Implementing Calendars and Temporal Rules in Next Generation Databases *

Rakesh Chandra, Arie Segev †

Walter A. Haas School of Business
University of California, Berkeley
and
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
Berkeley, CA 94720

Michael Stonebraker

Computer Science Division, EECS Department
University of California, Berkeley, CA 94720

email: crakesh@csr.lbl.gov, segev@csr.lbl.gov, mike@postgres.berkeley.edu

Abstract

In applications like financial trading, scheduling, manufacturing and process control, time based predicates in queries and rules are very important. There is also a need to define sets of time points or intervals. We refer to these sets as calendars. This paper presents a system of calendars that allows specification of natural-language time-based expressions, maintenance of valid time in databases, specification of temporal conditions in database queries and rules, and user-defined semantics for date manipulation. A simple set based language is proposed to define, manipulate and query calendars. The design of the parser and an algorithm for efficient evaluation of calendar expressions is also described. The paper also describes the implementation of time-based rules in POSTGRES using the proposed system of calendars.

Keywords: Calendars, Temporal Databases, Temporal Rules, Extensible Databases.

*Issued as LBL Technical Report 34229

†The work of this author was supported by an NSF Grant IRI-9116770 and by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

1 Introduction

Temporal conditions and constraints arise very often in applications like financial trading, scheduling, manufacturing and process control. Even though temporal conditions can easily be expressed in natural languages, it is difficult or impossible to express them in temporal query languages like TQUEL[Sno87] and TSQL[NA89]. For example, consider the condition:

The expiration date of an option is the 3^{rd} *Friday of November if it is a business day, else it is the business day preceding the above mentioned Friday*

Now suppose a user would like to find the closing price of the stock underlying the option on this expiration date. It would be ideal if a database query language provided the capability to define a variable called *expiration-date* that expressed the condition given above and then allow this variable to be used in the query:

Retrieve (stock.price) when stock overlaps expiration-date

As another example, consider the following query posed by an administrator to a university database:

Retrieve the names of all foreign students who worked more than 20 hours in any week during the semester

Note that the days in a *semester* are specific to the univerity and change from year to year. There is a need for a facility to define such application specific time points or collections of time points in the query language. Unfortunately temporal query languages do not support this facility[1].

Many interesting time-series studied in economics and finance are recorded at equal intervals of time. These are called regular time-series. Thus, we always know the future time points at which these time-series have values. If these sets of future time points could be expressed by a database query language, it would be unnecessary to store the time points associated with time-series observations, since they could be generated on request. Such a capability would allow the maintenance of valid time in the database. TQUEL allows the specification of an interval but an arbitrary subset of the time points within this interval cannot be easily expressed. For example, using TQUEL, it can be expressed that the GNP time-series, which records the sum total of economic activity in the country in a quarter, is stored for all valid time points in the interval (Jan 1, 1985 , Dec 31, 1993). But the valid time points, the last day of every quarter in every year, cannot be expressed in TQUEL.

Lastly, it is quite common for applications to have their own semantics for date arithmetic. For example, [Sto90a] pointed out that the yield calculation on financial bonds uses a calendar that has 30 days in every month for date arithmetic, but 365 days in the year for the actual yield

[1] Note that *Temporal Elements* [Gad88],[TCG+93] allow reference to a set of time intervals but not in the context of calendars. Also the time intervals must be specified explicitly.

calculation. If date functions supplied by commercial databases are used, results will be incorrect because these date functions always assume the underlying calendar as the gregorian calendar. Since many date conventions can exist, a facility that allows date functions to take user-defined calendars as arguments, would to solve this problem.

In [CS93], we outlined the database requirements for managing temporal financial data, including calendars and temporal rules. This paper elaborates on the calendar representation and manipulation, and presents a scheme for activating database rules based on calendar expressions. The proposed calendar system uses an algebra based on collection intervals [LMF86] to allow:

- the expression of temporal conditions in queries and rules that reflect the semantics of natural language time based expressions.
- the maintenance of valid time in databases.
- the creation of application specific calendars.
- different semantics for date arithmetic.

This paper describes the implementation of calendars and time-based rules in the extensible database, Postgres [Sto90b]. Extensible databases provide:

- object support by allowing the definition and manipulation of complex data types.
- support for the declaration of operators that take complex data types as arguments.
- knowledge management by allowing the definition of rules to reflect application semantics.
- a "fast path" capability that allows the creation of indexes to optimize the performance of these operators.

The rest of the paper is organized as follows. A summary of relevant research is presented in section 2. Section 3 discusses the model and implementation of calendars. The calendar expression language is presented along with the design of a parser and an algorithm for creating efficient evaluation plans for the parsed calendar expression. Section 4 discusses the utility of time-based rules and their implementation. Section 5 provides a comparative study of other calendric system proposals. Section 6 concludes the paper with a summary of the contributions of this research and problems for future research.

2 Relevant Research

Many researchers in the artificial intelligence community have studied the problems of temporal representation. [All85] defined the *interval* as a primitive temporal entity and also provided a

set of possible relationships between two intervals. He showed how the *interval* could be used to allow representation of indefinite and relative temporal knowledge. [LMF86] built on this work by introducing the operators *dicing* and *slicing* to allow the representation of natural language expressions. Their work is the basis for the constructs and language defined in this paper.

[SS92], [SS93] first introduced the idea of an extensible calendric system. The proposal described modifications to SQL2[Mel90] that support multiple calendars. The modifications reduced the complexity of the language while simultaneously increasing the expressive power. The paper also describes a toolkit that provided tools to define new calendars and calendric systems. A comparative study of Soo's proposal and our proposal is provided in section 5. In addition to the above research, there have been other extensions to SQL to accommodate date and time data types [Dat88]. In addition, the commercial database Oracle [Cor87] supports the date data type. The main drawbacks with these extensions is that (a) they assume a uniform single calendar, the gregorian calendar and (b) the extensions don't provide support for natural language time-based expressions. [DW90] provides a thorough critique of one of these proposals. Our proposal corrects the shortcomings of these models by providing multiple calendars and support for natural language expressions.

In addition to the work described above, many researchers Our work is in the context of extensible relational databases, but the results can be used to augment studies in the area of temporal object-oriented databases such as [RS91], [RS93], [SC91] and [WD92]. have developed temporal data models. Also, two surveys and a book on temporal databases are available in [Sno90],[RGM92], and [TCG+93].

3 Calendars

3.1 Calendar Algebra

In this section, we briefly discuss the algebra for collection intervals and operators developed in [All85] and [LMF86]. A *Calendar* is formally defined as a structured set of intervals and the *Order* of a calendar is defined as the depth of the set. ([LMF86] defined the *Order* of a collection of intervals). Thus, the set $S = \{(l_1, u_1), (l_2, u_2), \cdots, (l_n, u_n)\}$ is a calendar of order 1 while $R = \{S_1, S_2, \cdots, S_m\}$, where $S_i = \{(l_j, u_j)\}$, is a calendar or order 2.

The following relationships, introduced by [All85], between two intervals are useful. If int_1 and int_2 are two intervals, then

- int_1 overlaps $int_2 := (int_1 \cap int_2)$
- int_1 during $int_2 := ((l_1 \geq l_2) \wedge (u_2 \geq u_1))$
- int_1 meets $int_2 := (u_1 = l_2)$

- $int_1 < int_2 := (u_1 \leq l_2)$
- $int_1 \leq int_2 := ((l_1 \leq l_2) \wedge (u_2 \geq u_1))$

Based on these relationships, the following operators are defined:

- overlaps(int_1,int_2) := if $(int_1 \cap int_2) \neq \epsilon$ return(true)
- during(int_1,int_2) := if $(l_1 \geq l_2 \wedge u_2 \geq u_1)$ return($true$)
- meets(int_1,int_2) := if $(u_1 = l_2)$ return($true$)
- $<$(int_1,int_2) := if $(u_1 \leq l_2)$ return($true$)
- $\leq$(int_1,int_2) := if $(l_1 \leq l_2 \wedge u_2 \geq u_1)$ return($true$)

The operators *overlaps*, *during*, *meets*, $<$ and $\leq$ will be collectively referred to as *setops*.

Two other operators are used to facilitate the manipulation of calendars. The $foreach$ operator (similar to *dicing* introduced by [LMF86]) is used in conjunction with *setops*. For example, the strict $foreach$ operator (:), takes as arguments, a *setop*(Op), a calendar of order-1 (C) and an interval ($I =< t_s, t_e >$). It applies Op to each interval in C and I. Formally, the strict $foreach$ operator is defined as:

$$\{C : Op : I\} \equiv \{c \cap I | (c \in C) \wedge Op(c, I)\}/\{\epsilon\}$$

where ϵ denotes the interval $(-\infty, \infty)$ that is excluded from the result.

The $foreach$ operator can also take a calendar (C_1) as its third argument. In this case, the $foreach$ operator is applied as described above for every element in C_1. Formally,

$$\{C : Op : C_1\} \equiv \{c \cap i | (c \in C) \wedge Op(c, i) \wedge (i \in C_1)\}/\{\epsilon\}$$

The following examples illustrate the above formulas. Suppose $WEEKS$ is a calendar representing the weeks in the year 1993.

$$WEEKS \equiv \{(-4, 3), (4, 10), (11, 17), (18, 24), (25, 31), (32, 38), (39, 45), \cdots\}$$

Note that the first interval (-4,3) does not contain 0 (If 0 were allowed to be contained in the set then the interval would have been (-4,2) and January would have to be (0,30) for consistency. Since this is unintuitive, we adopt the convention that an interval will never contain 0.) Let {Jan-1993} be the interval {(1,31)}. Then by the formula for the strict $foreach$ operator, we have:

$$\{WEEKS : during : Jan - 1993\} \equiv \{(4, 10), (11, 17), (18, 24), (25, 31)\}$$

Note that {Jan-1993} is an interval and that the $foreach$ operator applied the *setop*, *during* to every element of $WEEKS$ and {Jan-1993}. Now suppose {Year-1993} is a calendar of the months in 1993. Thus,

$$\{Year - 1993\} \equiv \{(1,31),(32,59),(60,90),(91,120),\cdots\}$$

Then,

$$\{WEEKS : during : Year - 1993\} \equiv \{\{(4,10),(11,17),(18,24),(25,31)\},$$

$$\{(32,38),(39,45),(46,52),(53,59)\},$$

$$\{(60,66),(67,73),(74,80),(81,87)\},$$

$$\{(95,101),(102,108),(109,115)\},\cdots\}$$

In the above case, the third argument to the $foreach$ operator is a calendar. The result is a calendar of order-2 and reflects the weeks completely contained in every month of 1993.

The relaxed $foreach$ operator, denoted by (.) is formally defined as :

$$\{C.Op.I\} \equiv \{c|(c \in C) \wedge Op(c,I)\}/\{\epsilon\}$$

Since the *during* operator will have the same result with the strict and relaxed $foreach$ operator, we use *overlaps* to illustrate the difference.

$$\{WEEKS : overlaps : Jan - 1993\} \equiv \{(1,3),(4,10),(11,17),(18,24),(25,31)\}$$

The result is a calendar of all the weeks or partial weeks in January 1993. On the other hand,

$$\{WEEKS.overlaps.Jan - 1993\} \equiv \{(-4,3),(4,10),(11,17),(18,24),(25,31)\}$$

The result is a calendar with all the weeks that overlap with January 1993.

The *selection* operator (similar to *slicing* introduced by [LMF86]) $[x]$/C selects the x^{th} interval from the calendar C if C is an order-1 calendar and x is an integer. The selection predicate $[x]$ is also allowed to be a list of integers and an integer range. If n is used in the selection predicate, the last interval is selected from C and if a minus sign prefixes an integer in the predicate, selection is done from the end of the set. For example, $[-2]$/C, selects the second element from the end of C. If C is a calendar of order greater than 1, say n, then it selects the x^{th} element of each calendar of order $n-1$. For example, the third week in January 1993 would be expressed by:

$$[3]/WEEKS : overlaps : Jan - 1993 \equiv [3]/\{(1,3),(4,10),(11,17),(18,24),(25,31)\}$$

$$\equiv \{(11,17)\}$$

The third week of every month is expressed by:

$$[3]/WEEKS : overlaps : Year - 1993 \equiv [3]/\{\{(1,3),(4,10),(11,17),(18,24),(25,31)\}$$

$$\{(32,38),(39,45),(46,52),(53,59)\},\{(60,66),(67,73),(74,80),(81,87),(88,90)\},$$

$$\{(91,94),(95,101),(102,108),(109,115)\},\cdots\}$$

$$\equiv \{(11,17),(46,52),(74,80),(102,108),\cdots\}$$

[LMF86] showed that the operators described above are adequate to describe natural language time based expressions. The following section shows how this algebra can be implemented in an extensible database by creating database tables, procedures, a calendar expression language, a parser and an algorithm for efficient evaluation of calendar expressions.

3.2 Calendar Implementation

The set of basic calendars are SECONDS, MINUTES, HOURS, DAYS, WEEKS, MONTHS, YEARS, DECADES and CENTURY. Relationships between basic calendars are maintained in the table CALTABLE. This table has the following structure:

```
CALTABLE( calendar1 : text, calendar2 : text, list : int[])
```

Here $calendar_i$ is a text variable and the *list* is an array of integers. For example, the tuple {YEARS,MONTHS,12} expresses the relationship between years and months, i.e., 12 Months $\equiv$ 1 Year. The relationship between years and days is more complicated because of the presence of a leap year every 4 years. Since the UNIX system start date is taken as Jan 1, 1970, the entry in the table would be {YEARS, DAYS, (365,365,366,365)}. This means that in the first year (1970) there are 365 days, the second year has 365 days and so on. After 4 years the same pattern is repeated. The relationship between MONTHS and DAYS is described by the following tuple:
{MONTHS, DAYS, (31,28,31,30,31,30,31,31,30,31,30,31,31,28,31,30,31,30,31,31,30,31,30,31,
31,29,31,30,31,30,31,31,30,31,30,31,31,28,31,30,31,30,31,31,30,31,30,31)}

The table, CALENDARS, is used to store information on user-defined calendars and has the following structure:

```
CALENDARS( name :                text,
           derivation-script:    text,
           eval-plan:            text,
           lifespan:             float[2],
           materialization:      float[2],
           granularity:          text,
           values:               interval[])
```

Calendars	
Name	Tuesdays
Derivation-Script	[2]/DAYS:during:WEEKS
Eval-Plan	set of procedural statements
Lifespan	(1985,∞)
Materialization	(1991,1991)
Values	{(2,2),(9,9),$\cdots$,}

Figure 1: Table CALENDARS

This table records the *name* of the user-defined calendar. The set of statements in the calendar expression language (described below) that are used to derive the calendar is stored in *derivation-script*. The *eval-plan* is the list of procedural statements that are used to generate values of the calendar. This plan is created by the parser by parsing the *derivation-script*. *Lifespan* is the maximum and minimum time point that the calendar describes and *materialization* is the range of the subset of calendar values that the application generates and stores in *values*. Note that *values* is an array of intervals and can thus store order-1 calendars[2]. The *granularity* of a calendar must be one of the basic calendars. In most cases, the *granularity* can be inferred from the *derivation-script*. Figure 1 illustrates the calendar *Tuesdays*.

The tuple stored in the table *Calendars* is that for the calendar, *Tuesdays*. *Tuesdays* is derived by the statement:{[2]/DAYS:during:WEEKS} which means the 2^{nd} day of every week. (Note that Monday is taken to be 1 and Sunday as 7). The parser will read the *derivation-script* and output a set of procedural statements to generate the specific calendar values. The lifespan is from 1985 to ∞ and only *tuesdays* in the the year 1991 are explicitly stored in the database.

3.3 Calendar Expression Language

This language is needed to provide a way to create and manipulate new calendars. A calendar script consists of:

- Assignment statements of the form *variable = calendar expression*
  where *variable* is of type order-*n* calendar. These variables need not be declared before use. *Calendar expressions* use temporary variables, *setops*, the *foreach* operator and *selection* operators.

[2]A calendar of higher order can also be accommodated in this scheme. For example, a calendar of order-2 can be thought of as an array of an array of intervals and be stored as a one dimensional array.

- *if* clauses of the form *if (condition) action else action*
  action is another calendar script while *condition* is a calendar expression. If the calendar expression evaluates to a null set, the condition is false.

- while clauses of the form *while (condition) action*

The following examples are used to illustrate the calendar expression language. Consider the following script that is used to define the calendar $EMP\text{-}DAYS$. This script defines the days on which national employment figures are announced by the government and is "the last day of every month in the year. If this is a holiday, then the preceding business day".

{

$$\begin{aligned} LDOM &= [n]/DAYS : during : MONTHS; \\ LDOM\_HOL &= LDOM : intersects : HOLIDAYS; \\ LAST\_BUS\_DAY &= [n]/AM\_BUS\_DAYS :<: LDOM\_HOL; \\ &\quad return(LDOM - LDOM\_HOL + LAST\_BUS\_DAY); \end{aligned}$$

}

The temporary variable, $LDOM$ is a calendar containing the last day of every month. The calendar expression {DAYS:during:MONTHS} produces a calendar with the days in each month $\equiv$ {{(1,1),$\cdots$,(31,31)},{(32,32),$\cdots$,(59,59)},$\cdots$}. [n] is used to select the last day of every month, $\equiv$ {(31,31),(59,59),(90,90),$\cdots$}. $HOLIDAYS$ is a calendar containing the days on which there are holidays, e.g., {(31,31),(90,90)}, which indicates that Jan 31^{st} and Mar 30^{th} are holidays. {LDOM:intersects:HOLIDAYS} gives a calendar that contains the days that were last days in the month and also holidays, $\equiv$ {(31,31),(90,90)}. {AM_BUS_DAYS:<:LDOM_HOL} results in an order-2 calendar in which each component order-1 calendar contains a list of AM_BUS_DAYS that precede a holiday that was also the last day of the month. From this order-2 calendar, the last element of each order-1 calendar is chosen. This is shown below:

$$\begin{aligned} AM\_BUS\_DAYS &\equiv \{(1,1),(2,2),\cdots,(30,30),\cdots, \\ &\quad (88,88),(91,91),\cdots\} \\ AM\_BUS\_DAYS :<: LDOM\_HOL &\equiv \{\{(1,1),\cdots,(30,30)\} \\ &\quad \{(1,1),\cdots,(88,88)\},\cdots\} \\ then,\ [n]/AM\_BUS\_DAYS :<: LDOM\_HOL &\equiv \{(30,30),(88,88),\cdots\} \end{aligned}$$

The calendar returned by the script is (LDOM - LDOM_HOL + LAST_BUS_DAY)
$\equiv$ {(31,31),(59,59),(90,90),$\cdots$}$-${(31,31),(90,90),$\cdots$} + {(30,30),(88,88),$\cdots$}
$\equiv$ {(30,30),(59,59),(88,88),$\cdots$}.

The following example illustrates the use of the *if* clause. The script expresses the "third Friday of the expiration month if a business day else the preceding business day".

```
{
        Fridays = [5]/DAYS:during:WEEKS;

        temp1   = [3]/Fridays:overlaps:Expiration-Month;
        /* 3rd Friday of the expiration month where expiration
           month is a predefined calendar */

        if (temp1:intersects:holidays) /* if holiday */

                 return([n]/AM_BUS_DAYS:<:temp1);
                 /* last business day before 3rd friday of expiration month */

        else
                    return(temp1);
}
```

The following script illustrates the use of the *while* clause. It will alert the user when the current day is the last trading day of a financial option. The last trading day is the seventh business day preceding the last day of the expiration month.

```
{
         temp1 = [n]/AM_BUS_DAYS:during:Expiration-Month;
         /* last business day of the expiration month */

         temp2 = [-7]/AM_BUS_DAYS:<:temp1;
         /* -7 selects the seventh element from the end of
            the set. This expression selects the seventh
            business day preceding temp1 */

          while (today:<:temp2) ; /* do nothing */

          return ("LAST TRADING DAY");
          /* alert sent to user */
}
```

3.4 Parser Design

The parser for calendar expressions parses the script and creates an efficient evaluation plan. The evaluation plan is a set of procedural statements in a high level programming language, e.g., C.

Two database procedures to simplify the creation of an evaluation plan are described next, followed by the parser algorithm.

Calendar Procedures

The first procedure is called *generate* and takes the arguments of start time (T_s), end time (T_e) and a list of numbers. *generate* creates a calendar of order-1 and its operation can be formally defined as:

$$generate(T_s, T_e; x_1; \cdots; x_n) \equiv \{(T_s, T_s{+}x_1), (T_s{+}x_1, T_s{+}x_1{+}x_2), \cdots, (T_s{+}\Sigma_{i=}^{n} x_i, T_s{+}\Sigma_{i=1}^{n} x_i{+}x_1), \cdots\}$$

In *generate*, calendar intervals are generated by using the list of numbers in a circular way till the end time is exceeded by an interval. This is illustrated by the following example:

$$YRS\_SINCE\_1987 \equiv generate(Jan\ 1, 1987; Jan\ 3,\ 1992; (365, 366, 365, 365))$$

$$\equiv \{(1, 365), (366, 731), (732, 1096), (1097, 1461), (1462, 1826), (1827, 1829)\}$$

where the second element in the calendar, $\{(366, 731)\}$, denotes that the second year, 1988, began 366 days from Jan 1, 1987 and ended 731 days from Jan 1, 1987. For simplicity, January 1, 1987 is taken as 1. Note that T_e is January 3, 1992 and this results in the last element in the set being (1827,1829) where 1827 is January 1, 1992 and 1829 is January 3, 1992.

The second procedure is called *caloperate* and it takes as arguments, a calendar, a list of numbers and an end time. $caloperate(C, T_e; (x_1; x_2; \cdots; x_n))$, where C is the calendar from which the new calendar is to be derived, would create a new calendar whose first interval is a union of the first x_1 intervals of calendar C, the second interval is the union of the second x_2 intervals of C and so on. The list is considered a circular list as in *generate*. *caloperate* is illustrated by the following example. If YEARS $\equiv (1, 365)$, then $caloperate(YEARS, *; 7)$, would give the calendar of weeks in the year since:

$$WEEKS \equiv caloperate(YEARS, *; 7) \equiv \{(1, 7), (8, 14), (15, 21), \cdots\}$$

As specified (1,7) is the union of the first 7 intervals of YEARS and (8,14) is the union of the second 7 intervals of YEARS. Here $*$ indicates an arbitrary end time (it must be less than T_e of YEARS). Similarly, if MONTHS $\equiv \{(1, 31), (32, 59), (60, 90), (91, 120), \cdots\}$, the QUARTERS of the year are given by $caloperate(MONTHS, *; 3) \equiv \{(1, 90), (91, 181), \cdots\}$.

An Algorithm for Parsing the Calendar Script

For lack of space an informal description of the parsing algorithm is provided, followed by examples. For every calendar expression in the calendar script, parsing is done from right to left:

- When a derived calendar is encountered, if it is not materialized, replace it by its derivation script. Replace all temporary calendars (variables in calendar scripts) by the appropriate calendar expressions.

- Factorize the resulting calendar expression. The objective of this step is to remove the parts of the expression that are redundant. To factorize an expression, the following rule is used. If the expression is of the form: $\{(X : Op_1 : Y) : Op_2 : Z\}$ where X, Y and Z are calendars, Op_i are *setops* and if the granularity of Y and Z are the same with the condition that $Z \in Y$, the expression is reduced to $\{X : Op_1 : Z\}$ except when $\{Op_1\ is\ \leq$ and $Op_2\ is\ <\}$. In the latter case, the expression is reduced to $\{X : Op_2 : Z\}$.

- Create the parse tree.

- Determine the smallest time unit in the expression, e.g., DAYS, MINUTES, so that all calendars defined in the expression can be expressed in these units. Also mark any calendar that is encountered more than once to avoid generating values of the calendar unnecessarily.

- Create the calendar evaluation plan based on the parse tree. The evaluation plan uses the procedures *generate*, *caloperate*, for loop statements and temporary variables. For efficient execution of the evaluation plan, a time interval must be chosen within which the values of all relevant calendars are generated. In some cases, choosing this interval may be simple (as in the example shown below). In other cases, this time interval may not be uniform for all nodes of the parse tree. In these cases, at each node (N) of the parse tree, a simple look-ahead determines whether the next node is a *selection* node. If this node is a *selection* node, the selection predicate determines the time interval within which values of calendars are generated at N.

The two examples presented below illustrate the parsing algorithm.

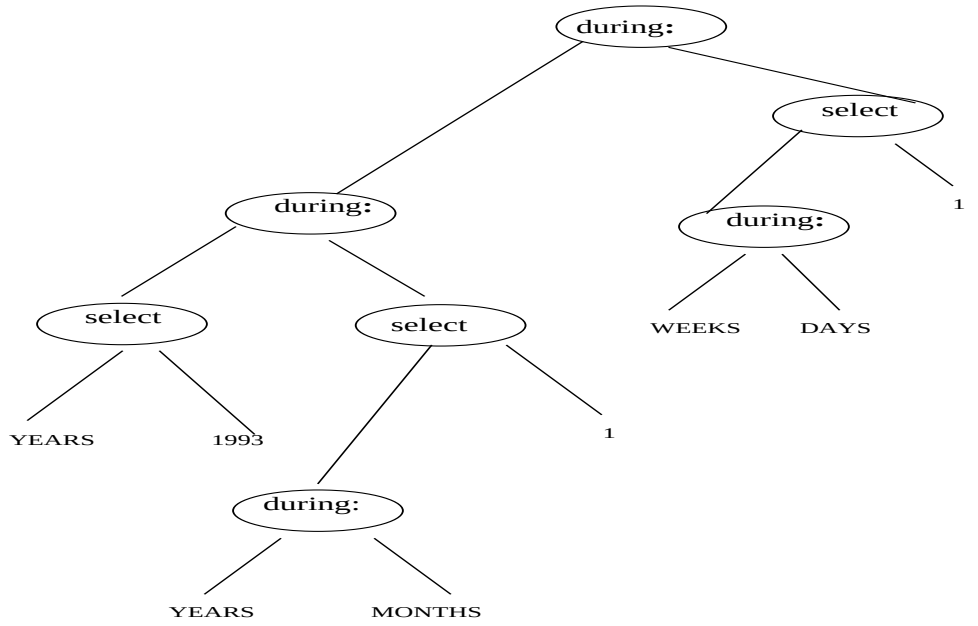
Example:

Consider the calendar expression that describes the mondays during January 1993.

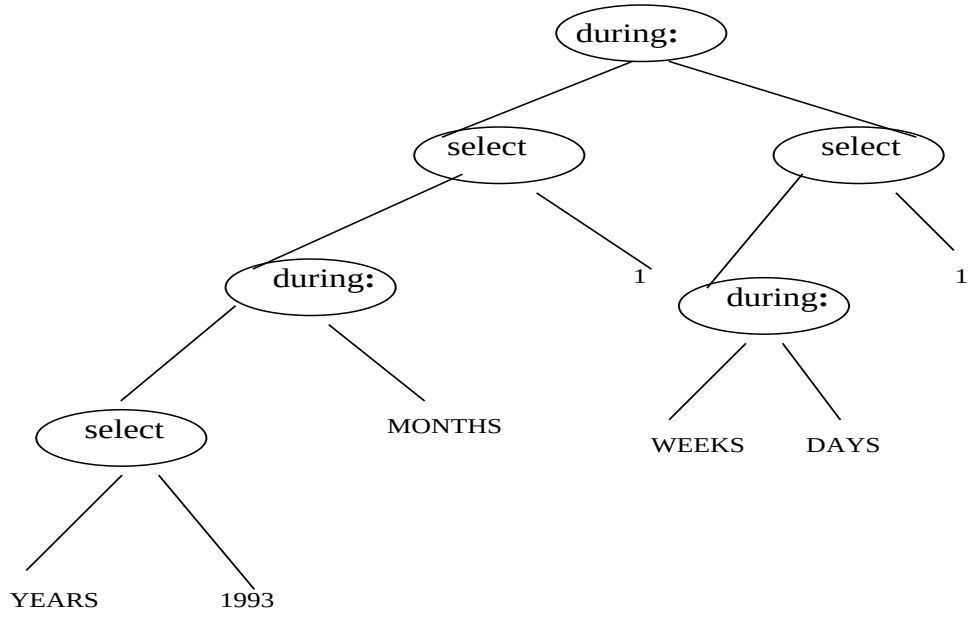
$$\{Mondays : during : Januarys : during : 1993/Years\}$$

Assuming $Mondays$ and $Januarys$ are predefined calendars, they are replaced by their *derivation-scripts*. We have:

$$\{([1]/DAYS : during : WEEKS) : during : ([1]/MONTHS : during : YEARS) : during : 1993/YEARS\}$$



(A)



(B)

Figure 2: Parse tree for "Mondays during January 1993"

Note that part of this expression, $\{([1]/MONTHS : during : YEARS) : during : 1993/YEARS\}$, is of the form $\{(X : Op_1 : Y) : Op_2 : Z\}$, where $X \equiv [1]/MONTHS$, $Y \equiv YEARS$ and $Z \equiv 1993/YEARS$. Since the granularity of Y and Z is the same, i.e., $YEARS$, and $Z \in Y$, this expression can be rewritten to $\{[1]/MONTHS : during : 1993/YEARS\}$.

Thus, the calendar expression reduces to:

$$\{([1]/DAYS : during : WEEKS) : during : [1]/MONTHS : during : 1993/YEARS\}$$

This expression can't be factorized any further because the granularity of $\{[1]/MONTHS : during : 1993/YEARS\}$ is different from WEEKS. The initial and factorized parse trees for the above calendar expression are shown in Figure 2(A) and 2(B) respectively.

Example:

Consider the calendar expression,

$$\{Third\_Weeks : during : Januarys : during : 1993/YEARS\}$$

where $Third\_Weeks \equiv \{[3]/WEEKS : overlaps : MONTHS\}$. Using the parsing algorithm we get,

$$\{([3]/WEEKS : overlaps : MONTHS) : during : [1]/MONTHS : during : 1993/YEARS\}$$

Note that the factorization applied in the previous example is shown here without explanation. Also note that this expression can still be factorized because we have an expression of the form $\{(X : Op_1 : Y) : Op_2 : Z\}$ where $X \equiv [3]/WEEKS, Y \equiv MONTHS$ and $Z \equiv \{[1]/MONTHS : during : 1993/YEARS\}$ and the granularity of Z is MONTHS. Thus, the calendar expression is rewritten to:

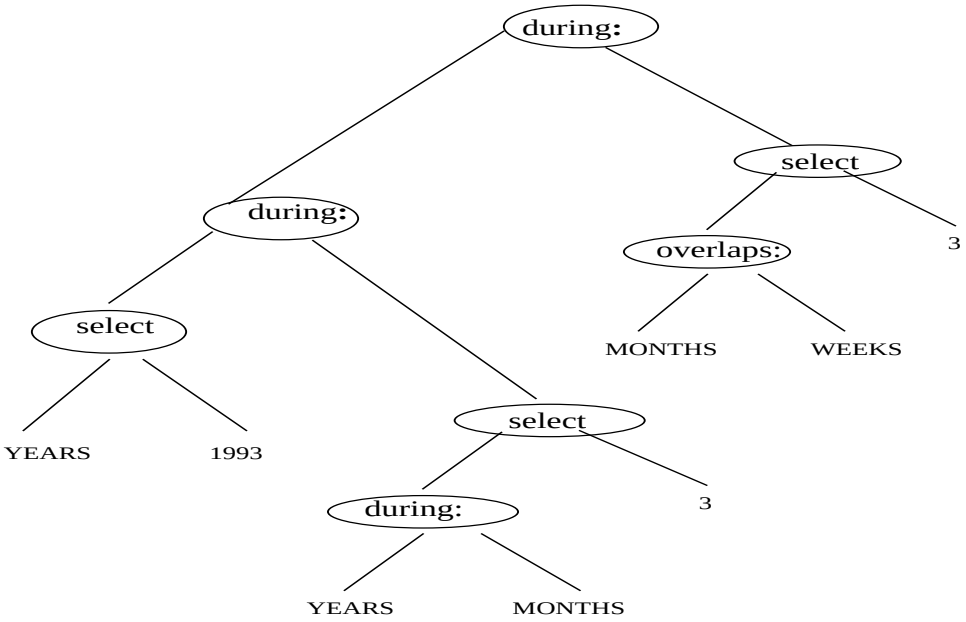
$$\{[3]/WEEKS : overlaps : [1]/MONTHS : during : 1993/YEARS\}$$

The initial and factorized parse trees for this expression are shown in Figure 3(A) and (B) respectively. In both parse trees, it is evident that for the expressions to be evaluated, calendars need only be generated for the time interval 1993.

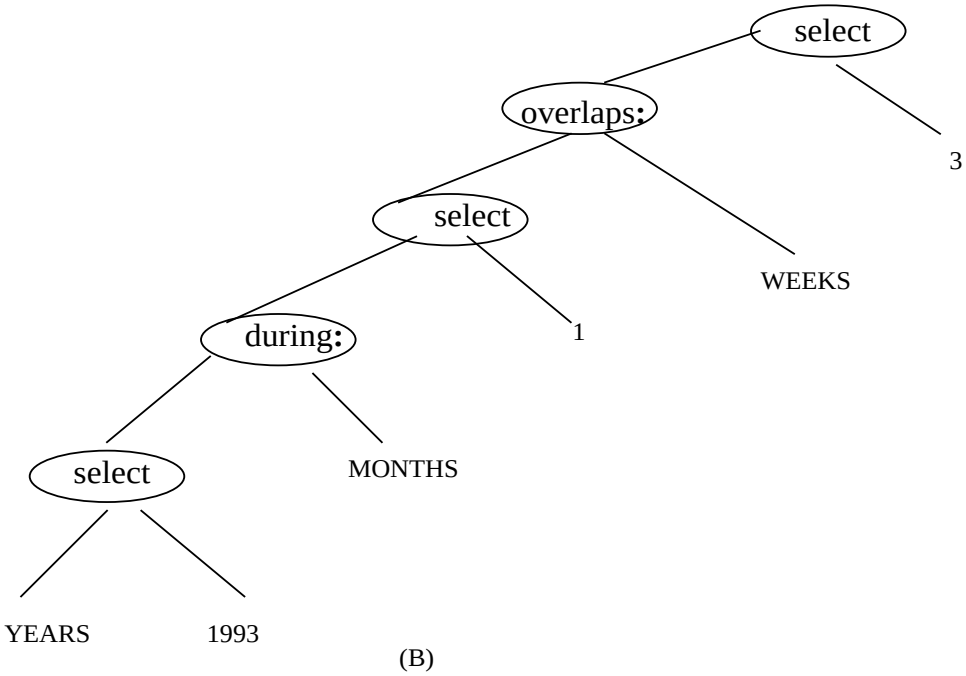
In this section, we described the calendar algebra and its implementation. The following section discusses the application of calendars to defining and implementing temporal rules.

4 Time-Based Rules

Rules are useful for testing integrity constraints, maintaining consistency, versioning, materialized views, updating derived data [SJGP90] and monitoring the database for specific events [De88].



(A)



(B)

Figure 3: Parse tree for "Third Week in January 1993"

Rules have been an active area of database research in the past few years. Examples of systems that support rules are Starburst[Wid92] and Ode[AG90]. There are several temporal aspects associated with rules. We say that a rule is temporal if at least one of the following cases is present: (a) the *On Event* clause involves temporal conditions, (b) the *Condition* includes temporal conditions, (c) the *Action* of the rule changes past or future states of the database. [EGS92] and [ESA93] addressed the issue of rule activation and processing due to retroactive and proactive transactions in active temporal databases. In this paper, we are concerned with temporal *On Event* clauses, and assume that the other parts of the rule can be handled by the underlying rule system.

The system of calendars described in the previous section can be used to incorporate temporal conditions for the triggering of rules. In this section, we show how a simple form of a temporal rule can be efficiently implemented. For the purpose of a concrete discussion, the Postgres Rule System is used as an example.

The Postgres rule system supports rules of the form *On Event where Condition do Action* where *Event* (as in many other systems that support rules) is a database operation - append /delete /retrieve /replace. The *Condition* is a Postquel clause that can be used to check the current or historical (with respect to transaction time) state of database objects. *Action* is a collection of Postquel commands with the additional feature that NEW and $CURRENT$ can be used instead of a tuple variable. The $CURRENT$ tuple refers to the tuple accessed during a retrieve, replace or delete, and NEW refers to the tuple that is to be appended.

The additional functionality discussed in this paper is the support for time-based rules of the form: *On Calendar-Expression do Action*, e.g., Every Tuesday do Proc_X $\equiv \{[2]/DAYS : during : WEEKS\}$ do Proc_X. Support for complex temporal conditions in rules is the subject of current research.

Overview

When a temporal rule is declared to the database system it is parsed by the parsing algorithm described in the previous section. The calendar expression, parse tree and evaluation plan of the new rule are stored in the database table, RULE-INFO. The next time point at which the rule should trigger is also evaluated and stored in the database table, RULE-TIME. RULE-TIME contains information on the next time point at which every temporal rule should trigger. RULE-TIME is probed by a daemon process, DBCRON, every T units of time to determine the temporal rules that trigger in the next T time units. DBCRON creates a main memory data structure that stores this information and is responsible for triggering rules at appropriate time points. It is modeled on the UNIX utility, CRON. An overview of temporal rule implementation is shown in Figure 4. The rest of the section describes the temporal rule implementation scheme in detail.

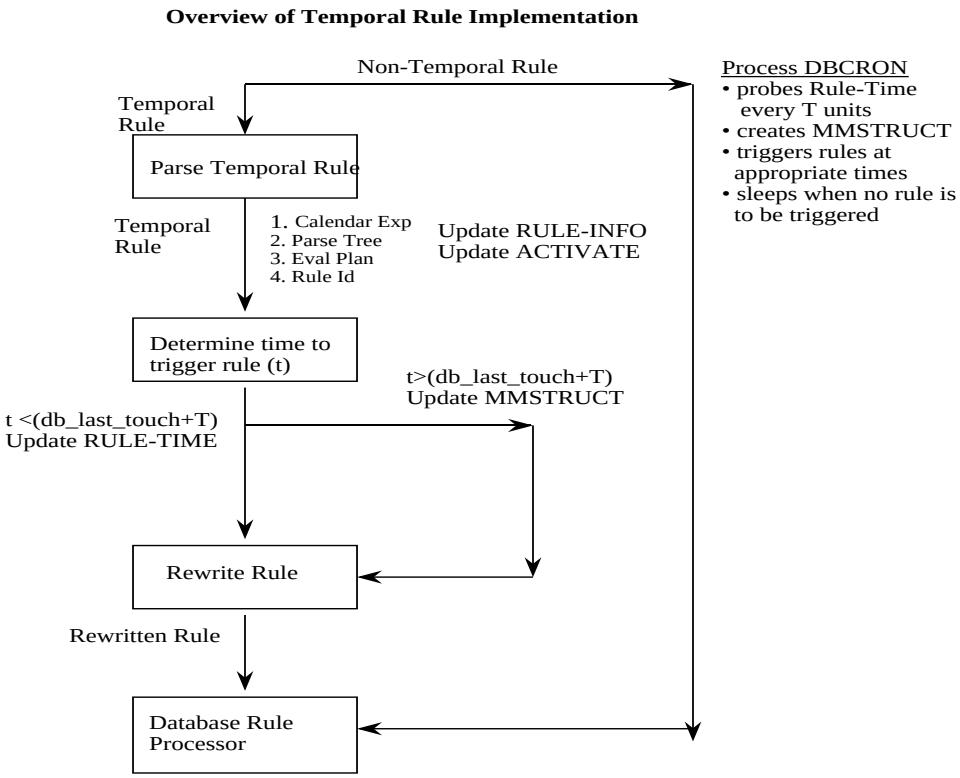


Figure 4: Temporal Rule Implementation

Implementation

Database Tables and Data Structures

This section describes the data structures used for processing rules. The processing algorithm is discussed in the next section.

Three database tables are used to maintain information about rules. RULE-TIME contains the next time point at which a rule is to be triggered. This table is maintained as a B+-tree index since range queries are needed to retrieve the rules that must be triggered in the next T units of time. RULE-INFO contains the calendar expression of the rule, the parse tree and the evaluation plan in addition to the rule id. The table, ACTIVATE contains just the rule id of all time-based rules that have been declared to the database and is used in the triggering of these rules. The structure of these tables are shown below:

```
RULE-TIME( time-point: abstime, rule-ids  : oid[])

RULE-INFO( rule-id:      oid, calendar-exp: text,
           parse-tree:   tree, exec-plan:    text)

ACTIVATE( rule-id: oid)
```

It is assumed here that the above tables are potentially very large and reside on secondary

storage. To provide for fast triggering of the rules, a main memory structure, MMSTRUCT, is maintained. MMSTRUCT is a main memory array that contains the rules to be triggered in the next T units of time. The array is sorted by time and contains a PTR which points to the next rule to be triggered. MMSTRUCT is created by probing RULE-TIME to retrieve the rules to be triggered in the next T units of time. A system variable, DB_LAST_TOUCH, stores the last time RULE-TIME was probed to update MMSTRUCT.

Rule Processing

When a rule is declared to the database, it is sent to the rule preprocessor which performs the following functions:

1. parses the rule to understand the calendar expression, create the parse tree and evaluation plan. RULE-INFO is updated with this information. If there is no calendar expression, the rule is released to the database rule processor without any change.

2. ACTIVATE is updated using the rule id.

3. The evaluation plan is executed to determine the first time, (t), the new rule will be triggered . If $(t) < (DB\_LAST\_TOUCH + T)$, it means that this rule is to be triggered before RULE-TIME is probed again. Thus MMSTRUCT is updated, resorted and the PTR reset. If $(t) > (DB\_LAST\_TOUCH + T)$, RULE-TIME is updated.

4. The rule is stripped of the calendar expression and is rewritten to have the following form:
   On Calendar-Exp do Action is rewritten to
   {*On retrieve to ACTIVATE where ACTIVATE.rule-id = $Rule*
   do Instead {Action; preprocess($Rule,1)}}.
   The new rule will trigger the *Action* when there is a retrieve to ACTIVATE. In addition to the *Action*, the procedure *preprocess*($Rule) is also triggered. *preprocess*($Rule,1) is the same procedure used for preprocessing the rule. The argument, 1, indicates that this rule already exists and thus only step 3 of the preprocessor algorithm is required. Note that $Rule refers to a rule-id that is resolved at execution time.

In the above discussion, it has been mentioned that RULE-TIME is probed periodically to determine the rules to be triggered in the next T units of time. The process which does this is called DBCRON. DBCRON is a daemon process that is always running in the background and is modeled on the Unix process, CRON. DBCRON has the following functions:

- probes RULE-TIME by performing a range query to retrieve the rules that are to be triggered in the next T time units.

- the retrieved set of rule ids and time points is sorted by time and stored in the main memory structure MMSTRUCT.
- triggers the rule pointed to by PTR at the appropriate time.
- based on a look-ahead, it determines how long it can remain dormant.

The algorithm for DBCRON is given below:

```
#define LAST_DB_TOUCH
#define T
dbcron(){
for (;;) { /* forever */
    while (time()=mmstruct[ptr].time)
         fork(trigger(mmstruct[ptr++].rule)
         /* trigger rule by sending retrieve to ACTIVATE and increment pointer */

    if (time() - LAST_DB_TOUCH >= T) {
                probe(); /* access RULE-TIME, create MMSTRUCT, set PTR */
                LAST_DB_TOUCH = time();
                lookahead(); /* determines how long it can sleep  MAX-SLEEP */
                }
    else
       { lookahead();
         sleep(MAX-SLEEP);
       }
  }/* for */
}/* dbcron */
```

5 Discussion of an Alternative Calendar System

This section provides a comparison between Soo's proposal for Mixed Calendar Query Language support [SS92] and our proposal. [SS92] define three temporal data types, *event*, *interval* and *span*. An *event* is an isolated instant in time, e.g., the time the option expired. An *interval* is a set of contiguous chronons with a minimum time element T_{min} and maximum element T_{max} such that $T_{min} \leq T_{max}$, e.g., July 1993. A span is defined as an unanchored duration of time that has a known length but the start time and end time are unknown, e.g., a WEEK. The proposal is based on set theory and the interval relationships defined in [All85]. Thus, all temporal queries supported by TQUEL and TSQL can be supported by this calendar system. The added advantage is that multiple calendars can be declared and queried.

In our work, only the *interval* data type is used for modeling calendars. Calendars are structured sets of intervals where each interval could also be a set of intervals. The depth of the calendar is defined by its order. An *event* is modeled by an interval in which $T_{min} = T_{max}$ and *spans* are

modeled by maintaining them in the relation CALTABLE. For example, the *span* 1 WEEK is maintained as a fixed number of SECONDS in this table. Relationships with other basic calendars such as MINUTES and HOURS are also maintained. Our proposal uses the collection interval algebra defined in [LMF86] and allows support for multiple calendars, queries supported by TQUEL and TSQL and also natural language time-based queries.

Soo's proposal requires the modification of SQL constructs while this proposal can be implemented in an extensible database without modifying the syntax of the query language. The calendar expression parser, procedures to generate calendars and procedures to evaluate calendar expressions are declared as operators to the extensible DBMS. Once declared to the DBMS, they can be used as part of the query language. This approach has the advantage that it can also be used with commercial systems including SYBASE and ORACLE to a limited extent. For example, ORACLE allows the creation of stored procedures in PL/SQL that can then be used as query language constructs in other PL/SQL procedures. Unfortunately it does not support direct integration of stored procedures with SQL.

6 Conclusion

In applications like financial trading, scheduling, manufacturing and process control, time based predicates in queries and rules are very important. There is also a need to define semantic sets of time points or intervals, referred to as calendars. This paper presented a system of calendars that allows specification of natural-language time-based expressions, maintenance of valid time in databases, specification of temporal conditions in database queries and rules, and user-defined semantics for date manipulation. The main contributions of this research can be summarized as follows:

- Proposal of a system of calendars in an extensible database that is useful for multiple calendar support and temporal queries.
- Definition of a simple set-based language used to define, manipulate and query calendars.
- The design of a parser for this language and the optimization of calendar expressions.
- A strategy for the implementation of time-based rules in an extensible database.

We are looking at the following areas for future research:

- Complex selection predicates: In the calendar language described, we allow only simple selections on the underlying calendars. Since regular time-series are associated with calendars, it is possible to modify the calendar language to allow selection predicates on the time-series

associated with calendars in addition to the calendars themselves. For example, a typical query to a stock price time series could be

Retrieve the time points at which the end-of-day closing prices for two successive days showed an increase.

The selection predicate in this case takes the form of a pattern: $\{S_t < Next(S_t)\}$. The language constructs should be general enough to incorporate numerical patterns in sequences. Implementation of this language also depends on efficient algorithms for searching numerical sequences.

- Implementation of Temporal conditions in Rules: This paper discusses the implementation of a simple form of a temporal rule. We are studying techniques to incorporate complex temporal conditions into rule events and conditions.

- The algorithm for evaluating calendar expressions does not expand materialized calendars when factorizing calendar expressions. Thus, a suboptimal evaluation plan may be created for certain calendar expressions. Generating efficient plans for calendar expressions with materialized calendars is the subject of current research.

References

[AG90] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and Data Model. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pages 36–45, May 1990.

[All85] J.F. Allen. Maintaining Knowledge about Temporal Intervals. In R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation*, pages 509–521. Morgan Kaufman Publishers, Inc., 1985.

[Cor87] Oracle Corp. ORACLE Terminal User's Guide. Technical report, Oracle Corp., 1987.

[CS93] R. Chandra and A. Segev. Managing Temporal Financial Data in an Extensible Database. In *Proceedings of the* 19^{th} *Int. Conf. on Very Large Databases, Dublin, Ireland*, September 1993.

[Dat88] C.J. Date. A Proposal for Adding Date and Time Support to SQL. *ACM SIGMOD Record*, 17(2):53–76, June 1988.

[De88] U. Dayal and et.al. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):51–70, March 1988.

[DW90] C.J. Date and C.J. White. *A Guide to DB2*, volume 1. Addison-Wesley, 3 edition, 1990.

[EGS92] O. Etzion, A. Gal, and A. Segev. Temporal Support in Active Databases. In *Proc. of the* 2^{nd} *Workshop on Information Technology and Systems*, December 1992.

[ESA93] O. Etzion, A. Segev, and G. Avigdor. On Rules in Temporal Databases. Technical Report LBL-33970, Lawrence Berkeley Lab, 1993.

[Gad88] S.K. Gadia. The Role of Temporal Elements in a Temporal Database. *Database Engineering*, 7(2):197–203, 1988.

[LMF86] B. Leban, D. McDonald, and D. Forster. A Representation for Collections of Temporal Intervals. In *Proceedings of the AAAI-1986 5th Int. Conf. on Artificial Intelligence*, pages 367–371, 1986.

[Mel90] J. Melton. Solicitation of Comments: Database Language SQL2. Technical report, American National Standards Institute, 1990.

[NA89] S. Navathe and R. Ahmed. A Temporal Relational and Query Language. *Information Science*, 49(2):147–175, 1989.

[RGM92] Snodgrass R., S. Gomez, and L.E. Jr. McKenzie. Aggregates in the Temporal Query Language TQUEL. Technical Report Technical Report TR89-26, University of Arizona, 1992.

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. of the* 10^{th} *Int. Conf. on the Entiry-Relationship Approach San Mateo, CA*, pages 205–229, 1991.

[RS93] E. Rose and A. Segev. TOOSQL - A Temporal Object-Oriented Query Language. Technical Report LBL-33855, Lawrence Berkeley Laboratory, 1993.

[SC91] Y.H.S. Su and H. M. Chen. A Temporal Knowledge Representation Model $OSAM^*$/t and its Query Language OQL/T. In *Proceedings of the* 17^{th} *Int. Conf. on Very Large Databases, Barcelona, Spain*, pages 431–442, September 1991.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, June 1990.

[Sno87] R. Snodgrass. The Temporal Query Language TQuel. *ACM Trans. on Database Systems*, 12(2), 1987.

[Sno90] R. Snodgrass. Temporal Databases: Status and research directions. *ACM Sigmod Record*, 19(4):83–89, December 1990.

[SS92] M. Soo and R. Snodgrass. Mixed Calendar Query Language Support for Temporal Constants. Technical Report TempIS No.29, University of Arizona, 1992.

[SS93] M. Soo and R. Snodgrass. Multiple Calendar Support for Conventional Database Management Systems. In *Proceedings of the Int. Workshop on an Infrastructure for Temporal Databases*, June 1993.

[Sto90a] M.R. Stonebraker. Ch. 7: Extensibility. In M.R. Stonebraker, editor, *Readings in Database Systems*. Morgan Kaufman, 1990.

[Sto90b] M.R. Stonebraker. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases.* Benjamin/Cummings Publishing Company, Inc., 1993.

[WD92] G.T.J Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proceedings of the 8th Int. Conf. on Data Engineering*, pages 584–593, February 1992.

[Wid92] J. Widom. The Starburst Rule System: Language Design, Implementation and Applications. *Data Engineering*, 15(4), December 1992.