Multiple Calendar Support for Conventional Database Management Systems

Michael D. Soo and Richard T. Snodgrass

Department of Computer Science
University of Arizona
Tucson, AZ 85721
{soo,rts}@cs.arizona.edu

Abstract

We propose a solution to the problem of supporting a time-stamp attribute domain in conventional relational database management systems. In contrast to existing proposals, which assume that a single interpretation of time is sufficient for all users and applications, we develop a general solution that supports multiple interpretations of time. The main concept underlying this proposal is that the universal aspects of time are separated from the user dependent aspects, at both the query language and the architectural levels. The user dependent aspects are encapsulated in calendars and calendric systems, each of which are extendible by local site personnel. In this way, the available time support can be customized to local requirements. We briefly describe modifications to SQL2 that support multiple calendars and calendric systems. These modifications reduce the complexity of the language while simultaneously increasing its expressive power. Finally, we describe a set of tools that aid in the generation of calendars and calendric systems. This work can be viewed as a limited but practical application of research into extensible database management systems.

Keywords: Time, temporal constants, architecture, extensibility, calendars, calendric systems, SQL, international software.

1 Introduction

The Lagunita workshop on future research in database systems identified the need for database management systems to support time [Silberschatz et al. 1990]. The workshop report notes that no consensus exists in support of any particular temporal model; the very nature of time implies different interpretations depending on the user's perspective.

In this paper, we propose a solution to the problem of supporting a time-stamp attribute domain in conventional relational database management systems (DBMSs). Our motivation is that time, perhaps more than any other data domain, is subject to user-interpretation—the DBMS must be capable of accommodating the interpretation of time applicable to a user or a site. Conventional relational database management systems do not address this problem at all; instead they impose a single interpretation of time at both the query language and architectural levels. We present a general solution that, in effect, internationalizes the time-stamp attribute domain provided by a DBMS. This work is also applicable to time representation in temporal database management systems [Snodgrass & Ahn 1986].

The focus of this paper is on architectural requirements for time value support, though we also address query language issues. We present a summary of modifications to SQL [Melton 1990] to support temporal data, and then develop an underlying system architecture. The main concept underlying the design is the separation of the universal aspects of time from those that are user dependent. The support for these aspects of time is partitioned at both the query language and the architectural levels. This separation allows customization of user-dependent time aspects by local site personnel.

The proposal employs a limited notion of *extensibility*. Architectural support is provided for addition and modification of the database management system components that impose a particular interpretation on temporal values. We propose a limited, but practical, application of the techniques proposed for extensible database management systems [Batory et al. 1988, Carey & Haas 1990, Carey et al. 1986, Haas et al. 1990, Stonebraker et al. 1990]. Our approach is related to that of extensible systems supporting abstract data types (ADTs). However, we believe that ADTs alone are inadequate for the temporal extensions developed here, and we argue why in Section 5.

The paper is organized as follows. Section 2 discusses general abstractions used to describe time and its use in society, motivating the basic data model we propose. Section 3 briefly describes SQL constructs supporting the concepts of Section 2. The primary focus of this paper, a system architecture supporting the proposed language features, is described in Section 4. The final two sections discuss areas of related research, summarize the contributions of this paper, and identify areas of future research.

2 Physical Time, Calendars, and Calendric Systems

This section describes the basic model of time we propose. We first examine how time is represented internally within the DBMS, and then introduce the concepts of calendars and calendric systems.

2.1 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 1992], but the details are not relevant here.

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.

2.2 Calendars

A *calendar* is a human abstraction of the physical time-line. One 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.

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

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*. 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.

Calendars have two types of characteristics, *intrinsic characteristics* that define the universal qualities of the calendar, and *extrinsic characteristics* that 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`, that 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 Gregorian calendar English is used to express dates in the United States, and French is used to express 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.

Properties, in conjunction with calendars, are crucial to supporting international use of the DBMS (c.f. [Digital 1991]). We have identified fourteen properties that are universal to all calendars [Soo et al. 1992]. Local adaptation of calendar properties is supported by defining 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.

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 parametrized by its properties and by supporting multiple calendars within the DBMS.

2.3 Calendric Systems

A calendric system defines the set of time values for an enterprise; it is the query language abstraction of the physical time-line. A calendric system is defined as a collection of calendars where each calendar is defined over non-overlapping periods of time, termed *epochs*. It is possible, and likely, that a calendric system has gaps in its time-line that are not covered by any calendar.

Figure 1 illustrates a single calendric system, the Russian calendric system. The figure shows the physical time line divided into a sequence of epochs. In the figure, the physical time-line is not shown to scale.

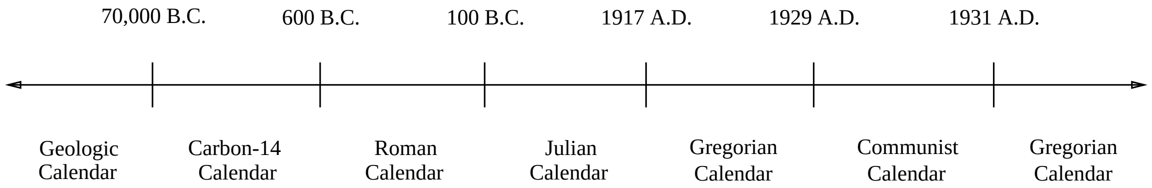

Figure 1: The Russian Calendric System

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 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.

2.4 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.

3 SQL Language Modifications

This section describes calendar independent language modifications to SQL. We add data types and operations to SQL that do not depend on the semantics of a particular calendar. In addition, we describe language constructs for calendric system and property table specification. The presentation is a significantly abridged description of the query language modifications we propose elsewhere [Soo & Snodgrass 1992].

The specific language being modified is SQL2, the most recent standardization of the SQL language, currently under development. SQL2 extended the previous SQL standard with several new features including time data types [Melton 1990]. We eliminate the temporal extensions proposed in SQL2 and incorporate our own. (It can be shown that our proposal subsumes the temporal extensions we replace.) We do not assume detailed knowledge of SQL2, only that the reader is familiar with the general concepts of SQL. In this paper, a reference to "SQL2" means the SQL2 language, while a reference to "SQL" implies a generic version of the language.

Unless specifically noted, we use the familiar Gregorian calendar in examples, and rely on the reader's intuition until the necessary language constructs are defined.

3.1 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. There are three important temporal notions, moments in time, periods in time, and durations of time. We define three new time-oriented data types, events, intervals, and spans corresponding to each of these notions, respectively. Example queries involving these data types are shown in Table 2.

Data Type	*Example Query*
event	"When was Ed hired?"
interval	"Did Ed work for Alice during 1956?"
span	"How long was Ed in school?"

Table 2: Examples of Time-Oriented Queries

An *event* is an isolated instant in time; it is said to occur during some chronon t. For example, if the implementation fixes the granularity of a chronon as one second, then an event is known to happen during a particular second, and two events which occur during a single second are assumed to happen simultaneously.

Specification of event values is done with a string-like notation. An `event` constant is syntactically delimited by vertical bars ("`|`"). The string of characters contained within the bars is interpreted to be an event constant defined by a calendar. As examples of event constants, `|Midnight December 31, 1991|` is a valid event constant 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.

An *interval* constant is syntactically delimited by square brackets ("`[]`"). The string of characters contained within the square brackets is interpreted to be an interval constant. For example, `[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.

We note that interval constants are not restricted to the form `[`*starting_event*`,` *ending_event*`]`. 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.

A *span* defines a duration of time, that is, a period of time with 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 a positive or negative duration of time. Span constants are syntactically delimited by percent signs ("%"). For example, `%1 week%`, `%2 years%` and `%-19 seconds%` are valid span constants.

The duration of a span is either context dependent or context independent. A *fixed span* represents the same duration independent of its usage. 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-one 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. 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 duration of `%1 month%`. Both interpretations are valid, and neither should be excluded by the DBMS.

The meaning of `%1 month%` is specific to the calendar that defines it. It is left to the calendar to specify the appropriate semantics. Generality and usability are increased since the calendar is free to ascribe any appropriate meaning.

We 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.

As an example of using the temporal data types, consider an employment database containing personnel information. We would like to store 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 store the employee records. The SQL statement to create a base relation with this schema is shown below.

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

3.2 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 session defaults or for individual data items.

3.2.1 Calendric System Specification

Calendric systems are specified by lexical scope in a sequence of SQL statements. The scope of a globally declared calendric system is all statements up to but not including the next global declaration of a calendric system. Global calendric systems are declared by a `declare calendric system` command. Conversely, for a particular item, a local declaration can be used to override the declaration at the global scope. Local declarations are made using an `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 2: Example of Calendric System and Property Selection

Figure 2 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 that is applies if no calendric system is globally declared.

3.2.2 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.

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 2, the procedure `set_x_properties` contains a command that 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 time-stamps are being converted for the age attribute.

As mentioned in Section 2.2, 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.

3.3 Built-in Functions

It is convenient to have simple mechanisms for data conversion and manipulation. To accomplish these tasks, we have defined nine built-in functions. These functions are categorized as either data constructors (e.g., `interval` to compose an interval from two events), data deconstructors (e.g., `begin` to return the starting event of an interval), or miscellaneous functions (e.g., `first` to return the prior of two events). We emphasize that these functions are calendar independent. Additional calendar specific functions may be defined by a calendar. For example, a function to extract the month field of an event could be implemented for the Gregorian calendar.

3.4 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.

We extended the basic arithmetic operators (`/`, `+`, `-`, `*`) for events, intervals, and spans. Our design goals were to maximize orthogonality whenever possible and to overload existing operators, thereby minimizing the complexity of the language. Expressions with intuitive semantics such as *event* `+` *span* are allowed while expressions with unintuitive semantics such as *interval* `+` *event* are not.

3.5 Comparison Expressions

An important motivation for incorporating time values into the query language is to determine the temporal relationships between objects. For example, for the employee relation of Section 3.1, one might be interested in who was hired during a particular year, or given two employees, who has more years of service.

Temporal comparison operators allows one to determine these relationships. A set of such operators was defined for the event, interval, and span data types. We modified the semantics of existing time comparison operators, `overlaps`, `<`, `=`, `>`, and `=`, and added three new comparison operators, `precedes`, `meets`, and `contains`. The operator set was derived by examining the comparison operators of other temporal extensions to SQL [Ariav 1986, Ben-Zvi 1982, Navathe & Ahmed 1989, Sarda 1990]. It can be shown that our operators subsume each of these proposals, and, in fact, our operators can express any possible relationship between any of the temporal data types. Full details are provided elsewhere [Soo & Snodgrass 1992].

3.6 Aggregate Functions

There are six SQL aggregate functions which operate on sets of values. We extended the semantics of five of these functions, `count`, `sum`, `avg`, `max`, and `min`, to accommodate the temporal data types. The sixth aggregate function `count(*)`, which determines the number of rows in a table, is not relevant to the discussion. As with arithmetic expressions, we identified all applications of these functions to the temporal types which 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 values is not.

3.7 Summary

Calendar independent constructs were added to SQL to support new time data types, temporal built-in functions, temporal arithmetic and comparison, and aggregate functions. In addition, we defined mechanisms for calendric system and property table selection, increasing the expressive power of the query language through calendar specific operations.

Interestingly, the complexity of the language has actually decreased after subtracting the previous temporal support and adding our temporal modifications. Seventeen keywords dealing with Gregorian calendar constructs were deleted, and nine keywords for calendar independent constructs were subsequently added. This simplification is primarily due to the fact that, in SQL2, calendar specific constructs are implemented as keywords in the query language, while we support them through extensions of the query language, in the form of calendar defined functions. In addition, we significantly increase the expressive power of the language, as SQL2 only supports a single calendar, the Gregorian calendar, and a single language, English.

4 System Architecture

The language constructs just described provide their expressive power through extensibility: local users can define new calendars and calendric systems, reference them in queries, and can also alter the properties of calendars. Supporting this extensibility requires a more open architecture than that employed in current database management systems.

In this section, we describe the architectural modifications a conventional relational database management system needs in order to support the language constructs of the previous section. We discuss only the components of the database management system which must be modified or extended to support our data model. Essential, but non-germane, components such as concurrency control, recovery management, and storage access methods are omitted.

4.1 Overview

Figure 3 contains a diagram showing the major components of the system. Each box in the figure represents a component of the system; a solid line arrow from one component to another indicates that the former utilizes services provided by the latter. Data structures (non-procedural components) are shown as ovals, and a dashed line arrow indicates a data structure contains a reference to another component. The figure shows the following components.

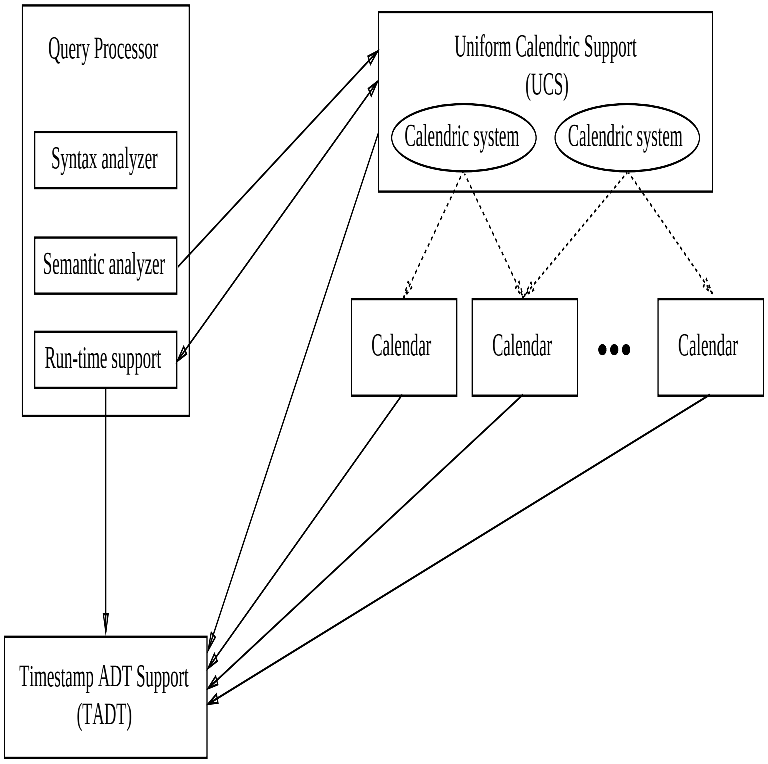

Figure 3: System Architecture Overview

- Query processor—a conventional query processing system extended to support the new temporal constructs.
- Uniform calendric support (UCS)—an interface that manages access to the services provided by calendars. Each calendric system is defined as a collection of data structures within the UCS. Within the architecture, calendric systems have no procedural component; they merely provide a mechanism for accessing the services exported by their calendars.
- Calendar—a set of routines and tables implementing calendar dependent operations. We note that, as shown in Figure 3, calendars can be shared by multiple calendric systems.
- Time-stamp ADT (TADT) support—a set of routines encapsulating operations on physical time-stamps. The TADT implements all temporal operations that do not require interpretation by a calendar.

In Figure 3, it is readily apparent that the support for calendar dependent operations is partitioned from the support for calendar independent operations. The UCS is responsible for executing, by using the appropriate calendric system, all calendar dependent operations. The TADT, on the other hand, provides calendar independent operations, specifically, built-in, arithmetic, comparison, and aggregate operations on time-stamps.

This distinction is the key aspect of our approach. We isolate operations requiring calendar interpretation by encapsulating them within a calendar, and provide calendar independent operations elsewhere. This allows the architecture to support extensibility and interchangeability of calendric systems and calendars.

As an example, consider the arithmetic operation of computing the sum of two span values. Variable spans require calendar interpretation while fixed spans do not. Therefore, the TADT exports an operation `fs_add_fs` which adds two fixed spans, while a calendar must provide an operation for adding variable spans to fixed spans, `vs_add_fs`, and an operation for adding variable spans to variable spans, `vs_add_vs`. The UCS exports a generic span addition operation, `s_add_s`. The query processor invokes `s_add_s` whenever a span addition operation is performed. `s_add_s` queries the TADT to determine if its parameters are variable or fixed spans, then calls the appropriate TADT or calendar routine.

Extensibility of calendric systems and calendars is central to our architecture. We therefore support definition of calendric systems and calendars by local site personnel. A base version of the DBMS will likely include several calendars and calendric systems, and these calendars and calendric systems will be adequate for most users. In addition, we anticipate a market for customized calendars and calendric systems, with third party vendors specializing in developing such solutions.

Calendar and calendric system definition will be performed by a *database implementor* (DBI), a person with sufficient knowledge of the internal workings of the DBMS to implement calendar-defined functions and routines [Richardson & Carey 1987]. The DBI is responsible for supplying the supporting components of calendars and calendric systems and generating the resulting database management system. To simplify this task, we provide a toolkit that accepts calendar and calendric system specifications provided by the DBI and composes the DBMS from those specifications and preexisting components. A preliminary design of the generation toolkit is presented in Section 4.7.

We continue by describing the system components shown in Figure 3 in more detail. We use pertinent examples, such as supporting variable span operations, to illustrate the design.

4.2 Time-stamp ADT Support

As previously mentioned, the TADT is responsible for all temporal operations that do not require calendar interpretation. This includes all operations on event, interval, and fixed span values plus auxiliary operations for time-stamp manipulation. The representations of all such values as stored in the database are necessarily calendar independent [Dyreson & Snodgrass 1992]. Operations involving variable spans require calendar support and are not implemented by the TADT.

The operations contained in the TADT derive directly from the query language operations described in Sections 3.3–3.6. Table 3 shows the types of operations supported by the TADT along with a count of routines for each operation type. A detailed description of each routine is provided elsewhere [Soo et al. 1992].

As shown in Figure 3, the operations provided by the TADT are used by the run-time support of the query processor, the UCS, and any calendars defined in the system. The query processor invokes the TADT to execute all built-in, arithmetic, comparison, and aggregate operations involving non-span operands. A calendar calls the time-stamp creation routines of the TADT while computing

Operation	*Number of Routines*
Built-in operations	9
Arithmetic operations	11
Comparison operations	13
Aggregate operations	30
Auxiliary operations	6

Table 3: Time-stamp ADT Operations

a time-stamp equivalent to a temporal constant encountered in the input. The UCS invokes the TADT to execute fixed span operations, as we discuss below.

4.3 Uniform Calendric Support

The UCS provides a generic interface to all calendar defined services. Table 4 lists the types of operations performed by the UCS. The UCS is invoked by the query processor to activate and deactivate calendric systems and properties, convert temporal constants to time-stamps, convert time-stamps to temporal constants, resolve calendar defined functions, and execute all span operations. It maintains data structures defining calendric systems, and invokes calendar operations on behalf of the query processor. In general, the UCS is responsible for executing any operation which could possibly be calendar dependent.

Operation	*Number of Routines*
Span arithmetic operations	9
Span comparison operations	3
Span aggregate operations	15
Constant translation	3
Time-stamp translation	3
Function binding	1
Calendric system activation	2
Property activation	4

Table 4: Uniform Calendric Support Operations

Specifically, event and interval computations are calendar independent. Hence, operations on events and intervals, once translated into time-stamps, can be executed directly by the TADT. For example, event values, i.e., time-stamps in the physical representation, do not require a calendar interpretation; their time-stamps completely describe their values. Therefore, operations on event values, such as *event* `precedes` *interval*, are simple time-stamp manipulations that can be performed directly by the TADT. However, for operations involving span values, it is not known if the operation is calendar independent until the time-stamps of the operands are examined. Variable spans require calendar support while operations involving only fixed spans do not. The query processor is not capable of resolving this since, in the type system of the query language, variable spans and fixed spans are the same type. Therefore, the TADT provides a routine `is_variable_span` which determines if a span is variable or fixed. This is consistent with our design of the TADT as an abstract data type for time-stamps. The UCS calls this routine to determine if any operand is variable and, if so, invokes a calendar to perform the given operation. Otherwise, the operation is passed to the TADT which performs the computation. Interestingly, over two-thirds of the UCS routines are these simple "traffic-control" routines related to variable spans. We will describe in more detail the interface between the UCS and a calendar in Section 4.6.

We describe the UCS operations supporting calendric system selection, property activation, constant translation, time-stamp translation, and calendar defined function binding in the next section.

4.4 Query Processing Subsystem

In our limited context, the query processing system is responsible for invoking the UCS when calendar support might possibly be required for a certain operation and for invoking TADT routines when executing operations that are clearly calendar independent.

The TADT solely provides run-time operations such as temporal arithmetic; the UCS provides operations that support the query processing system at both compile-time and run-time. The query processing system invokes the UCS during semantic analysis to perform calendric system binding, and type checking and binding of calendar defined functions, such as `month_name_of`. During query execution, the query processor invokes the UCS to translate temporal constants into time-stamps, translate time-stamps into output strings, and activate and deactivate calendric systems and properties. We note that the syntax analyzer in the query processing system does not require either UCS or TADT provided services. The syntax analyzer must, of course, be able to recognize and parse the language constructs described in Section 3.

We continue by describing, in detail, the modifications to the semantic analyzer and the run-time system of the query processor.

4.4.1 Semantic Analysis

Semantic analysis is responsible for ensuring the semantic correctness of the query, that is, such tasks as type checking and binding of names are performed by the semantic analyzer. It is preferable to perform these tasks at compilation time since programmer intervention is normally required when static semantic errors occur. We have attempted to maximize the amount of semantic checking possible at compile-time, though some semantic checking must be delayed until run-time for flexibility.

The binding of calendric systems and calendar defined functions occurs at compile-time. This is made possible by the static scoping of `declare calendric system` commands as discussed in Section 3. When this command is parsed, the semantic analyzer invokes the UCS to verify that the named calendric system actually exists. If so, the UCS records the named calendric system as being the currently active calendric system. Later, when a calendar defined function such as `month_name_of` in Figure 2 is encountered, the semantic analyzer invokes the UCS to bind that function to its implementation. The UCS verifies that the function is defined via the current calendric system, and performs type checking on the function's parameters. In Figure 2, this directs the UCS to use the `russian` calendric system when resolving the function `month_name_of`.

Other minor extensions to the semantic analyzer are required, including type evaluation and checking of temporal arithmetic and comparison expressions and of related constructs such as procedure parameters. Such extensions do not require UCS or TADT support.

4.4.2 Run-time System

Several aspects of our language proposal cannot be satisfied by compile-time resolution and, therefore, require run-time resolution. Specifically, temporal constants cannot be evaluated at compile-time, in contrast to arithmetic or string constants. This is because the meaning of a temporal constant such as `|1 January 1900|` depends not only on a calendar, but also on the set of active calendar properties, which cannot be determined at compile-time. For example, consider the declaration of cursor `x` in Figure 2. At compile-time, the semantic analyzer knows that the `russian` calendric system is to be used to evaluate the constant `[1975]`. However, SQL semantics state that the query is not evaluated until the cursor is opened, that is, until the procedure `open_x_cursor` is called, and, in general, it is impossible to tell what the active set of properties will be at that time. In this case, the semantic analyzer associates with `[1975]` the name of its calendric system. When the cursor is opened at run-time, the query processor retrieves the name of the calendric system and invokes the UCS to activate it. If a property table were specified for `[1975]` via a `with` clause, the properties in the property table would also be activated. The constant is then passed to the UCS for evaluation.

While this may delay the detection of errors, we feel that this flexibility is desirable. As discussed in Section 2.3, properties are, by nature, extrinsic to a calendar and are most appropriately stored extensionally where they can be manipulated and changed. Since the extension of a relation is only known at run-time, compile-time evaluation of temporal constants is precluded.

The run-time system of the query processor utilizes services exported by both the TADT and the UCS. When performing operations that could possibly require calendar support, the run-time system invokes UCS provided routines, and when performing operations that are clearly calendar independent, the run-time system invokes TADT provided routines.

This partitioning has been mentioned before, but we note here that the query processor is able to determine the correct module to invoke based on typing information and the kind of operation being performed. For example, calendric system and property manipulation statements such as `declare calendric system`, `set properties`, and the `with` and `as` clauses are supported by UCS routines. Furthermore, operations such as *span* `+` *span* might require calendar support depending on if either operand is a variable span. Such operations are routed to the UCS which makes the determination about the type of spans being added then either invokes the TADT if both spans are fixed or invokes a calendar any of the spans is variable. Conversely, operations such as *event* `-` *event*, *event* `precedes` *interval*, and `intersect(`*interval*`,` *interval*`)` which do not involve calendar dependent operands are passed directly to the TADT for evaluation.

4.5 Calendric System Data Structures

As previously mentioned, a calendric system is represented by data structures within the UCS; calendric systems contain no procedural components. From an architectural standpoint, a calendric system exists solely to integrate calendars and to supply a mechanism for accessing the facilities those calendars provide. As such, static data structures identifying calendars and the services exported by those calendars are all that is needed to implement a calendric system.

A calendric system data structure contains three components, the name of the calendric system, the set of calendars and epochs defined for the calendric system, and a list of routines. The list of functions is the union of the set of functions defined by each calendar named in the calendric system.

When a temporal constant is encountered in a query, the UCS must select a calendar of the calendric system to translate the constant into a timestamp. This is necessary since several calendars within the same calendric system may be capable of translating a given constant. We describe a DBI-controlled mechanism for calendar selection elsewhere [Soo et al. 1992].

4.6 Calendars

The calendar is the most critical component of the architecture. It represents the local adaptation of temporal semantics within the architecture, and so the majority of its contents must be provided by the DBI. These contents include calendar unique functions, routines supporting temporal constant evaluation and time-stamp evaluation, and calendar dependent aggregate, arithmetic, and comparison operations. These routines constitute the services the calendar exports to the UCS.

Table 5 identifies the operations that must be programmed as part of a calendar implementation. Detailed descriptions of these routines are contained elsewhere [Soo et al. 1992]. We note that most of these operations involve variable spans. If the calendar does not define any variable spans, then these functions are not required. Only the remaining eight translation routines are required to define a calendar. If only one variable span is defined, then the variable span routines can be quite simple. This is the case with our initial description of the Gregorian calendar which has a single variable span, `month`. When multiple variable spans are defined, then each routine must contend with all; some must handle the more complex combinations of two variable spans.

Operation	*Number of Routines*
Constant translation	3
Time-stamp translation	3
Auxiliary translation operations	2
Variable span aggregate operations	21
Variable span arithmetic operations	14
Variable span comparison operations	8
Variable span to fixed span conversion	1
Calendar specific functions	?

Table 5: Calendar Operations

Constructing calendar routines may be difficult for the DBI. Consequently, whenever possible we have identified common processing that must be present in all calendars, and shifted that code into the UCS to minimize the DBI's programming effort. Shifting processing to the UCS is made possible by using table-driven algorithms. Calendars provide tables describing data formats and field values to the UCS; the UCS uses this information to interpret input data or construct output data. In particular, in Section 2.2 we stated that properties are used by calendars to adapt to local requirements. At the query language level, properties are used to parametrize calendars, and property values affect the result of calendar operations. However, at the architectural level, our goal is to simplify the implementation of calendars as much as possible. Consequently, we have moved the interpretation and application of property values out of the calendar and into the UCS. Calendars are not required to interpret property values directly, and whenever possible, the UCS pre-processes the data to apply the effects of property values.

For example, Figure 4 shows a flow diagram for the processing that occurs when a time-stamp is converted into an output string. (This processing would occur when a time-stamp is retrieved in a `fetch` statement returning temporal attributes.) The query processor invokes the UCS to convert the retrieved time-stamp into an output string for assignment to a procedure parameter. Boxes in the figure denote actions which, in turn, represent UCS or calendar calls. Ovals represent data items used in or generated by the processing. Most actions present in the figure are implemented in the UCS; calendar routines are represented by broken outline boxes, and we note that there are only two such boxes.

Figure 4 is illustrative of how table-driven algorithms are used in the UCS. Consider the 8-

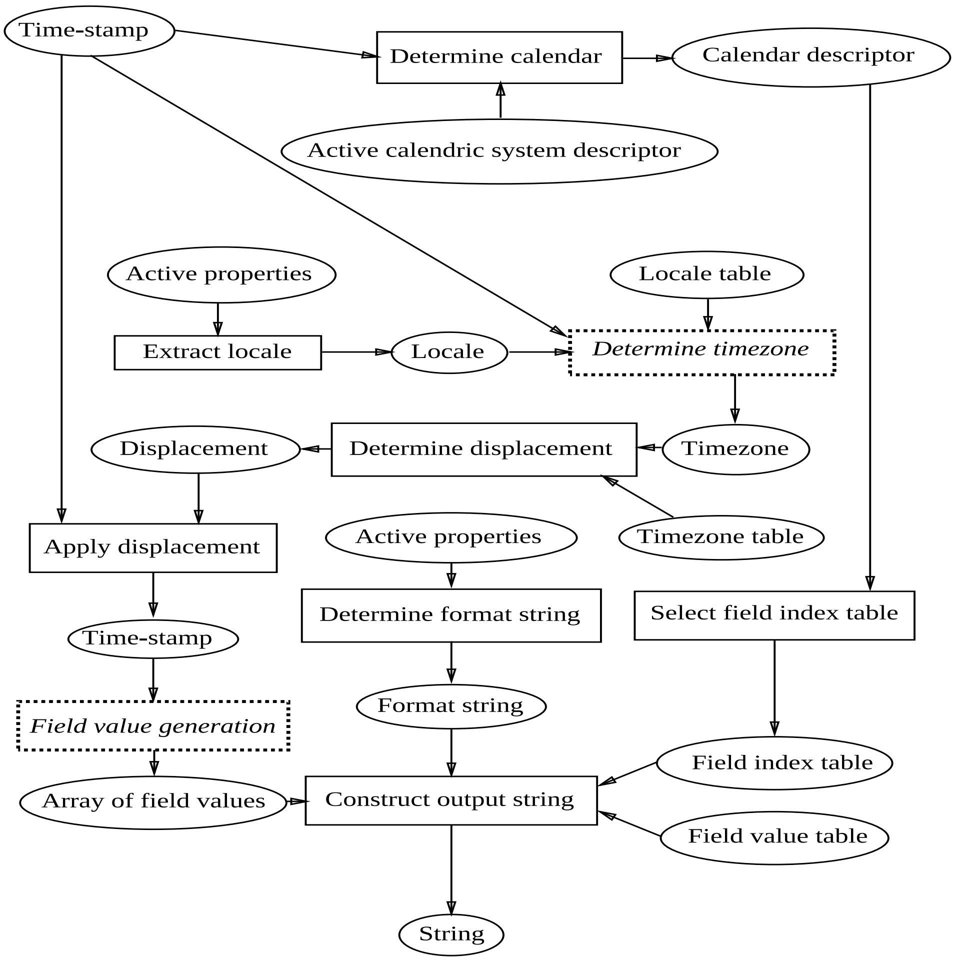

Figure 4: Time Value Retrieval

byte time-stamp `623446874400325001` stored in the *when_employed* attribute of Figure 2. For this time-stamp, the `american` calendric system is consulted, and the time-stamp is determined to be associated with the `gregorian` calendar. Translation begins by performing local processing to determine the correct timezone. The UCS checks the value of the *locale* property which names the location of interest. In this case, the locale is `Tucson,Arizona`. The *locale table* is then queried to determine the timezone in which Tucson is located. The locale table indicates that Tucson, and most of the state of Arizona, is always on Mountain Standard Time (MST), making the calendar operation *determine timezone* particularly simple. The time displacement for MST, which is 7 hours behind Greenwich Mean Time, is retrieved and subtracted from the original time-stamp.

Generation of the final output string begins by invoking the `gregorian` calendar to generate the *array of field values* shown below.

Index	*Value*
0	`2`
1	`0`
2	`1975`
...	...

The array of field values is simply an unparsed version of the time-stamp. The content of the array of field values is described by the following *field index table*, which is provided by the calendar.

Index in Array of Field Values	*Field Name*
`0`	`day`
`1`	`month`
`2`	`year`
...	...

The field index table associates indices in the array of field values with the components of a temporal constant. The field index table indicates that the day component, `2`, is found in the zeroth element of the array of field values, the month component, `0`, is found in the first element, and the the year component, `1925`, is found in the second element.

The UCS determines the format of the final output string by retrieving the *output format string* from the current property set. We schematically represent the output format string as follows.

<`month, english_month_names`>␣<`day, arabic_numeral`>`,`␣<`year, arabic_numeral`>

The format string lists the fields in the order that they are to appear in the output. The component associated with each field is either a *field value name table* or the name of a routine that computes the field's string. Field value name tables and routines are calendar provided. For example, the `english_month_names` field value name table is shown in Figure 6. The UCS retrieves the string `January` since the array of field values entry for `month` is `0`, and `January` is contained in the zeroth entry of the field value name table. The UCS iterates over the fields of the output format string adding one field to the output string on each iteration. The resulting string, `'January␣2,␣1975'`, is returned as the value of the *when_employed* attribute.

Index	*Field Value Name*
0	`January`
1	`February`
...	...
10	`November`
11	`December`

Table 6: `english_month_names` Field Value Name Table

This example illustrates how much of the the processing has been moved out of the calendar and into the UCS and TADT. In particular, the calendar need only provide two routines, one for determining the timezone (which is primarily table lookup) and one for converting an adjusted time-stamp into an array of field values. The UCS does the rest of the work of creating the associated string. Minimizing each calendar's responsibility is important since the UCS and TADT will be implemented once, by the DBMS implementor, whereas the calendar's implementation will be the responsibility of the DBI.

4.7 Generating Calendars and Calendric Systems

To ease the task of integrating new calendric systems and calendars into the DBMS, we have designed a toolkit that generates calendric system data structures and some of the components of calendars from higher-level specifications.

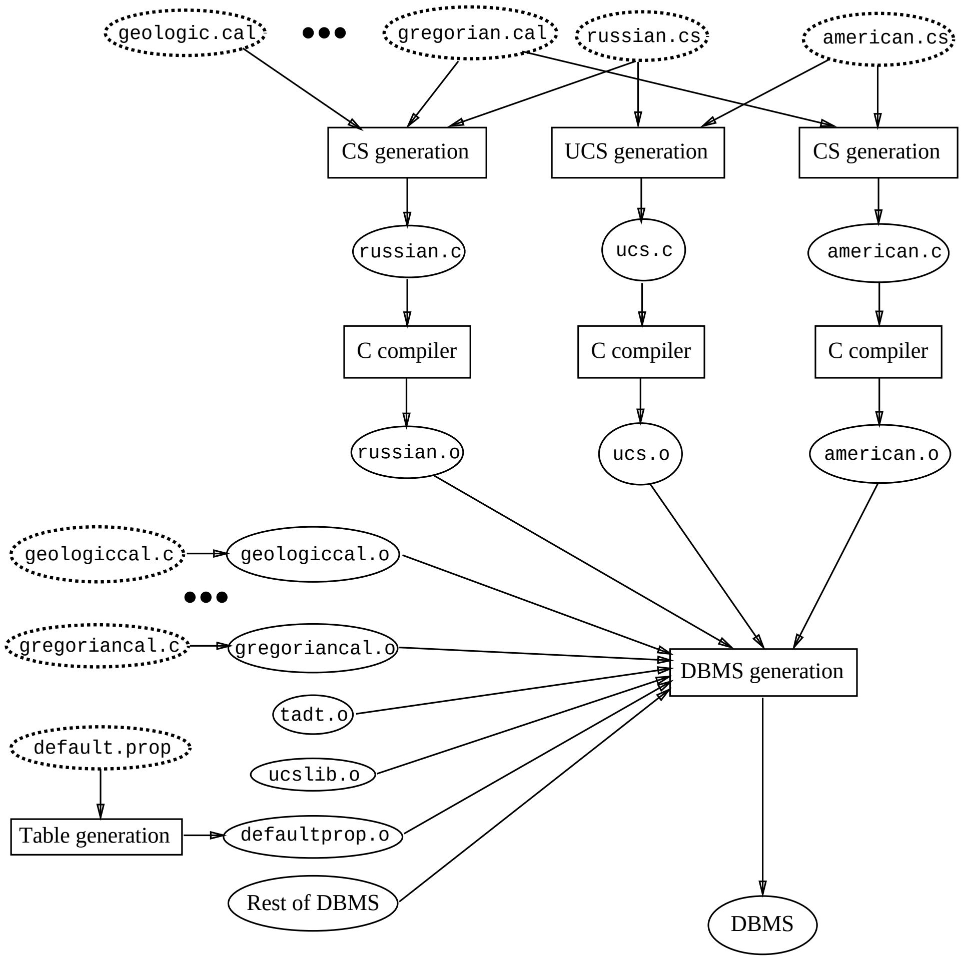

Figure 5: Generating an Example DBMS

An example of using the toolkit is shown in Figure 5. Boxes in the figure represent actions of the toolkit or a source language translator. Ovals represent files either read or generated during the processing. Specification and input files that are created by the DBI are shown as broken line ovals. Files ending in `.cal` contain calendar specifications. Similarly, `.cs` files contain calendric system specifications, and `.c` files contain C language programs. A calendar specification file contains a list of procedure signatures supported by the calendar. The toolkit verifies that this list includes all required routines as described in Section 4.6. A calendric system specification contains a list of calendars and epochs defined for the calendric system. The calendars appear in their input order from highest to lowest. Finally, C program files contain the implementations of procedures with signatures in calendar specification files.

Our eventual goal is to generate, as much as possible, calendar `.c` files from the declarative specifications contained in `.cal` files, thereby minimizing the DBI's programming effort. As previously described, we have taken a step toward this goal by simplifying the calendar routines. Further enhancements are discussed in Section 6.

In Figure 5, a database management system is being created with two calendric systems, the `russian` calendric system and the `american` calendric system. One calendric system must be declared as the default. The UCS data structures referencing these calendric systems are built by the toolkit, compiled, and stored as object code in `ucs.o`. Similarly, the internal calendric system data structures referencing calendars and procedures are built, compiled and stored as object code in `russian.o` and `american.o`. For each calendar in the DBMS, the required and optional functions for that calendar are created by the DBI in the files `geologiccal.c`, `carbon-14cal.c`, ..., and `gregoriancal.c` and then compiled, and a default property table is built by the DBI. Lastly, we note that predefined object libraries are provided for the TADT and the remainder of the UCS.

This architecture shares the characteristics of most extensible DBMSs, in that certain aspects are bound at DBMS-generation time, other aspects are bound at schema-definition time, and still other aspects are bound during query evaluation. Specifically, in our design calendars and calendric systems are declared when a DBMS is generated; the calendric system is bound at schema definition time (or more precisely, when an SQL module is compiled), and properties, such as output format, are bound at query evaluation time.

4.8 Architectural Implications for Extensibility

A well-known concern with extensible DBMSs is that extensions are error-prone—they interact with the core DBMS, but are developed separately, and usually by less experienced personnel. These errors can affect not only the correctness of the extension, but the correctness, performance, and security of the DBMS itself. In our context, the DBI must be concerned with protecting the DBMS from errors introduced by calendars. However, this protection must be balanced with the degree of flexibility needed to meet a given site's requirements.

There are several ways that a site's requirements can be met while still ensuring that calendars do not adversely affect the DBMS. Many site-specific adaptations can be accommodated through the manipulation of properties, with little exposure to security violations, and little impact on correctness or performance. For greater extensibility, the DBI can insist that only vendor supplied calendars be installed, assuming the site's requirements can be met through vendor developed packages. If the vendor code is well-tested, then it is safe assumption that the calendar will not adversely affect the DBMS. If the site's requirements cannot be met by available packages, the architecture has been designed to simplify the construction of calendars as much as possible. Much of the processing has been moved outside of the calendars and into the UCS and the TADT, and the generation toolkit is designed to minimize the actual programming effort required for a calendar. If the DBI can be trusted to write correct code, the architecture accommodates a high degree of flexibility.

A related issue is how to isolate calendars from the DBMS internally within the architecture. As previously mentioned, the interaction between the calendar and other modules has been minimized as much as possible. Calendars invoke only TADT operations, and this is necessary since the time-stamp representation is encapsulated within the TADT. Also, whenever possible, parameters passed from the UCS to a calendar and from a calendar to the TADT, are passed by value rather than by reference to minimize the chance of memory contamination. Lastly, the architecture accommodates a variety of implementation strategies for calendar address spaces. Calendars can share the DBMS's address space, or exist in a separate address space, either their own or the user's address space. These options represent a tradeoff between performance and risk of contamination. Highest performance is possible in the address space of the DBMS, at the risk of having the least isolation. The performance difference between the remaining two options is negligible. However, we note that the code space required to duplicate calendars in individual user processes could be substantial, and failure of a calendar could cause failure of the user process. Lastly, if calendars occupy their own, separate address spaces, their services can conceivably be made available system-wide, as opposed to the DBMS exclusively, thus encapsulating calendar services for the computing system as a whole.

The architecture accommodates a spectrum of strategies for calendar extensibility. The DBI must balance flexibility and performance against the risk of errors when selecting or developing new calendars for an installation. The architecture aids in this process by easing the development of new calendars, and simplifying their interaction with other components. If desired, a great degree of flexibility is available. Otherwise, the design attempts to simplify the extension process as much as possible.

5 Related Work

Several researchers have investigated time in databases from a conceptual viewpoint. Anderson developed a formal framework to support conceptual time spaces using inheritance hierarchies [Anderson 1982, Anderson 1983]. Her model also 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.

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.

Allen motivated the *interval* as a fundamental temporal entity [Allen 1983]. He formalized the set of possible relationships which 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 interval relationships as the basis for defining new temporal comparison operators in SQL.

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.

Many other researchers have developed sophisticated time-oriented data models and extended SQL to support these data models [Ariav 1986, Ben-Zvi 1982, Navathe & Ahmed 1989, Sarda 1990]. 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].

The extensibility of calendars and calendric systems is a limited form of database extensibility [Carey & Haas 1990]. Our proposal supports *query language extensibility* in the form of calendar functions, and *presentation extensibility* in the form of time display customization. We note that the temporal types utilized in the query language are *not* extensible, though the domain of spans can be enlarged with variable spans defined through a calendar.

Several extensible prototypes offer the capability to construct abstract data types (ADTs) [Stonebraker et al. 1990], and it is reasonable to ask whether time can be adequately supported as an ADT. We feel that it cannot. Time is a fundamental data type—many DBMSs provide it and most applications use it. Indeed, the SQL2 proposal [Melton 1990], in addition to the SQL variant supplied with IBM's DB2 [Date & White 1990], both provide special support for time. As such it is appropriate for temporal data to be supported by the DBMS directly rather than supported by a local extension. Furthermore, calendar selection would be awkward to specify in a query if added as a database extension rather than providing base query language constructs to the user. In particular, compile-time checking of calendar functions, which is possible using static scoping, would be precluded if time were supported strictly as an ADT.

6 Conclusions and Future Work

We have proposed an extension to SQL and a system architecture addressing the problem of time value representation in a conventional relational database management system. The contributions of this paper can be summarized as follows.

- We argued that many different calendars are in use, due to the cultural, linguistic, legal, and business concerns of users, and we showed how supporting multiple calendars and parametrization of calendars by properties can address these needs.
- We introduced the novel concepts of spans, calendric systems, calendars, and calendar properties.
- We extended SQL2 to support multiple calendars and calendric systems, and, in the process, reduced the complexity of the language while increasing its expressive power.
- We proposed an architecture that permits the database implementor (DBI) at a local site to define new calendars and calendric systems, and allows the database administrator and users to parametrize those calendars, providing limited extensibility of this simple but important component of the DBMS.
- Our architecture moves most of the processing of time into two modules, the temporal abstract data type module, and the uniform calendric support module, and out of the DBI-supplied calendar modules, thereby separating the universal aspects of time from the user dependent aspects.

The key aspect of the proposal is that the DBMS support needed for the user dependent aspects of time is partitioned from the support for the universal aspects of time, and this partitioning is present at both the query language and system architecture levels.

We are currently implementing a prototype database management system using the architecture described in this paper. This includes prototyping several calendars and calendric systems and the generation tools described in Section 4.7. A detailed design describing the interfaces between all modules in the architecture can be found elsewhere [Soo et al. 1992].

We plan several enhancements and additions to the current proposal. We hope to further automate the production of calendars by defining small declarative languages for most aspects of calendars, allowing the calendar specific routines to be generated without the DBI being required to write the C routines in most cases. Our current design for the generation tools assumes that input is fixed-format, that is, that temporal constants have a structure that is known a priori by the system. While existing DBMSs also make this assumption, we feel it is overly restrictive. Finally, we would like to investigate strategies for integrating historical indeterminacy [Dyreson & Snodgrass 1991] into the architecture.

7 Acknowledgements

Curtis Dyreson, Suchen Hsu, and Christian S. Jensen made comments on early drafts of this paper that greatly improved the presentation. Curtis Dyreson and Suchen Hsu also made significant contributions to the architectural design.

Support for this research was provided in part by the National Science Foundation through grant IRI-8902707 and by the IBM Corporation through contract #1124.

8 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.

[Batory et al. 1988] Batory, D., J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell and T. Wise. "GENESIS: An Extensible Database Management System." *IEEE Transactions on Software Engineering*, 14, No. 11, Nov. 1988, pp. 1711–1730.

[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.

[Carey & Haas 1990] Carey, M. and L. Haas. "Extensible Database Management Systems." *ACM SIGMOD Record*, 19, No. 4, Dec. 1990, pp. 54–60.

[Carey et al. 1986] Carey, M.J., D.J. DeWitt, J.E. Richardson and E.J. Shekita. "Object and File Management in the EXODUS Extensible Database System," in *1986 VLDB Conference*. VLDB. Kyoto, Japan: Aug. 1986, pp. 1–27.

[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 & 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, 31 pages.

[Dyreson & Snodgrass 1992] Dyreson, C. E. and R. T. Snodgrass. "Time-stamp Semantics and Representation." TempIS Technical Report 33. Computer Science Department, University of Arizona. Feb. 1992, 27 pages.

[Fraser 1987] Fraser, J. "Time the Familiar Stranger." Redmond, WA: Tempus Books, 1987.

[Haas et al. 1990] Haas, L., Chang, W., Lohman, G., McPherson, M., Wilms, P., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M., and E. Shekita. "Starburst Mid-Flight: As The Dust Clears." *IEEE Transactions on Knowledge and Data Engineering*, 2, No. 1, Mar. 1990, pp. 143–160.

[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 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.

[Richardson & Carey 1987] Richardson, J.E. and M.J. Carey. "Programming Constructs for Database System Implementation in EXODUS," in *Proceedings of the ACM SIGMOD Annual Conference*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: ACM Press, May 1987, pp. 208–219.

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

[Silberschatz et al. 1990] Silberschatz, A., M. Stonebraker and J. Ullman. "Database Systems: Achievements and Opportunities." *ACM SIGMOD Record*, 19, No. 4, Dec. 1990, pp. 6–22.

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

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

[Soo & Snodgrass 1992] Soo, M. and R. Snodgrass. "Mixed Calendar Query Language Support for Temporal Constants." TempIS Technical Report 29. Computer Science Department, University of Arizona. October 30, 1991 1992, 55 pages.

[Soo et al. 1992] Soo, M., 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, 30 pages.

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

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

[Stonebraker et al. 1990] Stonebraker, M., L. Rowe and M. Hirohama. "The Implementation of POSTGRES." *IEEE Transactions on Knowledge and Data Engineering*, 2, No. 1, Mar. 1990, pp. 125–142.

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