No. 2, June 1988, pp. 53–76.

[Dyreson & Snodgrass 1991] Dyreson, C. E. and R. T. Snodgrass. “Temporal Indeterminacy.” Technical Report TR 91-30. Computer Science Department, University of Arizona. Dec. 1991.

[Dyreson & Snodgrass 1992] Dyreson, C. E. and R. T. Snodgrass. “Historical Indeterminacy.” Technical Report TR 91-30a. Computer Science Department, University of Arizona. Revised Feb. 1992.

[Keretho & Loganantharaj 1990A] Keretho, S. and R. Loganantharaj. “An Interval Arithmetic for Temporal Distance Constraint Satisfaction Problems.” Technical Report CACS TR-90-2-2. University of Southwestern Louisiana. Apr. 1990.

[Keretho & Loganantharaj 1990B] Keretho, S. and R. Loganantharaj. “An Extended Time Point Algebra.” Technical Report CACS TR-90-2-3. University of Southwestern Louisiana. June 1990.

[Keretho & Loganantharaj 1990C] Keretho, S. and R. Loganantharaj. “Qualitative and Quantitative Time Interval Constraint Networks.” Technical Report CACS TR-90-2-4. University of Southwestern Louisiana. July 1990.

[Ligozat & Bestougeff 1989] Ligozat, G. and H. Bestougeff. “On Relations Between Intervals.” *Information Processing Letters*, 32, No. 4, Sep. 1989, pp. 177–182.

[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.

[McKenzie 1986] McKenzie, E. “Bibliography: Temporal Databases.” *ACM SIGMOD Record*, 15, No. 4, Dec. 1986, pp. 40–52.

[Melton 1990] Melton, J. (ed.) “Solicitation of Comments: Database Language SQL2.” American National Standards Institute, Washington, DC, 1990.

[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.

[Oracle 1987] Oracle Computer, Inc. "ORACLE Terminal User's Guide." Oracle Corporation, 1987.

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

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

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

[Snodgrass 1990] Snodgrass, R. "Temporal Databases: Status and Research Directions." *ACM SIGMOD Record*, 19, No. 4, Dec. 1990, pp. 83–89.

[Snodgrass & Ahn 1986] Snodgrass, R. T. and I. Ahn. "Temporal Databases." *IEEE Computer*, 19, No. 9, Sep. 1986, pp. 35–42.

[Soo 1991] Soo, M. D. "Bibliography on Temporal Databases." *ACM SIGMOD Record*, 20, No. 1, Mar. 1991, pp. 14–23.

[Soo & Snodgrass 1992] Soo, M. D. and R. Snodgrass. "Multiple Calendar Support for Conventional Database Management Systems." Technical Report 92-7. Computer Science Department, University of Arizona. Feb. 1992.

[Soo et al. 1992] Soo, M. D., R. Snodgrass, C. Dyreson, C. S. Jensen and N. Kline. "Architectural Extensions to Support Multiple Calendars." TempIS Technical Report 32. Computer Science Department, University of Arizona. Revised May 1992.

[Stam & Snodgrass 1988] Stam, R. and R. Snodgrass. "A Bibliography on Temporal Databases." *Database Engineering*, 7, No. 4, Dec. 1988, pp. 231–239.

[Tandem 1983] Tandem Computers, Inc. "ENFORM Reference Manual." Cupertino, CA, 1983.

[Vilain 1982] Vilain, M. B. "A System for Reasoning About Time," in *Proceedings of the American Association for Artificial Intelligence*. Pittsburgh: Aug. 1982, pp. 221–226.

[Vilain & Kautz 1986] Vilain, M. B. and H. Kautz. "Constraint Propagation Algorithms for Temporal Reasoning," in *Proceedings of the American Association for Artificial Intelligence*. 1986, pp. 377–382.

A Glossary of Terms

New terms introduced in this paper are collected and defined below.

beginning The event denoting negatively infinite time; an event that precedes all other events.

calendar A device by which the passage of time is measured and recorded.

calendric system A collection of calendars that collectively define the time values meaningful to an enterprise.

chronon A discrete, indivisible unit of time defined at the implementation level.

default property table The relation defining default properties for an installation.

event An isolated instant in time constrained to occur during some chronon.

extrinsic characteristic A calendar quality that is subject to change depending on the user environment. A synonym for "property."

fixed span A span whose duration is fixed and independent of context or usage.

forever The event denoting positively infinite time; an event that follows all other events.

interval A period of time delimited by two events. Intervals by definition have definite starting and ending times and a positive duration.

intrinsic characteristic A calendric quality dealing with or determining the inherent semantics of that calendar.

property A characteristic of a calendric system that is subject to local adaptation. A synonym for "extrinsic characteristic."

property table A relation used to define the properties of a calendar.

span A duration of time. Spans may be either positive or negative in duration, and, unlike intervals, do not have definite starting and ending times.

variable span A span whose duration is determined by its usage or the context in which it appears.

B Modified Language Syntax

The organization of this appendix follows that of the SQL2 proposal document [Melton 1990]. 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. Section 4 serves as a reference for these additional rules. We assume that the reader is familiar with the SQL2 proposal, and that a copy of the proposal is available for reference.

B.1 Section 5.2 <datetime qualifier>

The <datetime qualifier> non-terminal and its subtree are deleted. This is a calendar-specific language construct and definition of such would appear in a calendar.

B.2 Section 5.3 <literal>

The <datetime literal> non-terminal and its entire subtree are deleted. This is a calendar-specific language construct and definition of such would appear in a calendar.

The <interval literal> non-terminal and its entire subtree are deleted. The <interval literal> non-terminal which defined time durations is effectively replaced with a new non-terminal <span literal>.

The production for the non-terminal <general literal> is replaced with the following.

```
<general literal> ::=
            <character string literal>
          | <national character string literal>
          | <bit string literal>
          | <hex string literal>
•         | <event literal>
•         | <interval literal>
•         | <span literal>
```

The following productions are added.

<event literal> ::=
- [<introducer> <character set specification>]
  <vertical bar> <event value> <vertical bar> [<calendric-property specification>]

<interval literal> ::=
- [<introducer> <character set specification>]
  <left bracket> <interval value> <right bracket> [<calendric-property specification>]

<span literal> ::=
- [<introducer> <character set specification>]
  <percent> <span value> <percent> [<calendric-property specification>]

<event value> ::=
- !! *See the syntax rules*

<interval value> ::=
- !! *See the syntax rules*

<span value> ::=
- !! *See the syntax rules*

Format-related property values describe the contents of temporal constants. The BNF grammar for format strings is as follows.

<format string> ::=
- <quote> [<background character> | <field specification>]... <quote>

<background character> ::=
- <character representation>

<field specification> ::=
- <less than operator> <field identifier> <comma> <translation table name>
  [<comma> <field formatting specification>] <greater than operator>

<field identifier> ::=
- <identifier>

<translation table name> ::=
- <identifier>

<field formatting specification> ::=
- !! *See the syntax rules*

Additional syntax rules:

1. An <event value> is any sequence of characters not containing a single <vertical bar>. The format of an <event value> is described by the current value of the `event_input_format` property.
2. Within an <event value> a <vertical bar> is represented by <vertical bar><vertical bar>.
3. An <interval value> is any sequence of characters not containing a single <right bracket>. The format of an <interval value> is described by the value of the `interval_input_format` property.
4. Within an <interval value> a single <right bracket> is represented as <right bracket><right bracket>.
5. A <span value> is any sequence of characters not containing a single <percent>. The format of a <span value> is described by the current value of the `span_input_format` property.
6. Within a <span value> a single <percent> is represented as <percent><percent>.
7. The data type of an <event literal> is `event`.
8. The data type of an <interval literal> is `interval`.
9. The data type of a <span literal> is `span`.
10. The non-terminal <calendric-property specification> is defined in Appendix B.21.
11. If <calendric-property specification> is specified then the <event literal>, <interval literal>, or <span literal> is interpreted as a temporal value.
12. If <calendric-property specification> contains an <as clause> then the calendric system named in the <as clause> is used when interpreting this literal. Otherwise, the globally declared calendric system whose scope includes this literal is used.
13. If <calendric-property specification> contains a <with clause> then the properties contained in the named property table are activated before interpreting this literal, and deactivated after interpreting this literal.
14. If no `declare calendric system` command has been entered then the implementation defined default calendric system is assumed.
15. A <format string> defines the syntax of strings specified as the values of format properties in property tables.
16. A <field specification> specifies a translation table and a formatting specification for a particular field value.

17. A <field formatting specification> specifies how translated values of a particular data field, e.g., *day*, is to appear in the output. Valid format specifiers are as follows.

    - w*num*—place the value in an output field of width *num*. The default field width is just large enough to contain the constant and a sign if specified. Truncation will occur on the right if the value is too large, and the field is left-justified. Truncation will occur on the left is the field is too large, and the field is right justified. Only one `w` specification is permitted for each <field formatting specification>.
    - `l`—place the value left-justified in the field. Cannot be specified with `r`.
    - `r`—place the value right-justified in the field. Right justification is the default. Cannot be specified with `l`.
    - `z`—pad the field with zeros. Cannot be specified with `b`.
    - `b`—pad the field with blanks. Blank padding is the default. Cannot be specified with `z`.
    - `s`—include a sign character in the output. For negative numeric values the sign is always displayed. `s` forces a positive sign for positive numeric values. Cannot be specified for non-numeric data.

18. Within a <format string>, <less than operator><less than operator> denotes a single <less than operator>.

19. Within a <format string>, <quote><quote> (that is, a <quote symbol>) denotes a single <quote>.

20. Any <background character> appearing in the format string appears in an output string in the same relative position and order with respect to other <background character>s and <field specification>s.

B.3 Section 5.4 <token>

The production for the non-terminal <nondelimiter token> is replaced, and a new non-terminal <calendar defined identifier> is added to define the format of calendric system defined identifiers. Calendric system defined identifiers are prefixed with an ampersand (“`&`”).

```
<nondelimiter token> ::=
          <regular identifier>
        | <key word>
        | <unsigned numeric literal>
        | <hex string literal>
•       | <event literal>
•       | <interval literal>
•       | <span literal>

<calendar defined identifier> ::=
•         <ampersand> <identifier>
```

The production for the non-terminal <key word> is changed. The keywords `current`, `current_date`, `current_time`, `current_timestamp`, `current_utc_time`, `current_utc_timestamp`, `date`, `day`, `extract`, `hour`, `local`, `minute`, `month`, `second`, `time`, `timestamp`, `utc`, `year`, and `zone` are deleted.

The following productions are added for the non-terminal <key word>. To conserve space, we do not copy the existing keyword definitions from the SQL2 proposal.

<key word> ::=
- `calendric`
- | `contains`
- | `event`
- | `meets`
- | `precedes`
- | `present`
- | `previous`
- | `properties`
- | `span`
- | `general`

Nineteen keywords are deleted and ten are added—the number of keywords is reduced by nine.

B.4 Section 5.5 <names>

The following productions are added.

<calendric system name> ::=
- <identifier>

<property table name> ::=
- <table name>

Additional syntax rules:

1. The identifiers denoting calendric systems and property tables are implementation dependent.

B.5 Section 5.6 <data type>

The production for the non-terminal <data type> is replaced with the following.

```
<data type> ::=
          <character string type> [ character set <character set specification> ]
        | <national character string type>
        | <bit string type>
        | <numeric type>
•       | <event type>
•       | <interval type>
•       | <span type>
```

The <datetime type> non-terminal and its entire subtree are deleted. This is a calendar specific language construct.

The production for the <interval type> non-terminal is replaced, and new productions for the non-terminals <event type> and <span type> are added as follows.

```
<event type> ::=
•         event [ <open paren> general <close paren> ]

<interval type> ::=
•         interval [ <open paren> general [ <comma> general ] <close paren> ]
•         interval [ <open paren> [ general ] <comma> general ] <close paren> ]

<span type> ::=
•         span [ <open paren> general <close paren> ]
```

Additional syntax rules:

1. Specifying `general` allocates an extended range, fine granularity time-stamp.[13]
2. Not specifying `general` allocates an extended range, coarse granularity or narrow range, fine granularity time-stamp.
3. For an <interval type> the generality of the beginning and ending events can be separately specified.

[13]Essentially, specifying a data item as `general` allocates a 12 byte time-stamp while not specifying `general` allocates an 8 byte time-stamp. We considered other options for time-stamp size specification, such using separate keywords for range and granularity and directly specifying the size of the time-stamp, e.g. `8`. The first option was rejected for minimality considerations, and the second was rejected since we felt constants would be unintuitive to the user.

B.6 Section 5.7 <value specification> and <target specification>

The productions for the non-terminals <parameter specification> and <variable specification> are augmented to allow calendric system and property selection per-item.

<parameter specification> ::=
- <parameter name> [<indicator parameter>]
  [<calendric-property specification>]

<variable specification> ::=
- <embedded variable name> [<indicator variable>]
  [<calendric-property specification>]

Additional syntax rules:

1. The non-terminal <calendric-property specification> is defined in Appendix B.21.
2. If <calendric-property specification> is specified then <parameter name> must have the data type <character string type>. Similar remarks apply to <embedded variable name>.
3. If <calendric-property specification> is specified then the value contained in <parameter name> or <variable name> is interpreted as a temporal value according to the calendric system and/or calendar properties named by the <calendric-property specification>.
4. If <calendric-property specification> contains a <as clause> and the data type of the column corresponding to the <parameter specification> or <variable specification> is `event`, `interval`, or `span`, then the calendric system named in the <as clause> is used to translate the time-stamp into a temporal value.
5. If <calendric-property specification> contains a <with clause> and the data type of the column corresponding to the <parameter specification> or <variable specification> is `event`, `interval`, or `span`, then the property table named in the <with clause> are activated before translating the time-stamp, and deactivated immediately after translating the time-stamp.
6. If no `set calendric system` command has been entered then the implementation defined default calendric system is assumed.

B.7 **Section 5.9** <set function specification>

The production for the non-terminal <set function specification> is changed to allow for aggregate functions defined by a calendric system. No new syntactic rules are required to extend the existing aggregate functions for the temporal data types, but some disambiguating syntax rules are necessary.

<set function specification> ::=
    `count(*)`
  | <distinct set function>
  | <all set function>
• | <calendar defined set function>

<calendar defined set function> ::=
•   <distinct calendar defined set function>
• | <all calendar defined set function>

<distinct calendar defined set function> ::=
•   <calendar defined identifier> <open paren> `distinct` <value expression> <close paren>

<all calendar defined identifier> ::=
•   <calendar defined identifier> <open paren> [`all`] <value expression> <close paren>

Additional syntax rules:

1. A <calendac defined set function> is denoted by a <calendar defined identifier> prefixed with an <ampersand>.
2. Specifying `distinct` in a <calendar defined set function> removes null values and duplicate values before computing the the aggregate function.
3. Specifying `all` in a <calendar defined set function> removes null values but does not remove duplicate values before computing the the aggregate function. `all` is the default option.
4. Let T be the data type of the values given as arguments to the <set function specification>.
5. If `sum` is specified then T may not be `event` or `interval`.
6. If `avg` is specified then T may not be `interval`.
7. If `max` is specified then T may not be `interval`.
8. If `min` is specified then T may not be `interval`.

B.8 Section 5.10 <datetime value function>

The <datetime value function> non-terminal and its entire subtree are deleted.

The following productions are added.

<event value function> ::=
- `begin` <open paren> <interval value expression> <close paren>
- | `end` <open paren> <interval value expression> <close paren>
- | `first` <open paren> <event value expression> <comma> <event value expression> <close paren>
- | `last` <open paren> <event value expression> <comma> <event value expression> <close paren>
- | `present`
- | <calendar defined identifier> [<left paren> <token> [<comma> <token>]... <close paren>]

<interval value function> ::=
- `interval` <open paren> <event value expression> <comma> <event value expression> <close paren>
- | `intersect` <open paren> <interval value expression> <comma> <interval value expression> <close paren>
- | <calendar defined identifier> [<open paren> <token> [<comma> <token>]... <close paren>]

<span value function> ::=
- `span` <open paren> <interval value expression> <close paren>
- | `absolute` <open paren> <span value expression> <close paren>
- | <calendar defined identifier> [<open paren> <token> [<comma> <token>]... <close paren>]

Additional syntax rules:

1. <calendar defined identifier> must be a function defined via the currently active calendric system.
2. The number and types of the tokens in the parameter list of a <calendar defined identifier> function must agree in number and type with the definition of the function in the calendric system. A compilation error will be generated if any type errors are detected.

B.9 Section 5.13 <value expression>

The production for the non-terminal <value expression> is replaced with the following.

```
<value expression> ::=
          <numeric value expression>
        | <string value expression>
•       | <event value expression>
•       | <interval value expression>
•       | <span value expression>
```

B.10 Section 5.14 <numeric value expression>

The non-terminal <extract expression> and its entire subtree is deleted, and the production for the non-terminal <numeric primary> is changed to the following.

```
<numeric primary> ::=
          <unsigned value specification>
        | <column reference>
        | <set function specification>
        | <scalar subquery>
        | <case expression>
        | <open paren> <numeric value expression> <close paren>
•       | <calendar defined identifier> [ <open paren> <token> [ <comma> <token> ]...
                            <close paren> ]
        | <numeric cast specification>
        | <position expression>
        | <length expression>
```

Additional syntax rules:

1. <calendar defined identifier> must be an <identifier> representing a calendar defined function that returns a numeric value.
2. <calendar defined identifier> can be used to define field extraction functions for temporal values.

3. A calendar of the currently active calendric system is assumed to define the <calendar defined identifier>. If no `set calendric system` command has been entered during the session then a calendar of the implementation defined default calendric system is assumed to define the <calendar defined identifier>. Any other situation generates a compilation error.

4. The number and types of the tokens in the parameter list of a <calendar defined identifier> function must agree in number and type with the definition of the function in the calendric system. Any other situation generates a compilation error.

B.11 **Section 5.15** <string value expression>

The production for the non-terminal <character primary> is changed to the following.

```
<character primary> ::=
            <unsigned value specification>
          | <column reference>
          | <set function specification>
          | <scalar subquery>
          | <case expression>
          | <substring>
          | <fold>
          | <open paren> <character value expression> <close paren>
          | <character cast expression>
          | <character translation>
          | <form-of-use conversion>
•         | <calendar defined identifier> [ <open paren> <token> [ <comma> <token> ]...
                                  <close paren> ]
```

Additional syntax rules:

1. <calendar defined identifier> must be an <identifier> representing a calendar defined function that returns a string value.

2. A <calendar defined identifier> can be used to define field extraction functions for temporal values.

3. A calendar of the currently active calendric system is assumed to define the <calendar defined identifier>. If no `set calendric system` command has been entered during the session then a calendar of the implementation defined default calendric system is assumed to define the <calendar defined identifier>. Any other situation generates a compilation error.

4. The number and types of the tokens in the parameter list of a <calendar defined identifier> function must agree in number and type with the definition of the function in the calendar. Any other situation generates a compilation error.

B.12 Section 5.16 <datetime value expression>

The non-terminal <datetime value expression> and its entire subtree are replaced with the following productions.

<event value expression> ::=
- <event term>
- | <span value expression> <plus sign> <event value expression>
- | <event value expression> {<plus sign> | <minus sign>} <span value expression>

<event term> ::=
- <event factor>

<event factor> ::=
- <event primary>

<event primary> ::=
- <event literal>
- | <column reference>
- | <scalar subquery>
- | <case expression>
- | <event value function>
- | <open paren> <event value expression> <close paren>
- | <cast specification>

<interval value expression> ::=
- <interval primary>
- | <span value expression> <plus sign> <interval value expression>
- | <interval value expression> {<plus sign> | <minus sign>} <span value expression>

<interval primary> ::=
- <interval literal>
- | <column reference>
- | <scalar subquery>
- | <case expression>
- | <interval value function>
- | <cast specification>

<span value expression> ::=
- <span term>
- | <span value expression> {<plus sign> | <minus sign>} <span term>
- | <event value expression> <minus sign> <event term>
- | <open paren> <event value expression> <minus sign> <event term> <close paren>

<span term> ::=
- <span factor>
- | <span term> {<asterisk> | <solidus>} <factor>

- | <factor> <asterisk> <span term>

<span factor> ::=
- [<minus sign>] <span primary>

<span primary> ::=
- <span literal>
- | <column reference>
- | <scalar subquery>
- | <case expression>
- | <open paren> <span value expression> <close paren>
- | <cast specification>
- | <span value function>

Additional syntax rules:

1. The data type of an <event primary> is `event`.
2. The data type of an <interval primary> is `interval`.
3. The data type of a <span primary> is `span`.
4. The data type of an <event value expression> is `event`.
5. The data type of an <interval value expression> is `interval`.
6. The data type of a <span value expression> is `span`.
7. Invalid expressions as described in Appendix E are disallowed.

B.13 **Section 5.17** <interval value expression>

The non-terminal <interval value expression> and its subtree are deleted. It is effectively replaced by the new non-terminal <span value expression> defined in Section B.12.

B.14 Section 5.20 <predicate>

The production for the non-terminal <predicate> is replaced with the following.

```
<predicate> ::=
        <comparison predicate>
      | <between predicate>
      | <in predicate>
      | <like predicate>
      | <null predicate>
      | <quantified comparison predicate>
      | <exists predicate>
      | <unique predicate>
      | <match predicate>
•     | <precedes predicate>
•     | <meets predicate>
•     | <overlaps predicate>
•     | <contains predicate>
```

B.15 Section 5.21 <comparison predicate>

No new syntax rules are required, but additional disambiguating rules are required for span comparison.

1. The <less than operator>, <greater than operator>, and <equals operator> are valid for span comparison.

B.16 Section 5.30 <overlaps predicate>

The following productions are added for the new comparison operators. (The production for the `overlaps` predicate is left as is though its semantics have changed as described in Section 4.6.)

```
<precedes predicate> ::=
•       <row value expression 1> precedes <row value expression 2>

<meets predicate> ::=
•       <row value expression 1> meets <row value expression 2>

<contains predicate> ::=
•       <row value expression 1> contains <row value expression 2>
```

This grammar is overly permissive in that it generates semantically illegal expressions. This is, however, consistent with the grammar originally provided in the SQL2 proposal for datetime value comparison. Expressions violating type constraints will be detected during semantic analysis.

Additional syntax rules:

1. Let T_1 be the type of <row value expression 1>.
2. Let T_2 be the type of <row value expression 2>.
3. T_1 and T_2 must be either `interval` or `event`.
4. T_1 and T_2 shall be comparable as defined in Table 9.
5. Any comparison involving the `interval` or `event` data types not listed in Table 9 is disallowed.

B.17 **Section 6.1** <schema definition>

The production for the non-terminal <schema definition> is changed to the following.

<schema definition> ::=
        `create schema` <schema authorization clause>
            [<schema character set specification>]
            [<schema element>...]

B.18 **Section 6.2** <time displacement table definition>

The non-terminal <time displacement table definition> and its entire subtree are deleted.

B.19 **Section 6.7** <default clause>

The production for the non-terminal <default clause> is changed to the following.

<default clause> ::=
        <literal>
• | <event value function>
• | <interval value function>

• | <span value function>
| `user`
| `system user`
| `null`

Additional syntax rules:

1. If <event value function>, <interval value function>, or <span value function> is specified then any parameters passed to these functions must be property values representing a special time value or literal values.
2. Let T be the type of the column being initialized.
3. If T is `event`, `interval`, or `span` then `user` and `system user` may not be specified.
4. If T is `event` then either a <literal> representing an <event literal> or an <event value function> may be specified. The calendric system used to interpret the constant is the calendric system whose scope is the smallest scope which encompasses the literal. The properties used to interpret the constant are the set of properties active when the default clause is executed.
5. If T is `interval` then either a <literal> representing an <interval literal> or an <interval value function> may be specified. The calendric system used to interpret the constant is the calendric system whose scope is the smallest scope which encompasses the literal. The properties used to interpret the constant are the set of properties active when the default clause is executed.
6. If T is `span` then either a <literal> representing a <span literal> or a <span value function> may be specified. The calendric system used to interpret the constant is the calendric system whose scope is the smallest scope which encompasses the literal. The properties used to interpret the constant are the set of properties active when the default clause is executed.

B.20 Section 8.1 <module>

The production for the non-terminal <module contents> is changed to include a global calendric system declaration statement, and a new non-terminal <declare calendric system> is added to define this statement.

<module contents> ::=
<declare cursor>
• | <declare calendric system>
| <dynamic declare cursor>
| <procedure>

<declare calendric system> ::=
• `declare calendric system` <as clause>

B.21 Section 8.4 <SQL statement>

The production for the non-terminal <SQL set environment statement> is changed to include a session-level calendric system selection command, and an additional production is added to define the calendric system selection command.

```
<SQL set environment statement> ::=
          <SQL set identifier statement>
        | <set constraints mode statement>
        | <set transaction statement>
•       | <set properties statement>

<set properties statement> ::=
•         set properties <property specification>

<calendric-property specification> ::=
•         <as clause> [ <with clause> ]
•       | [ <as clause> ] <with clause>

<as clause> ::=
•         as { default | <calendric system name>}

<with clause> ::=
•         with { previous | default | <property table name> | <table value expression> }
```

Additional syntax rules:

1. In a sequence of SQL statements, the last calendric system specified in a `set calendric system` command remains in effect until a new `set calendric system` command is entered.
2. A `set calendric system as default` statement reactivates the implementation defined default calendric system.
3. The non-terminal <calendric system name> must be an <identifier> naming a calendric system.
4. The non-terminal <property table name> is the name of a property table defining properties of the named calendric system.

5. Specifying `set properties with previous` causes the previous set of active properties to be reactivated.

6. Specifying `set properties with default` causes the implementation defined set of default properties to be activated.

7. The non-terminal <table value expression> enumerates the rows of a property table.

8. A property table must have the schema (property:*character string*, value:*character string*). The command to create a persistent property table with property values of length at most twenty characters is the following. (The keyword `varchar` indicates a varying length character string.)

```
create table property_table (property varchar 20, value varchar 20)
```

B.22 Section 13.6 <set local time displacement table>

The non-terminal <set local time displacement table> and its entire subtree are deleted. Local time displacement is calendar specific.

B.23 Section 13.7 <set local timezone displacement>

The non-terminal <set local timezone displacement> and its entire subtree are deleted. Local time displacement is calendar specific.

C SQL2 Calendric System

This appendix briefly summarizes how the calendric system in the SQL2 proposal document could be implemented in our proposal. This calendric system consists of the single calendar, the Gregorian calendar, defined over the epoch 1 A. D. to 9999 A. D. We therefore concentrate on how the semantics of this calendar can be accommodated in this proposal. For each feature of the calendar, we show constructs in our proposal that provide equivalent facilities.

The SQL2 *date* data type can be implemented using the event data type. To simulate a *date*, format properties `event_input_format` and `event_output_format` can be inserted into the calendric system's default property table which interpret event input and output in *date* format.

The SQL2 *time* data type can be implemented using the event data type and defining override input and output format properties with the fields *hour*:*minutes*:*seconds* and optional fractional seconds, if desired. These properties would be kept in a separate table that could be used to override the default event input and output formats defined in conjunction with *dates* and contained in the default property table. Assuming the overriding property table is named `time_format_table`, a session-level override for the *time* format would use the following command.

```
set properties with time_format_table
```

A by-item override would use the modifier `with time_format_table` in a select clause.

The SQL2 *timestamp* data type can be implemented using the event data type and defining override input and output format properties with the fields *year*−*month*−*day*:*hour*:*minutes*:*seconds* and optional fractional seconds, if desired. These properties would be kept in a separate table that could be used to override the default event input and output formats defined in conjunction with *dates* and contained in the default property table. Assuming the overriding property table is named `timestamp_format_table`, a session-level override for the *timestamp* format would use the following command.

```
set properties with timestamp_format_table
```

A by-item override would use the modifier `with timestamp_format_table` in a select clause.

A property is added to the property table to indicate the name of the local timezone. The local timezone name must be the name of a timezone defined in the timezone displacement table.

A calendar defined function, `at(`*time_value*`,` tz_1`,` tz_2`)`, is defined for conversion of time value between different timezones. *time_value*, a time value local to timezone tz_1, is converted to the simultaneous time in timezone tz_2. tz_1 and tz_2 must be defined in the current time displacement table.

Conversion to UTC before storage is implemented by `at(`*time_value*`, 'local', 'utc')`. Conversion to local time from UTC on retrieval is implemented by `at(`*time_value*`, 'utc', 'local')`. The strings `'utc'` and `'local'` denote the those timezones in the time displacement table.

The built-in function `present` returns an event representing the current time. Selecting the appropriate output format property as described above will display this event as the current date, time, or timestamp. To compute the current UTC time or timestamp, the `at` function may be used, e.g., `at(present, 'local', 'utc')`.

The functions `year_number(`*time_value*`)`, `month_number(`*time_value*`)`, `day_number(`*time_value*`)`, `hour_number(`*time_value*`)`, `minute_number(`*time_value*`)` and `second_number(`*time_value*`)` are defined by the calendar. Each function returns a numeric value representing the given field. Additional functions `year_name(`*time_value*`)`, `month_name(`*time_value*`)`, `day_name(`*time_value*`)`, `hour_name(`*time_value*`)`, `minute_name(`*time_value*`)` and `second_name(`*time_value*`)` can be defined which return the string name of a field. These functions provide facilities that are similar to the SQL2 `extract(`*field* `from` *time_value*`)` function.

Property values would be set as follows:

- *locale*—the calendric system must support all timezones as might be specified in a *local time displacement table*. The value of this property is set to one of these supported values.
- *event_input_format*—the value of this property is set to the following string corresponding to the SQL2 format for *datetimes*.

  ```
  '<year,arabic_number>-<month,english_month_names>-<day,english_day_names>:
  <hour,arabic_number>:<minute,arabic_number>:<seconds,arabic_number>'
  ```

  An overriding format string for SQL2 *date*s is constructed as follows.

  ```
  '<year,arabic_number>-<month,english_month_names>-<day,english_day_names>'
  ```

  An overriding format string for SQL2 *time*s is constructed as follows.

  ```
  '<hour,arabic_number>:<minute,arabic_number>:<seconds,arabic_number>'
  ```

- *event_output_format*—the value of this property is set to the same string as the value of the *event_input_format* property. The same overriding format strings are used.

- *interval_input_format*—the value of this property is shown below. This string assumes that two SQL2 *time-stamps* are used to represent the beginning and end of an interval. Overriding format strings can be constructed from the *date* and *time* formats above.

```
'<year1,arabic_number>-<month1,english_month_names>-<day1,english_day_names>:
<hour1,arabic_number>:<minute1,arabic_number>:<seconds1,arabic_number>,
<year2,arabic_number>-<month2,english_month_names>-<day2,english_day_names>:
<hour2,arabic_number>:<minute2,arabic_number>:<seconds2,arabic_number>'
```

- *interval_output_format*—the value of this property is the same as for the *interval_input_format* property. The same comments on overrides apply.

- *span_input_format*—the value of this property corresponds with the SQL2 notion of *year-month* interval.

```
'<year,arabic_number>-<month,arabic_number>'
```

  An override for *day-time* intervals is as follows.

```
'<day,arabic_number>:<hour,arabic_number>:<minute,arabic_number>:
<seconds,arabic_number>'
```

- *span_output_format*—the value of this property is the same as for the *span_input_format* property. The same comments on overrides apply.

- *input_epoch_override*—the value of this property is set to the empty string (`''`) since the SQL2 calendric system is composed of a single calendar.

- *beginning*—the value of this property is set to the string `'beginning'`.

- *forever*—the value of this property is set to the string `'forever'`.

D Time-card Calendric System

In this appendix, we show how a simple payroll oriented calendric system can be constructed. This calendric system consists of a single calendar. We assume that the pay period for a certain company is one month, where a month is four consecutive weeks of seven days each. We assume that months are numbered from 1 to 12 (their actual names being immaterial to the payroll system), and that years retain the traditional Christian numbering.

We will assume that employee hours are recorded for an entire week on Friday, the last day of the week. For example, if an employee works 40 hours during some week, then on Friday of that week this fact will be recorded in the database. Since payroll checks are issued monthly, we can sum a single employee's hours for the current month using the following query.

```
select sum(hours)
from employee
where month_number(week) = month_number(present) and
    year_number(week) = year_number(present)
group by employee_number
```

The above query assumes that the calendar defines two field extraction functions `month_number` and `year_number` that, given an event, return the month component and the year component of the event, respectively, as integers.

Properties values are set as follows:

- *locale*—time cards generally do not involve timezones, and so we set the value of this property to the empty string (`''`).
- *event_input_format* and *event_output_format*—events in this calendar name the year, month, week, and day as numeric values. For example, `91/3 week 3 day 2` is equivalent to the Gregorian date March 15, 1991. Assuming that event constants have the same format on both input and output, we set the event input and output property values equal to the string below. (`arabic_number_2` and `arabic_number` are translation tables producing arabic numerals.)

```
'<year,arabic_number_2>/<month,arabic_number>␣week␣<week,arabic_number>␣
                    day␣<day,arabic_number>'
```

- *interval_input_format* and *interval_output_format*—interval constants are assumed to be two events separated by commas and delimited by square brackets. We set the interval input and output formats to the following string.

```
'<year1,arabic_number_2>/<month1,arabic_number>␣week␣<week1,arabic_number>␣
          day␣<day1,arabic_number>,␣<year2,arabic_number_2>
<month2,arabic_number>␣week␣<week2,arabic_number>␣day␣<day2,arabic_number>'
```

- *span_input_format* and *span_output_format*—spans are a number of years, months, weeks, and days. The span input and output formats are set to the following.

```
'year␣<year,arabic_number>␣month␣<month,arabic_number>␣<week,arabic_number>␣
                      day␣<day,arabic_number>'
```

- *input_epoch_override*—since this calendric system consists of a single calendar, the value of this property is set to the empty string (`''`).

- *beginning* and *forever*—since the payroll information deals only with the recent past, the concepts of *beginning* and *forever* are not relevant. We set the values of these properties to the empty string (`''`).

E Invalid Arithmetic Expressions

Table 19 and Table 20 list all arithmetic expressions that are invalid within our algebra of expressions. Such expressions are invalid because no reasonable semantics could be defined for the expression or the expression violates typing rules. Section 4.5 provides a detailed discussion of arithmetic involving temporal values.

Operand 1	*Operator*	*Operand 2*
span	−	*event*
span	+	*numeric*
span	−	*numeric*
span	*	*span*
span	*	*event*
span	/	*event*
numeric	+	*span*
numeric	−	*span*
numeric	+	*event*
numeric	−	*event*
numeric	/	*span*
numeric	*	*event*
numeric	/	*event*
event	+	*event*
event	+	*numeric*
event	−	*numeric*
event	*	*span*
event	/	*span*
event	*	*event*
event	/	*event*
event	*	*numeric*
event	/	*numeric*
interval	*	*span*
interval	/	*span*

Table 19: Invalid Arithmetic Expressions (Part 1)

Comparing Table 8 in Section 4.5 to Table 19 and Table 20 shows that there are over three times as many illegal expressions as legal ones. We attribute this to two main causes. First, many of the operations have intuitive semantics only when operands appear in the correct order. For example, *event* − *span* is legal while *span* − *event* is not. Second, we allow very limited arithmetic support for interval calculation. In fact, intervals may appear in only three types of arithmetic expressions. All remaining combinations of intervals and other data types and operators are shown in Table 20.

Operand 1	*Operator*	*Operand 2*
span	−	*interval*
span	*	*interval*
span	/	*interval*
interval	+	*event*
interval	−	*event*
interval	*	*event*
interval	/	*event*
event	+	*interval*
event	−	*interval*
event	*	*interval*
event	/	*interval*
numeric	+	*interval*
numeric	−	*interval*
numeric	*	*interval*
numeric	/	*interval*
interval	+	*numeric*
interval	−	*numeric*
interval	*	*numeric*
interval	/	*numeric*
interval	+	*interval*
interval	−	*interval*
interval	*	*interval*
interval	/	*interval*

Table 20: Invalid Arithmetic Expressions (Part 2)

F Properties

The properties in Table 2 can be classified as either timezone-related, format-related, input priority, or special time values.

The `locale` property is used to specify a location for timezone displacement. The locale is calendar dependent, and interpretation of this property is the responsibility of the calendar. Locales provide more flexibility than timezones since time displacements may not be uniformly applied to all locations within that timezone. For example, the state of Arizona is located within the mountain timezone of the United States. However, most locations in Arizona do not observe daylight savings time, though some do.

The format-related properties describe the formats of temporal constants that are to be translated into time-stamps, and the formats of strings that are returned as the result of a query. Specifically, format properties identify format strings which describe input and output formats. Each *field* in the output string is described by a *field specification* which names the field, a table of translation strings, and a formatting specification for the field.

The format-related properties are listed below. An input and output property is defined for each temporal type. The value of a format-related property is a string containing a sequence of <field specifications>. The field specification describes names a field, a translation table, and a formatting specification for the field. For output properties, each field specification describes how one particular field of the output string is formatted. Background characters within the format string are translated directly to their corresponding positions within the output string. The syntax is shown in Appendix B.2. Format string examples are shown in Appendices C and D.

- `default_input_distribution`—a string specifying the name of default distribution.
- `distribution_input_format`—a string specifying the format of indeterminate temporal constants with specified distribution in an SQL query.
- `distribution_output_format`—a string specifying the format of indetermiante temporal constants returned the result of a query.
- `indeterminacy_input_separator`—a string specifying the separator for indeterminate datetime or interval temporal constants in an SQL query.
- `indeterminacy_output_separator`—a string specifying the separator for indeterminate datetime or interval temporal constants returned as the result of a query.
- `instant_input_format`—a string specifying the format of datetime temporal constants in an SQL query.
- `instant_output_format`—a string specifying the format of datetime constants returned as the result of a query.
- `interval_input_format`—a string specifying the format of interval temporal constants in an SQL query.
- `interval_output_format`—a string specifying the format of interval temporal constants returned as the result of a query.
- `missing_distribution`—a string specifying the name of missing distribution.
- `period_input_separator`—a string specifying the separator which separates two datetime temporal constants within a period temporal constants in an SQL query.
- `now`—a string specifying the format of temporal instant `now` in an SQL query.
- `now_separator`—a string specifying the separator which separates an instant `now` and an interval within temporal instant `now-relative` in an SQL query.
- `period_output_separator`—a string specifying the separator which separates two datetime temporal constants within a period temporal constants returned as the result of a query.
- `period_output_delimiters`—a string specifying the left and right delimiters of period temporal constants returned as the result of a query.

We note that the parsing and interpretation of format strings is necessarily done at run-time. Should an error occur when interpreting a format string, an error will be returned to the application through the `sqlcode` parameter.

The value of the ıoverride_input_epoch property names a calendar in the current calendric system that will first be used to translate temporal constants.

The special time value properties allow language independence for special time values. We defined two special time value properties in Section 4.4. They are described below.

- `beginning`—the name of a special event value preceding any event representable on the physical time-line.
- `forever`—the name of a special event value following any event representable on the physical time-line.