D E F I N I N G D A T A T Y P E S

I N A D A T A B A S E L A N G U A G E

Alternative title:

A P R O P O S A L F O R A D D I N G

D A T E A N D T I M E S U P P O R T T O S Q L

by

C. J. Date

Codd and Date International
6489 Camden Avenue, #109
San Jose, Calif. 95120

February 17th, 1988

Abstract

The question of defining data types in a database language is examined. In order to illustrate the general ideas and make them more concrete, the specific case of adding support for dates and times to the database language SQL is considered in detail.

This material is planned for inclusion in a forthcoming book by C. J. Date

INTRODUCTION

There is a continuing requirement as database languages evolve to extend those languages to incorporate new data types -- certainly system-defined (builtin) data types, and eventually user-defined data types also. An examination of existing database products will readily show that this activity has sometimes been carried out in the past in a rather ad hoc manner. The purpose of this paper is to describe a systematic approach to the problem, in the hope that future data type extensions will be defined in a more systematic fashion. Please note, however, that I do not claim that the ideas presented are particularly original -- they are basically part of the conventional computer language designer's stock-in-trade -- but they do not appear to be common knowledge in the database world, and I am aware of only a few papers that consider the problem from a database perspective [1,2,4].

In order to illustrate the ideas and make them more concrete, I will consider a specific (and nontrivial) example in some detail -- namely, the problem of adding support for dates and times to the database language SQL. The paper may thus also be regarded as a concrete change proposal for SQL. (Though I feel bound to add that this choice of example is less than ideal, involving as it does an attempt to extend an already unsystematic language in a systematic manner. It is of course a choice that more or less makes itself, given the widespread acceptance and importance of the SQL language.)

Of course, several commercial products that include SQL date and time support do already exist, and the ANSI and ISO SQL standards committees are also currently at work on the problem. Most of the existing commercial implementations are regrettably ad hoc, however. A detailed tutorial treatment of one particular implementation, that of IBM in its products DB2 and SQL/DS, appears in references [7] (for DB2) and [9] (for SQL/DS); a critical analysis of that implementation can be found in reference [10]. It is perhaps worth mentioning that the process of preparing this paper involved development and investigation of a variety of different attacks on the date/time problem; the approach described herein seems (to my mind) to be more satisfactory than most.

As will be seen, the process of defining a new data type involves a sequence of several interdependent steps. I list those steps here for purposes of reference.

* Deciding required semantics
  - what is required in intuitive terms ?
  - basic data elements
  - conversion operations
  - comparison operations
  - computational operations
  - builtin functions
  - assignment operations

* Defining syntactic details
  - constants
  - variables
  - operators and expressions
  - input/output formats

* Deciding implementation details
  - internal representation
  - operator implementation
  - host language interfacing

PRELIMINARIES

As the foregoing list of steps indicates, there are at least three sets of issues involved in language design: semantic, syntactic, and implementation issues. It is important to keep these three aspects separate as far as possible. Unfortunately, this goal of separation has not not always been achieved in the past. One problem is that semantic ideas must necessarily be presented in some syntactic form, with the result that semantic and syntactic issues, at least, do tend to get confused. Another problem is that designers in the past have all too often been overly concerned with implementation issues, with the result that the design has proceeded from the inside out and internal details have been allowed to dictate external function. In this paper I will attempt to keep separate issues separate as much as I can. My general approach will be to try to get the semantic aspects right first, then worry about the syntactic and implementation aspects later; these latter aspects are important, but they are secondary.

(Of course, I am not so naive as to think that the various issues can always be clearly differentiated. There will certainly be situations in which syntactic or implementation considerations will have some impact on the semantic design. But I still believe that concentrating on semantics first is the right approach.)

Turning now to the example at hand (adding date and time support to SQL): I will assume for the purposes of this paper that we do already have a reasonable idea of the objective we are trying to achieve (i.e., we already have a good intuitive understanding of the desired functionality) -- basically, the ability to represent dates and times in the database, the ability to perform sensible arithmetic operations on such dates and times (e.g., to subtract one date from another to yield an interval), the ability to extract the year, month, etc. components of a date separately when necessary, the ability to compare two dates to find the more recent, and so on. The first thing we have to do, then, is decide precisely what the basic data elements are that we need to deal with.

In the real world we are accustomed to regarding past, present, and future as forming an infinite linear continuum, the timeline. Individual dates and times correspond to individual points on the line. For brevity, let us agree to refer to each point on the line as a date, even though it actually includes a time component. In order to provide a way of uniquely identifying individual dates, we choose some specific point on the line as the origin; the origin is actually arbitrary, but convention dictates that we choose some commonly accepted point, such as midnight on the day immediately preceding January 1st, Year One A.D., Greenwich Mean Time (GMT). We can then refer to individual dates in terms of their relative position with respect to that origin.

As already indicated, the real-world timeline is infinitely long; furthermore, an infinite number of distinct dates (time points) exist within any given segment of the line. The computerized version has to be constrained by the limitations of finite digital devices, however. We therefore have to adopt some compromises. Specifically, we have to decide the granularity we will use (i.e., the time distance between one point and "the next" on the line); we also have to decide the range of values we can represent (i.e., the extreme points on the line). Once again, these decisions are somewhat arbitrary, of course; our later decisions and discussions will not be affected in any fundamental way by our choices at this stage. For purposes of definiteness, therefore, let us agree on the following:

granularity	=	1 second
origin	=	midnight on the day immediately preceding January 1st, 1 A.D., GMT
range of values	=	from midnight, December 31st, 10,000 B.C., to one second before midnight, December 31st, 9,999 A.D., both GMT

Notes:

1. It is a peculiarity of our calendar that there is no "Year Zero"; December 31st, Year 1 B.C., immediately precedes January 1st, Year 1 A.D.

2. For the purposes of this paper, we ignore the complications caused prior to the 17th century A.D. by the irregularities of the calendar in use at that time (the Julian calendar).

3. The extreme values in the range are basically arbitrary; indeed, there is no particular reason why they even have to lie on exact year boundaries. As already stated, we take the extreme points we do purely for reasons of definiteness.

4. Facilities must be provided by which users who are not interested in precisions down to the second -- for example, users who wish to deal only with precisions to the nearest day -- can easily ignore the less significant portions of any given date. Facilities must also be provided for the extraction of individual components (in particular, the time components, either individually or en bloc) of a given date.

Next, in order to be able to perform arithmetic on dates, we need the concept of an interval, so that we can (for example) add an interval to a date to obtain another date. Now, at first sight it might appear that intervals and dates are fundamentally different kinds of object. However, a date is really nothing more than a special case of an interval, representing as it does a specific interval of time relative to the origin point. What is more, attempting to treat dates and intervals as different kinds of object leads to certain logical inconsistencies. For example, if D is a date and I is an interval, then the expressions

```
D + I
I / 2
```

both seem to make sense (assuming that we agree that normal arithmetic operators can be applied to dates and intervals); the result is a date in the first case and an interval in the second case. Likewise, if D1 and D2 are two dates, the expression

```
D1 - D2
```

also seems to make sense (the result is an interval). On the other hand, the expression

```
D1 + D2
```

does not seem to make much sense and would therefore probably have to be outlawed. It therefore follows that the expression

```
( D1 + D2 ) / 2
```

would have to be outlawed also, even though the overall expression does make sense ("find the date midway between D1 and D2"). So one consequence of treating dates and intervals as distinct objects is that certain apparently meaningful expressions would have to be declared illegal.

A second consequence, and one that makes matters even worse, is that the expression

```
D1 + ( D2 - D1 ) / 2
```

would not be outlawed -- it is of the form "date + interval" -- and yet it is logically equivalent to the previous one.

Here is another example of an inconsistency: If we decide that date subtraction does make sense but date addition does not, then, if D1, D2, D3, and D4 are all dates, the comparison

```
D1 - D2  >  D3 - D4
```

is obviously legal, but the logically equivalent comparison

```
D1 + D4  >  D3 + D2
```

is not.

As a result of considerations such as the foregoing, we will not distinguish between dates and intervals internally. Instead, we will simply regard a date -- sometimes called a calendar date -- as an interval relative to the origin. Of course, we will probably still wish to make certain distinctions externally between calendar dates and intervals, for obvious pragmatic reasons. From this point on, therefore, we will regard the interval as the primary (indeed, the only) fundamental data object, but we will use the term "date" (or "calendar date") to refer to an interval that we wish to interpret as a date in the conventional sense. Please note, however, that the distinction is purely one of convention; all dates can be regarded as intervals and all intervals can be regarded as dates. The algorithm for converting from intervals to calendar dates and vice versa is of course well-defined (assuming that all calculations are based on the Gregorian calendar). For example, the interval "10 hours 30 minutes" can be converted to the calendar date "January 1st, Year One A.D., 10:30 a.m.," and vice versa.

There is, of course, one major complicating factor in regard to the concept of an interval (or date), namely as follows: Different months have different numbers of days. (Indeed, different years have different numbers of days also, owing to the existence of leap years.) This fact can be the source of much complexity and confusion. For example, it is the direct cause of the following anomaly in the IBM support for dates and times [10]:

```
DATE ('1987-3-31')  + 1 MONTH  - 1 MONTH   yields  '1987-3-30'
```

(not '1987-3-31'); that is, adding one month to a date and subtracting it again does not necessarily take us back to where we started. And yet:

```
DATE ('1987-3-31')  + 2 MONTHS - 2 MONTHS  yields  '1987-3-31'
```

Many similar anomalies can be identified [10].

For reasons such as the foregoing, we do not include a years or months component in an interval. Instead, we define an interval as consisting of just four components: days, hours, minutes, and seconds (in major-to-minor order, of course; remember that we have adopted a one-second granularity). Operations such as adding two intervals or comparing two intervals can then be defined to obey normal mathematical laws, such as the law of associativity (which guarantees among other things that x + y - y is always equal to x). Special operators can be defined to support "calendar-style" arithmetic (e.g., ADD_MONTHS; see later in the paper for details). In informal contexts we may choose to regard an interval as so many years, so many months, so many days, etc.; but we have to understand that such a perception is only approximate and may lead to erroneous conclusions if taken too literally. Formally, to repeat, an interval consists of an integral number of days, hours, minutes, and seconds.

Any interval, then, can be logically regarded as a signed concatenation of four numeric components d, h, m, and s, all with the obvious meanings. For intervals in the conventional sense, the sign has the obvious interpretation; for calendar dates, positive = A.D., negative = B.C. Without loss of generality, we assume that all four components are always present (some of them might be zero, of course); we also assume that each component is always within its legal range -- for example, the hours component is never greater than 23. Legal ranges are defined as follows:

```
d  :  0 - x  (where x is actually arbitrary, but for the purposes
              of this paper is the number of days in 9,999
              Gregorian years, minus one)
h  :  0 - 23
m  :  0 - 59
s  :  0 - 59
```

Note: Remember that we are discussing a formal representation here, in which, e.g., the first day of the year is considered to be day zero, not day one. This choice of formal representation does not necessarily preclude users from representing intervals (and calendar dates) in a more conventional manner externally (see later). Nor does it mean that users always have to worry about precisions down to the second in every case.

We have now pinned down the basic data objects we will be dealing with. Next we have to decide what kinds of operations we want to be able to perform on those objects. We can divide those operations up into various classes:

* conversions

* comparisons

* computational operations

* builtin functions

* assignment

We consider each class in turn.

CONVERSIONS

The first question to address is: What conversions are needed between the new data type (intervals) and existing types in the language? At least the conversions listed below seem to be desirable. [Note: We assume that each is performed by means of an explicit builtin conversion function; for reference, we give the names of those functions in square brackets.]

* The ability to convert a specified interval to some standard external representation of an interval (e.g., a character string of the form d:h:m:s, with a leading minus sign if the interval is negative), and vice versa [INTERVAL_TO_CHAR, CHAR_TO_INTERVAL]

* The ability to convert a specified interval, considered as a calendar date, to some standard external representation of such a date (e.g., a character string of the form y:n:d:h:m:s, with a BC indication if necessary), and vice versa [DATE_TO_CHAR, CHAR_TO_DATE]

* The ability to convert a specified interval to a number representing the number of specified time units (days, minutes, etc.) in that interval, and vice versa [INTERVAL_TO_NUM, NUM_TO_INTERVAL]

* The ability to convert a specified interval, considered as a calendar date, to an integer (1 to 7, where 1 = Sunday, etc.) representing the corresponding day of the week [DAY_OF_WEEK]

* The ability to convert a specified interval, considered as a calendar date, to an integer (1 to 366) representing the corresponding day of the year [DAY_OF_YEAR]

* The ability to convert a specified interval, considered as a calendar date, to an integer (1 to 52, or maybe 53) representing the corresponding week of the year [WEEK_OF_YEAR]

The following can also be regarded as conversions of a kind:

* The ability to treat a specified interval as a calendar date, to extract any individual component (years, months, etc.) of that date, and convert the result to an integer [YEARS_PART, MONTHS_PART, DAYS_PART, HOURS_PART, MINUTES_PART, SECONDS_PART]

* The ability to extract the time components (hours, minutes,

and seconds) of a specified interval en bloc and treat the result as another interval [TIME]

* The ability to extract the days component of a specified interval (ignoring the time components) and treat the result as another interval [TRUNC_TO_DAYS]

Notes:

1. The choice of keywords such as "HOURS_PART" (etc.), rather than the more obvious "HOURS" (etc.), is made deliberately. The keyword "HOURS" might tend to suggest that the given interval is to be converted to the total number of hours in that interval. Similarly for "MINUTES," etc., of course.

2. It would obviously be possible to define some implicit conversions (also known as coercions) to be applied on, e.g., assignment from an interval to a character string or comparison between an interval and a character string (etc., etc.). Such implicit conversions can be regarded as shorthands for the appropriate explicit conversions; as such, they are merely matters of concrete syntax, not worthy of consideration at this stage. In fact, I think it just confuses the issue to get into such questions before the semantic requirements have been properly pinned down.

3. The problem of conversion also arises in the context of source/sink I/O (how should dates and intervals be presented externally to the user?) and in the context of interfacing the database language to a host language (how do dates and intervals interact with the data types of that host?). Once again, however, these are not truly semantic issues; the first is syntax again and the second is more of an implementation issue. I will address them (briefly) toward the end of the paper.

COMPARISONS

There is really not too much to be said under this heading, thanks to our choice of intervals (days, hours, minutes, and seconds) as the basic data object. Essentially, any two intervals can be compared by means of the usual scalar comparison operators =, <>, <, <=, >, and >=; such comparisons are algebraic, with the days component being more significant than the hours component, the hours component more significant than the minutes component, etc. Fancy comparison operators such as IN, BETWEEN, etc. can be defined (if desired) in terms of the basic operators. Likewise, the aggregate operators MAX and MIN applied to intervals can be defined in terms of repeated scalar comparisons; the result in each case is another interval.

COMPUTATIONAL OPERATIONS

Let I, I1, and I2 be intervals and let N be a number. Because intervals can be regarded as (mixed-radix) integers, the following operations are clearly well-defined and return another interval as a result in each case:

```
    + I
    - I

I1 + I2
I1 - I2

I  *  N
N  *  I
I  /  N
```

Multiplication and division of an interval by a number are defined in terms of repeated addition and subtraction, respectively; the result is an interval, with rounding (if any) to the nearest second. The aggregate operators COUNT, SUM, and AVG applied to intervals are defined in terms of repeated scalar arithmetic operators; COUNT yields a numeric value (actually an integer), SUM and AVG each yield another interval.

It is also possible to divide one interval by another:

```
I1 / I2
```

Such a division is defined by converting each operand to seconds and performing a numeric division. The result is a number, not another interval.

Finally, in order to support calendar arithmetic, we introduce two special computational functions:

```
ADD_MONTHS ( I1, N )   adds N months to I1

ADD_YEARS  ( I1, N )   adds N years to I1
```

ADD_MONTHS operates as follows. Let interval I1, considered as a calendar date, have components

```
Y1   N1   D1   H1   M1   S1
```

(where Y1 is the years component, N1 the months component, etc.; note that N1 is in the range 1 to 12 and D1 is in the range 1 to 31). Then the value of ADD_MONTHS(I1,N) is an interval I2, with components (when considered as a calendar date)

```
Y2   N2   D2   H2   M2   S2
```

defined by the following pseudocode:

```
let N = 12y + n (0 <= n <= 11) ;
S2 := S1 ;
M2 := M1 ;
H2 := H1 ;
D2 := D1 ;
N2 := N1 + n ;
Y2 := Y1 + y ;
if N2 > 12
   then do ;
           N2 := N2 - 12 ;
           Y2 := Y2 + 1 ;
        end ;
if N2 in (April,June,September,November) and D2 = 31
   then D2 := 30 ;
if N2 = February and Y2 is a leap year and D2 in (31,30)
   then D2 := 29 ;
if N2 = February and Y2 is not a leap year and D2 in (31,30,29)
   then D2 := 28 ;
```

We are assuming here that I1 and N are both positive. The extension to deal with negative values of I1 and/or N is tedious but straightforward.

ADD_YEARS is defined analogously (the only "interesting" case is that in which I1 represents February 29th for some leap year and adding N years takes us to a nonleap year).

BUILTIN FUNCTIONS

This heading is a catchall for any additional functions that may be needed over and above the conversion and computation functions already covered (briefly) above. In the case of dates and intervals, at least the following additional functions seem to be required:

* A zero-argument function that returns the absolute date "now," i.e., as of the point of invocation [NOW]

* Functions to convert a GMT date to some local timezone and vice versa. Details of these functions are not given in this paper; for simplicity we assume throughout that all dates are GMT.

ASSIGNMENT

The only new form of assignment we need in the database language per se is the assignment of an interval value to an interval variable (i.e., column). In SQL terms, assignment is performed only in the context of INSERT and UPDATE operations.

Notes:

1. The "interval assignment" operation should include the ability to assign to individual components (such as the hours component) of an interval variable. Such functionality can be provided by using the component extraction functions discussed earlier (YEARS_PART, MONTHS_PART, etc.) as "pseudovariables" (to use a PL/I term). For example (using the syntax to be introduced later):

```
UPDATE LAUNCH_SCHEDULE
SET    HOURS_PART ( LIFTOFF ) = 15
WHERE  ROCKET = 'Saucy Sue' ;
```

The intent of this example is to set the hours component of the LIFTOFF value within the specified record to 3 p.m.

2. Assignment of an interval value to an interval variable may include truncation or padding to match the precision of the target. See the discussion of "Variables," later.

CONSTANTS

Having decided the semantics (functionality) we want, we can now turn our attention to syntactic issues. Note: My primary objective in this paper with respect to syntax is simply to arrive at a concrete syntax that is capable of expressing all required functionality in an unambiguous fashion. The specific syntax given could probably do with some refinements in order to improve its usability, but first things first.

First we need a syntax for constants:

```
interval-constant
   ::=    [ + | - ] DAYS     'd[:h[:m[:s]]]'
        | [ + | - ] HOURS    'h[:m[:s]]'
        | [ + | - ] TIME     'h[:m[:s]]'
        | [ + | - ] MINUTES  'm[:s]'
        | [ + | - ] SECONDS  's'
        |           DATE     'y[:n[:d[:h[:m[:s]]]]] [ AD | BC ]'
```

Examples:

```
    DAYS    '100'
  - HOURS   '3:10:30'
    TIME    '3:10:30'
  + MINUTES '5'
    SECONDS '1'
```

```
DATE '1941:1:18:9:30:45'
DATE '1941:1:18:9:30'
DATE '1941:1:18:9'
DATE '1941:1:18'
DATE '1941:1'
DATE '1941'
DATE '1941 BC'
```

Syntax rules:

1. The DAYS, HOURS, TIME, MINUTES, SECONDS, and DATE keywords specify the units for the leftmost integer value (y, d, h, m, or s) appearing within the single quotes; DAYS, HOURS, MINUTES, and SECONDS have the obvious meanings, TIME is a synonym for HOURS, and DATE means years. Note that there are no YEARS or MONTHS forms.

2. If neither AD nor BC is specified for a DATE constant, AD is assumed.

3. For DATE constants, the syntactic categories y, n, d, h, m, and s are defined as follows:

    y = unsigned decimal integer of 1-4 digits (1-9999)

    n = unsigned decimal integer of 1-2 digits (1-12)

    d = unsigned decimal integer of 1-2 digits (1-31)

    h = unsigned decimal integer of 1-2 digits (0-23)

    m = unsigned decimal integer of 1-2 digits (0-59)

    s = unsigned decimal integer of 1-2 digits (0-59)

For other constants, y and n do not appear, and d, h, m, and s are basically as for DATE, except that the legal range for d is 0 - x (where x is the number of days in 9,999 years, minus one).

4. In all cases, omitting a value for h or m or s is equivalent to specifying a value of 0. For a DATE constant, omitting a value for n or d is equivalent to specifying a value of 1. For other constants, omitting a value for d is equivalent to specifying a value of 0.

5. Illegal DATE constants such as DATE '1987:4:31' and DATE '1987:2:29' (etc., etc.) are prohibited, and appropriate syntax rules are required to define the details. I omit those details here.

Discussion:

1. There are no YEARS or MONTHS constants because it is impossible to construct such constants in such a way that they can be simultaneously (a) precisely defined and (b) intuitively useful. For example, what would MONTHS'5' mean? 150 days (five average months of 30 days)? 151 days (the actual number of days in the first five months of the year)? Or 152 days (if it is a leap year)?

2. DATE constants and DAYS, HOURS, etc. constants both correspond to the same data type (the interval), and they can be used interchangeably. The reason for including both is usability.

3. The interval constant DAYS '0:0:0:0' represents the origin point. The DATE equivalent is DATE '1:12:31:23:59:59 BC'.

4. The purpose of the DATE, DAYS, etc. keywords is to make interval constants syntactically distinguishable from all other tokens in the language, including in particular string and numeric constants. As a general principle, I favor a context-free approach to syntax. Not all current systems abide by this principle. The opposite of "context-free" is "context-sensitive"; and the trouble with context-sensitivity is that it can lead to strange and complicated syntax rules. For example, in ORACLE [5], date constants are written in the form

```
'dd-mon-yy'
```

and hence are indistinguishable from string constants. What then are we to make of the following expression?

```
'29-APR-87' - '18-JAN-41'
```

Subtraction of one date from another is legal, but subtraction of one string from another is not. I have no idea what ORACLE will do with this expression, though I do know the following is legal:

```
'29-APR-87' + 5
```

(it evaluates to '4-MAY-87').

5. A small point: Intervening blanks could be permitted within an interval constant if desired without causing any ambiguity, but by analogy with string constants it is probably better to prohibit them. On the other hand, it is probably desirable to permit blanks between the keyword (DATE, DAYS, etc.) and the actual value, for reasons of readability.

6. Another small point: The proposals of this paper assume a 24-hour clock; in particular, there are no explicit "a.m." or "p.m." constants. Of course, a.m. and p.m. support could easily be added if desired.

VARIABLES

The next thing to do is to define a syntax for the definition of interval variables (i.e., columns, in SQL terms):

```
interval-column-definition
   ::=  column INTERVAL [ precision ] [ NOT NULL [ WITH DEFAULT ] ]

precision
   ::=  ( start [ : end ] )

start
   ::=  DAYS | HOURS | MINUTES | SECONDS

end
   ::=  DAYS | HOURS | MINUTES | SECONDS
```

Examples:

```
REVIEW_DATE INTERVAL ( DAYS : DAYS )
BIRTH_DATE  INTERVAL ( DAYS )
LIFTOFF     INTERVAL
WAIT_TIME   INTERVAL ( HOURS : SECONDS )
```

Syntax rules:

1. If "precision" is omitted, a precision of (DAYS : SECONDS) is assumed.

2. If "end" is omitted, it is assumed to be equal to "start".

3. "Start" must be greater than or equal to "end," according to the ordering DAYS > HOURS > MINUTES > SECONDS.

Discussion:

1. The precision specification allows the user to indicate that he or she is interested only in certain components of the general interval. One common special case would be "(HOURS:SECONDS)" (meaning time components only -- ignore the days). A special shorthand could be introduced for this case if desired.

2. The WITH DEFAULT specification follows the style of IBM SQL, not ISO/ANSI standard SQL; IBM SQL includes the concept of system-defined default values, which ISO/ANSI SQL does not. On the other hand, the proposed extensions to the ISO/ANSI standard [11] include the concept of user-defined default

values, which IBM SQL does not. Either way, default values (whether system- or user-defined) are used in lieu of nulls if the user does not supply a value for the column on INSERT -- also for values in existing rows for columns added to an existing table via ALTER TABLE (assuming in both cases that nulls are not allowed for the column in question).

[Just as an aside, it is extremely annoying in SQL that several options, such as "defaults not allowed," can be specified only by the absence of some particular syntactic construct. This state of affairs makes it very difficult to talk about such cases. A good rule in designing a language's concrete syntactic form is that every option should be explicitly expressible. But I digress.]

3. The obvious system-defined default value to choose is the origin point (DAYS '0:0:0:0').

4. There would be no real harm in allowing DATE as an alternative spelling for INTERVAL, for usability reasons -- except that the DATE version of the "obvious" system-defined default, namely DATE '1:12:31:23:59:59 BC' might seem a trifle unobvious to the user.

OPERATORS AND EXPRESSIONS

Every time we add a new kind of data object to an existing language, we need to decide

(a) whether existing operators of the language apply to such objects, and

(b) whether any new operators will be needed for such objects.

Then we need to decide how objects and operators can be combined to form expressions. We also need to decide the data type of the result of any such expression. Finally, it is desirable that all such expressions conform (to the maximum extent possible) to the well-known principle of orthogonality. (A language is said to be orthogonal if independent concepts are kept separate and are not mixed together in confusing ways.)

Since the objects we are adding to SQL (namely, intervals) are scalar objects, let us begin with the concept of a scalar expression. A scalar expression is any expression that evaluates to a scalar value. Orthogonality dictates that such expressions should be permitted to appear wherever a scalar constant can appear -- which (in the case under discussion) means all of the following contexts:

* Within a SELECT clause (to represent a value to be "selected" or retrieved);

* Within a WHERE or HAVING clause (to represent a comparand in a comparison expression -- see later);

* Within a VALUES clause (to represent the value to be assigned to a column in an INSERT operation);

* On the right-hand side of an assignment within a SET clause (to represent the value to be assigned to a column in an UPDATE operation);

and also (arguably)

* Within a GROUP BY or ORDER BY clause.

Note: Clearly, any expression that selects a scalar value from the database should also be regarded as a scalar expression. However, SQL -- regrettably -- does not currently abide by this principle. In fact, SQL violates orthogonality in numerous additional ways; for example, even a simple scalar expression such as X + 1 is not currently permitted in the VALUES clause [3,6,8]. Details of further violations are beyond the scope of this paper.

Here then is an orthogonal syntax (with explanations) for scalar expressions that includes intervals. For brevity, I abbreviate "expression" to "exp" in this syntax.

```
1. scalar-exp
   ::=   interval-exp
       | numeric-exp
       | string-exp
       | comparison-exp
```

Aside: Comparison expressions cannot strictly be regarded as scalar expressions in existing SQL because they are truth-valued and SQL does not currently support a truth-valued data type (which means in turn that such expressions cannot appear "wherever a scalar constant can appear," but only in certain specific contexts -- to be precise, in WHERE and HAVING clauses). Support for a truth-valued data type in SQL is desirable.

```
2. interval-exp
   ::=   interval-constant
       | interval-variable
       | interval-exp { + | - } interval-exp
       | interval-exp { * | / } numeric-exp
       | numeric-exp * interval-exp
       | NOW
       | ADD_MONTHS ( interval-exp, numeric-exp )
       | ADD_YEARS ( interval-exp, numeric-exp )
       | { SUM | AVG } ( [ ALL | DISTINCT ] interval-exp )
       | { MAX | MIN } ( interval-exp )
       | TRUNC_TO_DAYS ( interval-exp )
       | TIME ( interval-exp )
       | CHAR_TO_DATE ( string-exp )
       | CHAR_TO_INTERVAL ( units, string-exp )
       | NUM_TO_INTERVAL ( units, numeric-exp )
       | { + | - } interval-exp
       | ( interval-exp )
```

The only cases requiring further explanation here are those involving the builtin functions NOW, TRUNC_TO_DAYS, TIME, CHAR_TO_DATE, CHAR_TO_INTERVAL, and NUM_TO_INTERVAL.

* The zero-argument function NOW returns the date (including the time) "now," i.e., as of the point of invocation. Multiple references to NOW within the same SQL statement are defined to return identical values.

* The TRUNC_TO_DAYS function returns an interval equal to the days portion (only) of its argument.

* The TIME function returns an interval equal to the hours, minutes, and seconds portions (only) of its argument.

* The CHAR_TO_DATE function takes the value of its argument, which must be a character string representation of a DATE constant (excluding the DATE keyword and enclosing quotes), and returns the corresponding interval.

* The CHAR_TO_INTERVAL function takes the value of its argument, which must be a character string representation of an interval constant (excluding the keyword and enclosing quotes, and not of DATE format), and returns the corresponding interval. The "units" argument effectively supplies the missing keyword (DAYS, HOURS, TIME, MINUTES, or SECONDS: see below).

* The NUM_TO_INTERVAL function returns an interval corresponding to N "units," where N is the value of the second argument and "units" is the value of the first argument (DAYS, HOURS, TIME, MINUTES, or SECONDS: again, see below). The result is rounded (if necessary) to the nearest second.

3. units
   ::= string-exp

The string expression must evaluate to one of the strings 'DAYS', 'HOURS', 'MINUTES', 'SECONDS', or 'TIME'. TIME is a synonym for HOURS. In practice the string expression will usually consist of a simple character string constant.

Here are some examples (refer back to the sample column definitions given under "Variables" earlier). Note in the first two examples that (a) some table has to be named in the FROM clause, (b) the statement unfortunately yields a surprising result if that table happens to be empty, and (c) the DISTINCT specification is highly desirable, though not strictly necessary.

```
SELECT DISTINCT NOW
FROM   EMP ;

SELECT DISTINCT TIME ( NOW )
FROM   EMP ;

UPDATE EMP
SET    REVIEW_DATE = ADD_MONTHS ( REVIEW_DATE, 8 )
WHERE  EMP# = '123456' ;

SELECT MIN ( BIRTH_DATE )
FROM   EMP ;

SELECT EMP#, TRUNC_TO_DAYS ( NOW ) - BIRTH_DATE
FROM   EMP
WHERE  BIRTH_DATE =
     ( SELECT MAX ( BIRTH_DATE )
       FROM   EMP ) ;

SELECT *
FROM   EMP
WHERE  REVIEW_DATE < CHAR_TO_DATE ( :DATE_PARAM ) ;

SELECT *
FROM   LAUNCH_SCHEDULE
WHERE  TIME ( LIFTOFF ) > TIME '12:30' ;

SELECT *
FROM   LAUNCH_SCHEDULE
WHERE  TIME ( LIFTOFF ) >
       CHAR_TO_INTERVAL ( 'TIME', :TIME_PARAM ) ;

UPDATE EMP
SET    REVIEW_DATE  =
       NOW + NUM_TO_INTERVAL ( 'DAYS', :NUM_PARAM )
WHERE  EMP# = '123456' ;
```

4. numeric-exp

```
::=   COUNT ( [ ALL | DISTINCT ] interval-exp )
    | DAY_OF_WEEK ( interval-exp )
    | DAY_OF_YEAR ( interval-exp )
    | WEEK_OF_YEAR ( interval-exp )
    | INTERVAL_TO_NUM ( units, interval-exp )
    | YEARS_PART ( interval-exp )
    | MONTHS_PART ( interval-exp )
    | DAYS_PART ( interval-exp )
    | HOURS_PART ( interval-exp )
    | MINUTES_PART ( interval-exp )
    | SECONDS_PART ( interval-exp )
    | interval-exp / interval-exp
    | ... other formats beyond the scope of this paper
    | ( numeric-exp )
```

* The DAY_OF_WEEK function returns an integer of default precision representing the day of the week (1 = Sunday, etc.) corresponding to the interval argument, considered as a calendar date; the DAY_OF_YEAR and WEEK_OF_YEAR functions are defined analogously.

* The INTERVAL_TO_NUM function returns the float value N (with default precision), where N is the value of the second argument considered as a number of "units" and "units" is the value of the first argument.

* The YEARS_PART function returns a signed integer of default precision representing the years portion (-9999 to +9999) of the interval argument, considered as a calendar date; the MONTHS_PART, DAYS_PART, HOURS_PART, MINUTES_PART, and SECONDS_PART functions are defined analogously.

* Dividing one interval by another yields a float value of default precision.

Some examples:

```
SELECT *
FROM   EMP
WHERE  DAY_OF_WEEK ( BIRTH_DATE ) = 7 ;

SELECT INTERVAL_TO_NUM ( 'SECONDS', LIFTOFF - NOW )
FROM   LAUNCH_SCHEDULE
WHERE  ROCKET = 'Saucy Sue' ;

SELECT EMP#, EMPNAME
FROM   EMP
WHERE  MONTHS_PART ( BIRTH_DATE ) = 12 ;

SELECT X.EMP#, Y.EMP#
FROM   EMP X,  EMP Y
WHERE  YEARS_PART ( X.HIRE_DATE ) = YEARS_PART ( Y.HIRE_DATE ) ;
```

```
5. string-exp
   ::=  DATE_TO_CHAR ( interval-exp )
      | INTERVAL_TO_CHAR ( interval-exp )
      | ... other formats beyond the scope of this paper
```

The DATE_TO_CHAR function converts the specified interval, considered as a calendar date, to its standard character string representation (i.e., y:n:d:h:m:s [BC]). Likewise, the INTERVAL_TO_CHAR function converts the specified interval to its standard character string representation (i.e., [-] d:h:m:s). Note: Some minor syntactic details need to be pinned down in both cases here regarding leading zeros, blanks, etc.

```
6. comparison-exp
   ::=  interval-comparison
      | numeric-comparison
      | string-comparison

7. interval-comparison
   ::=  interval-exp  scalar-comparison-op  interval-exp

8. scalar-comparison-op
   ::=  = | <> | < | <= | > | >=

9. numeric-comparison
   ::=  numeric-exp  scalar-comparison-op  numeric-exp

10.string-comparison
   ::=  string-exp  scalar-comparison-op  string-exp
```

INPUT/OUTPUT FORMATS

A comment: The question of input/output formats is primarily a syntactic issue. Some systems seem to get hung up on this secondary question before getting the primary (i.e., semantic) issues properly resolved. This is probably why many existing proposals involve a certain degree of context sensitivity. But I would obviously agree that it is desirable to be able to represent intervals in general, and calendar dates in particular, on external media (e.g., on the terminal screen) in a variety of different formats -- for example:

1941:1:18	5 hrs 32 mins 15 secs
1/18/1941	5:32:15
18/1/1941	- 1 hour
1-18-41	1 yr 2 mos 3 days
18.i.1941	- 1 yr 2 mos 3 days
January 18th, 1941	100 years 1 day
18 Jan 1941 AD	96 hours
Saturday, Jan 18th, 1941	100 days
Jan. 18th 41, 9:30 a.m.	0 hrs 10 mins
1/18/41 9:30 am PST	17.5 hours
le 18 janvier, 1941	1.000 ans

(etc., etc., etc.). It would clearly be possible to define some kind of FORMAT option that can be specified at the session level and/or the column definition level and/or the individual statement level. Again, the details are tedious but should be straightforward.

Note: It might prove desirable to support alternative formats for constants also. My only comment in this regard is that, again, any such alternative formats must not violate the context-free requirements discussed earlier in this paper.

IMPLEMENTATION ISSUES

Implementation details are really beyond the scope of this paper. However, I offer the following brief comments:

1. Internal representation: How are dates and intervals to be represented internally? A string representation may facilitate input/output format conversions; it may also make it easier to extract specific components of the value (e.g., the days or minutes component) and to check for illegal dates. On the other hand, a numeric representation -- e.g., integer number of seconds (relative to the origin, in the case of dates), may simplify calendar arithmetic operations. Either way it may prove desirable to introduce some form of data compression on the disk, for storage space reasons. Of course, whatever internal representation is chosen should not be permitted to affect the language semantics.

2. Operator implementation: It would be nice if the new operators (ADD_MONTHS, etc.) could be defined in terms of constructs already existing in the language -- in effect, as SQL subroutines. Unfortunately this is not possible, in general; SQL is only a "data sublanguage" and is not computationally complete. Note the implication that adding new user-defined data types to SQL is therefore likely to require expertise in some other language in addition to SQL per se and will thus get into the question of interfacing between SQL and that other language.

3. Host interfacing: Most host languages do not provide direct

support for dates and intervals. The question therefore arises of mapping between dates and intervals in SQL and host data types. The functions CHAR_TO_DATE, DATE_TO_CHAR, CHAR_TO_INTERVAL, INTERVAL_TO_CHAR, NUM_TO_INTERVAL, and INTERVAL_TO_NUM can be used for this purpose (assuming that the host language does at least support character strings and numbers).

SUMMARY

In this paper I have considered what is involved in adding a new data type to an existing database language such as SQL. The paper has been primarily concerned with system-defined data types, though most of the ideas are in fact directly relevant to the question of adding support for user-defined data types also. By way of example, I have considered in some depth the case of adding support for dates and times to SQL specifically. To recap, the following steps are involved:

* Deciding required semantics
  - what is required in intuitive terms ?
  - basic data elements
  - conversion operations
  - comparison operations
  - computational operations
  - builtin functions
  - assignment operations

* Defining syntactic details
  - constants
  - variables
  - operators and expressions
  - input/output formats

* Deciding implementation details
  - internal representation
  - operator implementation
  - host language interfacing

The paper is intended to serve a dual purpose: first, to assist in the process of formulating proposals for adding new data types to database languages in a systematic manner; second, to assist in the process of critically examining any such proposals. In addition, of course, it can be regarded as the beginnings of a systematic proposal for adding date and time support to the database language SQL.

ACKNOWLEDGMENTS

I am grateful for helpful comments on earlier drafts of this paper to my friends and colleagues Ted Codd, Nat Goodman, Andrew Warden, Sharon Weinberg, and Colin White.

REFERENCES

1. Michael Stonebraker, Brad Rubinstein, and Antonin Guttman: "Application of Abstract Data Types and Abstract Indices to CAD Data Bases." Proc. ACM SIGMOD Database Week, Engineering Applications Stream, San Jose, CA (May 1983).

2. J. Ong, D. Fogg, and M. Stonebraker: "Implementation of Data Abstraction in the Relational Database System INGRES." ACM SIGMOD Record, Vol. 14, No. 1 (March 1984).

3. C. J. Date: "A Critique of the SQL Database Language." ACM SIGMOD Record, Vol. 14, No. 3 (November 1984). Republished in C. J. Date: Relational Database: Selected Writings, Addison-Wesley (1986).

4. Michael Stonebraker: "Inclusion of New Types in Relational Data Base Systems." Memorandum No. UCB/ERL M85/67, UC Berkeley, CA 94720 (July 1985).

5. Oracle Corp.: ORACLE Terminal Users' Guide (1987).

6. C. J. Date: "What's Wrong With SQL?" Available from Codd and Date International, 6489 Camden Avenue, Suite 109, San Jose, California 95120 (March 1987). Republished in abridged form under the title "Where SQL Falls Short" in Datamation (May 1st, 1987).

7. C. J. Date and Colin J. White: A Guide to DB2 (Addison-Wesley, 1988).

8. Andrew Warden: "The Naming of Columns." The Relational Journal, Issue No. 3 (to appear 1988).

9. C. J. Date and Colin J. White: A Guide to SQL/DS (Addison-Wesley, to appear 1988).

10. C. J. Date: "Dates and Times in IBM SQL: Some Technical Criticisms." InfoDB, Vol. 3, No. 1, Spring 1988 (to appear).

11. ISO/IEC JTC1/SC21/WG3 / ANSI X3H2: ISO-ANSI (working draft) SQL2. Document ISO/IEC JTC1/SC21/WG3 N449 / ANSI X3H2-88-1 (December 1988 [sic]).