Efficiently Supporting Temporal Granularities

Curtis E. Dyreson, William S. Evans, *Member*, *IEEE*, Hong Lin, and Richard T. Snodgrass, *Senior Member*, *IEEE*

Abstract—*Granularity* is an integral feature of temporal data. For instance, a person's age is commonly given to the granularity of *years* and the time of their next airline flight to the granularity of *minutes*. A granularity creates a discrete image, in terms of *granules*, of a (possibly continuous) time-line. We present a formal model for granularity in temporal operations that is integrated with temporal indeterminacy, or "don't know when" information. We also minimally extend the syntax and semantics of SQL-92 to support mixed granularities. This support rests on two operations, *scale* and *cast*, that move times between granularities, e.g., from days to months. We demonstrate that our solution is practical by showing how granularities can be specified in a modular fashion, and by outlining a time- and space-efficient implementation. The implementation uses several optimization strategies to mitigate the expense of accommodating multiple granularities.

Index Terms—Calendar, granularity, indeterminacy, SQL-92, temporal database, TSQL2.

1 Introduction

There is one feature common to all temporal data: *temporal granularity*. Temporal granularity is the unit of measure for a temporal datum.[1] For instance, birth dates are typically measured in or known to the granularity of days and train schedules to that of minutes.

Granularities incorporate the cultural, legal, and even business orientation of the user to define the time values that are of interest. Many different granularities exist and no granularity is inherently "better" than another; the value of a particular granularity is wholly determined by the population that uses it. For example, an employee time card can be regarded as a granularity which measures time in eight hour increments and is only defined for five days of each week. It is essential that users be able to define their own granularities; any fixed system of granularities, such as those supported by SQL from the Gregorian calendar, will not meet the needs of all users.

The mixing of temporal data at different, user-defined granularities in a single database will become common when databases can fully support this mixing. This paper offers a practical design for that support. We see the following as the seven main contributions of this paper.

1. In this paper, the term "granularity" will be used in place of the longer phrase "temporal granularity." Our focus is on time and granularity in other domains such as space will not be directly addressed.

- *C.E. Dyreson is visiting the Department of Mathematics and Computer Science, Aalborg University, Fr. Bajers Vej 7E, Dk-9220 Aalborg Øst, Denmark. E-mail: curtis@cs.auc.dk.*
- *W. Evans and R.T. Snodgrass are with the Department of Computer Science, University of Arizona, Tucson, AZ 85721. E-mail: {will, rts}@cs.arizona.edu.*
- *H. Lin is with IBM Global Services, Dept. FA2A, 9000 S. Rita Rd., 031-1/Rm. 517, Tucson, AZ 85744. E-mail: honglin@us.ibm.com.*

Manuscript received 30 June 1998; revised 12 July 1999; accepted 22 Sept. 1999.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 107020.

First, various semantics have been proposed for temporal operations that have operands at different granularities [1], [7], [20], [22], [23], [25], [31], [33]. For instance, in a comparison operation between a time known to the granularity of days and one known to the granularity of hours, the comparison could be performed at days, or it could be done at hours, or an error could be reported. In this paper, we propose two simple operations that can be utilized to support *all* of the previous semantics for temporal operations.

Second, we describe an architecture that permits the rapid development and integration of granularities. In our approach, a user specifies a granularity declaratively, as a mapping from another granularity. One benefit of this approach is that it supports the modular definition of collections of related granularities, which we call *calendars*. Only one granularity in each calendar must be related directly to either a granularity in some other calendar or to the underlying time-line. So, calendars can be developed largely in isolation, yet can be rapidly integrated in a multicalendar database management system (DBMS).

Third, to convert an instant in one granularity to a different granularity, the DBMS must be able to construct a function from one to the other. For example, to convert a time known to the granularity of Gregorian days to the same time expressed in the granularity of Chinese lunar months, the DBMS must be able to convert days to lunar months. It is unlikely that a user will provide a function that converts directly between days and lunar months; instead the function must be dynamically constructed from other user-supplied functions. In this paper, we describe in detail how user-supplied functions provided via calendars are used during query processing to perform a desired conversion.

Fourth, we suggest that an important implementation concept is the identification of "regular" conversions. For example, in the Gregorian calendar, the granularity of

Flight_Departures

Flight#	At_Time
53	'1997-11-20 14:38'
200	'1997-11-27 14:34'
653	'1997-11-27 12:38'
658	'1997-11-30 10:03'

Vacations

Vacation	From_Time	To_Time
'Labor Day'	'1997-09-01'	'1997-09-03'
'Thanksgiving'	'1997-11-24'	'1997-11-28'
'Christmas'	'1997-12-24'	'1997-12-26'

Fig. 1. A flight database.

weeks is regular with respect to days in the sense that each week is composed of seven consecutive days. In contrast, months is "irregular" with respect to days since each month in a year has a different number of days and February sometimes includes an additional leap day (February 29). In general, regular conversions are more efficient that irregular ones. We present a query evaluation strategy that is sensitive to the different conversion costs.

Fifth, we recognize that *indeterminacy*, or "don't know when" information is a companion to granularity. Temporal granularity and indeterminacy are two sides of the same coin, in that a (determinate) time at a given granularity is indeterminate at all finer granularities. For example, a birth date of July 1, 1998 indicates that the person was born *sometime* during the indicated day, but the precise minute is unknown. Indeterminacy also arises naturally in many conversions, e.g., when converting a birthday, given to the granularity of days, to the granularity of minutes.

Sixth, to further underscore the practical focus of this paper, we extend the syntax and semantics of SQL-92 with support for mixed granularities. The bulk of this proposal has been adopted into TSQL2, a temporal extension of SQL-92 [26], constructs from which are now being considered for inclusion into the SQL3 standard [27].

Finally, while support for mixed granularities is a highly desirable database feature, previous research has focused on theoretical concerns and has largely ignored performance. In this paper, we quantify the cost of storing and querying data at different granularities. We show that times at differing granularities can be stored efficiently and that optimization strategies can be used to mitigate the expense of temporal operations at mixed granularities.

In summary, our approach is based on a realistic model of time, is fully integrated with SQL-92 syntax, supports several semantics for temporal operations on operands at differing granularities, and admits an efficient implementation.

We have implemented this approach in C and C++, including support for indeterminacy, granularity conversions, calendar-specification, and multiple conversion semantics [17]. This package could be incorporated into a DBMS to provide a comprehensive solution for efficiently supporting temporal granularities.

The paper is organized as follows: We first give an example that illustrates mixed granularities. We then introduce our model of time. A granularity in this model is a segmentation of the time-line. Next, we present a theory for the semantics of temporal operations on operands at different granularities. We show how to model a wide variety of semantics. We then extend the syntax and semantics of SQL-92 to permit the definition of temporal values at various granularities. We also extend the semantics of temporal operations to handle operands at differing granularities. This query language support rests on two operations that convert temporal values from one granularity to another. Next, we describe the implementation, in particular, how to determine the mapping between granularities, and how to efficiently apply this mapping. Finally, we summarize related work and our work. Throughout the paper, we use examples that involve valid or user-defined time, but the research we present is applicable to any kind of time where granularity is an issue.

2 Motivation

Consider the airline flight database depicted in Fig. 1. The database consists of two relations: *Flight_Departures* and *Vacations*. The *Flights_Departures* relation stores information about airplane flight departures. The flight departure time is recorded in the granularity of minutes.[2] The *Vacations* relation stores information about vacations, specifically, the days that make up a vacation. The temporal information in *Vacations* is ostensibly stored to the granularity of days, with each tuple recording a "period" of days rather than just a single day. The vacations listed in *Vacations* include traditional American holidays such as Labor Day, Christmas, and Thanksgiving. The Thanksgiving vacation is a four-day weekend beginning on the fourth Thursday in November.

A user, interested in flying home for Thanksgiving, queries this database to determine which flights leave during the Thanksgiving vacation. In SQL-92 [22], this query might be formulated as follows.

```
SELECT *
FROM Vacations, Flight_Departures
WHERE Vacation = 'Thanksgiving' AND
        Flight_Departures.At_Time OVERLAPS
            (Vacations.From_Time,
             Vacations.To_Time);
```

In this query, the user utilizes the temporal intersection operator, OVERLAPS, to determine which flights leave during the Thanksgiving vacation. The times participating in the OVERLAPS are at different granularities; flight times are in the granularity of minutes whereas vacation times are in the granularity of days. The SQL-92 query processor is unable to handle the mixed granularities—it would return a

2. In this paper, we do not consider *periodic time*, such as a flight departing at the same time each day [24], [30]. Our approach could be extended to encompass such situations.

minutes_in_days

Minute_Value	Day_Value
...	...
'1997-01-01 00:00'	'1997-01-00'
'1997-01-01 00:01'	'1997-01-01'
...	...
'1997-11-27 14:34'	'1997-11-27'
'1997-11-27 14:35'	'1997-11-27'
...	...
'1997-12-31 11:59'	'1997-12-31'
...	...

Fig. 2. A coversion table between minutes and days.

syntax error stating that the two arguments to OVERLAPS are of incomparable types. (Vacations.From_Time and Vacations.To_Time are of the DATE type; Flight_Departures.At_Time is of the TIMESTAMP type.)

The intent of the query is to select those flights that depart during the Thanksgiving vacation. To make progress in answering the query, the query processor needs information about the relationship between minutes and days. For example, it might know that each minute is contained in some day. With this extra information, the query processor can "scale" or convert the granularity of flight departure times from minutes to days, allowing the OVERLAPS to determine which flights leave during the Thanksgiving vacation.

The missing part of the puzzle is not that minutes can be related to days, rather, what is missing is the design of the *mechanism* that relates times in these two different granularities. Currently, users must manually provide this mechanism. For example, the user could create a conversion table [16] to map minutes to days, a fragment of which is shown in Fig. 2, and rewrite the query to utilize the table as follows. (Alternatively, the conversion could be effected by a user-supplied function, minutes_in_days().)

```
SELECT *
FROM Vacations, Flight_Departures,
     minutes_in_days AS C
WHERE Vacation = 'Thanksgiving' AND
       C.Day OVERLAPS (Vacations.From_Time,
       Vacations.To_Time) AND
       Flight_Departures.At_Time = C. Minute;
```

For temporal granularities, the "user does it all" solution has several disadvantages. First, queries that make explicit use of conversion tables are more difficult for users to formulate, and consequently, increase the likelihood of an incorrect query (e.g., suppose the user forgets the final conjunct; the query would be run to completion, but produce an incorrect result). Second, there is little the query optimizer can do to optimize the granularity conversions since it is unaware that those conversions are occurring; we discuss relevant optimizations in Section 8.6. (Note that these two drawbacks also apply to user-supplied conversion functions such as minutes_in_days().) Third, some of the conversion tables will be quite large. For example, a table to convert seconds to days over the range 1970 to 2000 A.D. would contain over a billion tuples. It would be better if most conversions were implemented by a (short) program fragment rather than a table. Finally, the user must predefine every possible conversion either as a table or a view. So for N granularities, $N^2 - N$ tables must be predefined, and when the user desires a new granularity, the user must create $2 \cdot N$ new conversions. In this paper, we present a strategy whereby a minimal set of conversions is specified, with the rest automatically constructed as needed during query evaluation. This strategy supports the addition of a new granularity via the specification of a conversion from some existing granularity.

For these reasons, we believe a better design is to build support for temporal granularity directly into a DBMS. In the rest of this paper, we describe how a DBMS can be engineered to automatically and efficiently construct relationships between granularities, such as days and minutes. We also explain how to get the query evaluator to perform the OVERLAPS in the granularity of days or in minutes. Finally, we quantify the cost of storing times in different granularities and the additional overhead on querying such times.

3 Model of Time

This section presents the model of time used in this paper, which expert readers will find to be mostly review material. SQL-92 terminology [21] is used for basic temporal concepts, consistent with the terminology proposed by the temporal database community [15]. We also utilize the terminology proposed for granularities [5], as well as the general model of time described there.

3.1 The Time Domain

The time domain is the set of time points used to define and interpret time-related concepts. Formally, a *time domain* is a totally ordered set T of time points with the ordering relation '$\leq$'. In our model, time can be either continuous, dense, or discrete. However, we adopt a discrete "view" of time, which we describe below.

3.2 A Discrete Image of Time

In this section, we summarize the relevant terminology and general framework for granularity presented elsewhere [5].

Portions of the time domain are "grouped" into aggregations called *granules* [33]. Specifically, a granule is a (not necessarily contiguous) subset of the time domain. A *granularity* is a mapping G from the integers to granules such that

1. if $i < j$ and $G(i)$ and $G(j)$ are nonempty, then each element of $G(i)$ is less than every element of $G(j)$,
2. if $i < k < j$ and $G(i)$ and $G(j)$ are nonempty, then $G(k)$ is nonempty, and
3. $G(0)$, the *origin* of G, is nonempty.

The first requirement implies that granules within a granularity are nonoverlapping and totally ordered, with the ordering inherited from the integers. The second ensures that the set of integers mapping to nonempty

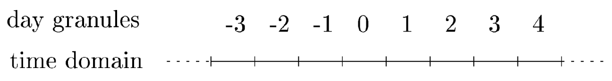


Fig. 3. The time-line at a granularity of days.

granules is contiguous. The third is mainly for convenience (see Section 8.2).

A granularity may cover only a subset of the time domain. There may be times that are less than those in any granule, or that are greater than those of any granule. The *extent* of a granularity is the set of time points from the earliest to the latest time points in any granule of that granularity. Within the extent of a granularity, there may be holes, time points that are not in the granularity. The *image* of a granularity is the union of the granules of the granularity. The image can be contiguous, or might have holes. If the image of a granularity is contiguous, it is equal to that granularity's extent. Finally, for each granule, $G(i)$, i is known as the *index* of the granule.

For example, the granularity of Gregorian days creates a discrete image of days. Fig. 3 shows a portion of the time-line grouped into granules belonging to the granularity of days. The figure depicts a continuous time domain partitioned into day granules. Each granule is a segment of the time-line. The granule's index is shown above each segment.

Granularities are related in the sense that the granules in one granularity may be further aggregated to form larger granules belonging to a *coarser* granularity. For example, as every Gregorian year is an aggregation of 365 or 366 Gregorian days, it follows that years is a coarser granularity than days. Similarly, Gregorian days may be considered to be a *finer* granularity than Gregorian years.

Let G and H designate two granularities. H is said to be *coarser* than G ($H \trianglerighteq G$) and G is said to be *finer* than H ($G \trianglelefteq H$) if for each granule $h \in H$, there exists a set of granules $S \subseteq G$ such that $h = \bigcup_{g \in S} g$. If G is finer or coarser than H, then the two granularities are said to be *comparable*.[3] (The *finer-than* relation employed here is also called the *groups-into* relation [5].)

For example, in the Gregorian calendar, years is a coarser granularity than months since every year is composed of a sequence of months. Conversely, days is a finer granularity than months since every month is composed of a sequence of days. But months are neither finer nor coarser than weeks since some months are not exactly composed of a sequence of weeks.

3.3 Textual Representation

In addition to the (integer) index, each granule is associated with a textual representation, a string, used for input and output, which is called the *label*. The label can be mapped to the index for input, and the index can later be mapped to this label for output. This mapping can be quite complicated, involving different languages and character sets [29].

3. This notion is more general than that found in the SQL-92 standard which effectively states that two values are mutually comparable only if they are of the same granularity [21, p. 24]. We examine how values of different granularities can participate in SQL operations in Section 4.

For expository purposes, in this paper, we will use the textual representations of the SQL-92 language to denote granules often subscripted by the name of the granularity. For SQL `DATE`s, at a granularity of days, the label consists of a four-digit year, followed by a two-digit month, followed by a two-digit day, separated with single hyphens. For example, the set of days in the Gregorian year 1997 A.D. is $\text{1997-01-01}_{days}, \text{1997-01-02}_{days}, \ldots, \text{1997-12-31}_{days}$.

3.4 Instants

An *event* occurs at a particular time point in T [15]. In general, the database cannot know this precise time, both because the measurement of the time is imprecise at the resolution of the time domain [7], and because the database cannot always accurately represent an arbitrary element of the time domain (e.g., when T is continuous).

For example, assume that a wristwatch reports that the current time is 3:45:23 P.M. This means that it is (was) sometime during that second, *but it is unknown exactly when*. The wristwatch can only measure to the accuracy of the granularity of seconds. In this sense, our model of time is faithful to many "real-world" temporal measurements.

We choose to model the time point at which an event occurs by an *instant timestamp* or just *instant*. An instant is a sequence of granules, called the *support*, together with an optional probability distribution on the support. The support indicates the possible granules during which the event occurs while the distribution records the probability that the event occurs during a particular granule. The support extends from a *lower support* granule, *l*, to an *upper support* granule, u, in a granularity, G, and is designated using the following notation:

$$l \sim u \equiv \{g \in G \mid l \leq g \leq u\}.$$

It is possible that the lower and upper supports are the same, indicating that the event occurs during a single granule. In this case, the instant is called a *determinate* instant. Otherwise, it is called an *indeterminate* instant.

Granularity and indeterminacy are two sides of the same coin. A general maxim is that a determinate instant is indeterminate with respect to *all* strictly finer granularities. In other words, for any determinate instant, $g \in G$, and any finer granularity H, there exists an indeterminate instant $l_H \sim u_H$ such that $g = l_H \sim u_H$. For instance, suppose we record that a plane took off on 1997-06-12_{days}. This is a determinate instant at the granularity of days; we know the exact day that the plane departed. At the granularity of minutes, the departure time is indeterminate, since we did not record the exact minute that the plane departed. We only know that it left sometime during $\text{1997-06-12} : 00 : 00_{minutes} \sim \text{1997-06-12} : 23 : 59_{minutes}$.

Conversely, an indeterminate instant is determinate with respect to *some* coarser granularity. In other words, for any indeterminate instant $l \sim u$, there exists a determinate instant h in some coarser granularity H, such that $l \sim u \subseteq h$. For example, suppose we record, at the granularity of hours, that a flight departs sometime between 2 P.M. and 4 P.M. on 1997-06-12. At the granularity of days, months, and years, the flight departs wholly within a single granule: $1997\text{-}06\text{-}12_{days}$, $1997\text{-}06_{months}$, and 1997_{years}, respectively.

While it is important to recognize that instants are specified only to the precision of a particular granularity, it is equally important to choose the correct granularity. Sometimes, for reasons of linguistic convenience, humans under-specify a time, that is, they specify a time in a very coarse granularity when the time that it signifies is actually known or intended to be at a very fine granularity. For example, a ship schedule states that a ship departs at 3 P.M. The time of the ship departure is given in the granularity of hours, but "3 P.M." is (probably) accurate to a much finer granularity, specifically to the granularity of minutes.

3.5 Periods and Intervals

A period is a contiguous subset of the time domain. A *period of granularity* G, encoded with the indexes of a pair of granules g_1 and g_2, is the set of granules in G between g_1 and g_2, under the constraint that $g_1 \leq g_2$. We assume that both the starting and terminating granules are in the same granularity.

An *interval* is a signed integral number of granules in some granularity, that is, an amount of time with known length but no specific starting or ending instants. For example, the interval 6_{days} is known to have a duration of six days, but can refer to any block of six consecutive days. An interval can be either positive, denoting forward motion in time, or negative, denoting backwards motion in time.

Periods and intervals can also be indeterminate. An indeterminate period is a period that has indeterminate bounding instants. An indeterminate interval is an interval with a partially-known duration; however, we know that the interval is at least as long as the lower support and no longer than the upper support.

3.6 Summary of the Data Model

The theme for our model of time is that users manipulate a discrete image of a time-line that is itself possibly continuous, dense, or discrete. The discrete image is a by-product of modeling temporal information at a given granularity. A granularity is a grouping of the time domain; each group is called a granule. Granules model durationless temporal values, that is, time points, that are located sometime during that particular granule. Periods model temporal values with duration that span a range of the time domain. Intervals model unanchored durations of the time domain. Indeterminate instants, periods, and intervals model partially-known temporal information. Granularity and indeterminacy are related issues. All instants (periods, intervals) are indeterminate at finer granularities and determinate at some coarser granularities.

4 Granularity in Operations

The granularity of time values impacts the semantics of expressions involving those values. For instance, what happens when we compare a granule at the granularity of a day to one at the granularity of a minute? In this section, we discuss support for granularity in temporal operations.

4.1 Conversions

We propose two operations to convert time values between granularities: *scale* and *cast*. We focus on instants in this discussion, but these functions can easily be extended to periods and intervals. The conversion function $scale(g, H)$ takes a (possibly indeterminate) instant $g = l_G \sim u_G$ in granularity G and a granularity H and returns the smallest (possibly indeterminate) instant $h = l_H \sim u_H$ such that $l_G \sim u_G \subseteq l_H \sim u_H$, and returns *invalid* if no such instant exists. (Here, we specify the conversion functions in terms of the granules themselves, rather than their indexes.) If $H \trianglelefteq G$ or $G \trianglelefteq H$, then the '$\subseteq$' will in fact be an equality.

A scale operation that converts an instant from a coarser to a finer granularity usually produces an indeterminate instant, even when applied to a determinate instant (where $l_G = u_G$). For various reasons, a user may not want an indeterminate result. Instead, a user might desire a result that is determinate when applied to a determinate value, even though that result might not be strictly consistent with the input value. To meet these user needs, we propose a new operation, called *cast*, that allows one to "create" information.

The cast operation is similar to scale but produces a determinate instant when applied to a determinate instant. The cast of a determinate instant is the lower support of the scale of the instant. For example, to cast a determinate instant from a coarser to a finer granularity, cast first scales the instant, resulting in an indeterminate instant. From that indeterminate instant, it returns the first granule, a determinate instant, as the result. In effect, for any determinate instant, cast assumes that the modeled time point is contained in the first granule at all finer granularities. When cast is applied to an indeterminate instant it separately casts both the upper and lower supports, as though they were determinate instants.

The conversion function $cast(g, H)$ returns the instant $h = l_H \sim u_H$ in H such that $l_h \in \min(scale(l_G, H))$ and $u_H \in \min(scale(u_G, H))$. Note that if $l_G = u_G$ then $l_H = u_H$, ensuring that a cast of a determinate instant always produces a determinate instant.

The mapping functions can be combined as shown in the following examples.

$$\begin{aligned}
&scale(1997_{years},\ days)\\
&\qquad = 1997\text{-}01\text{-}01_{days} \sim 1997\text{-}12\text{-}31_{days}\\
&scale(scale(1997_{years},\ days),\ months)\\
&\qquad = 1997\text{-}01_{months} \sim 1997\text{-}12_{months}\\
&cast(1997_{years},\ days)\\
&\qquad = 1997 - 01 - 01_{days}\\
&cast(cast(1997_{years},\ days),\ years)\\
&\qquad = 1997_{years}\\
&scale(cast(1997_{years},\ days),\ months)\\
&\qquad = 1997\text{-}01_{months}
\end{aligned}$$

Observe that some combinations of mappings result in the identity function on subsets of the time domain. That is, the support of the input instant equals the support of the output instant. These sequences of mappings are called "information-preserving" sequences, in the sense that they lose none of the original instant's precision.

We are now in a position to specify the various proposed semantics for binary instant operations over different granularities. Let $g \in G$ and $h \in H$ be (determinate or indeterminate) instants at the indicated granularities, $\odot$ be a binary instant operation or predicate, F be a granularity that is finer than both G and H, and C be a granularity that is minimally coarser than both G and H. We express the semantics in terms of operators over single granularities.

Mismatch. Give a mismatched granularity error [1].

Left-operand semantics. Perform the operation at the granularity of the first operand. This is reminiscent of the assignment operator in many strongly typed languages, which casts the value of the right-hand side to the type of the left-hand side.

$$g \odot h = g \odot \mathit{scale}(h, G).$$

Right-operand semantics. Perform the operation at the granularity of the second operand. This is reminiscent of some expressions in C++, e.g., `7/2.0`, which converts the value of the left-hand side of the division operator to the floating point type, because the right-hand side is a floating point number.

$$g \odot h = \mathit{scale}(g, H) \odot h.$$

Finer semantics. Perform the operation to the finer granularity [7], [25], [33]. If the two granularities are incomparable (neither is finer than the other), then perform the operation to a granularity finer than both arguments.

$$g \odot h = \begin{cases} g \odot \mathit{scale}(h, G) & \text{if } G \text{ is finer than } H \\ \mathit{scale}(g, H) \odot h & \text{if } G \text{ is coarser than } H \\ \mathit{scale}(g, C) \odot \mathit{scale}(h, C) & \text{otherwise.} \end{cases}$$

Coarser semantics. Perform the operation to the coarser granularity [4], [23]. For incomparable granularities, perform the operation to a granularity that is minimally coarser. This approach avoids adding indeterminacy not already present, but is, in some sense, the most conservative possibility, as information at the finer granularity is discarded.

$$g \odot h = \begin{cases} \mathit{scale}(g, H) \odot h & \text{if } G \text{ is finer than } H \\ g \odot \mathit{scale}(h, G) & \text{if } G \text{ is coarser than } H \\ \mathit{scale}(g, C) \odot \mathit{scale}(h, C) & \text{otherwise.} \end{cases}$$

It turns out that SQL-92 adopts each of these in particular contexts, as discussed further in Section 6.3.

Except for generating a mismatch error, the operands are first converted to the same granularity and then the operation is carried out, usually on the indexes of the granules which are integers. By converting both operands to a common granularity, the complete set of temporal operations currently available in a query language (c.f. the instant, period, and interval operations in TSQL2 [26]) can be utilized unchanged.

In any of the above semantics, *cast* may be used in place of *scale*. The drawback of using *cast* is that the finer conversion function discards some temporal information. The binary operator may then return a perhaps unexpected result (though one consistent with the semantics). For example, $\text{1997-01}_{months} < \text{1997-01-15}_{days}$ translates to three possible comparisons under the various semantics; one of which evaluates to *true*.

- $\text{1997-01-01}_{days} < \text{1997-01-15}_{days} = true$ (*cast* using Right-operand or Finer semantics.)
- $\text{1997-01}_{months} < \text{1997-01}_{months} = false$ (*cast* or *scale* using Left-operand or Coarser semantics.)
- $\text{1997-01-01}_{days} \sim \text{1997-01-30}_{days} < \text{1997-01-15}_{days} = maybe$[4] (*scale* using Right-operand or Finer semantics.)

4.2 Scaling Mass Functions

The probability mass function gives the probability that the instant is located within a given granule. Since a scale operation (whether regular or not) modifies the size and number of granules in the support of the distribution, the scale also changes the mass function. Each mass function is described, in the implementation, as a function on a domain $[0, 1]$. In scaling from a finer to a coarser granularity, the mass of each fine granule is effectively added to the mass of all the other fine granules that belong to a given coarse granule. For example, suppose an indeterminate instant with a seven day support (from Sunday through Saturday) and a *uniform* mass function is scaled to the granularity of weeks. In the resulting instant, the probability that the instant is located during each day, a probability of $\frac{1}{7}$, is accumulated to give the probability that the instant is located during the given week, a probability of 1. In scaling from a coarser to a finer granularity, the mass of each coarse granule is dispersed. (We show elsewhere that it is in practice more efficient to shrink or stretch the mass during the comparison operation, rather than during the scale operation [11].)

4.3 Scaling Intervals

To scale a period, the instants that start and end the period are scaled separately. Scaling intervals, however, is slightly more complicated.

An interval is an unanchored duration. In our model of time, it is encoded as a count of granules in some granularity. The interpretation of an interval is that it is a duration that *necessarily* displaces any instant by the represented number of granules. For example, an interval of 1_{days} represents a duration that when added to an instant at the granularity of days, will displace that instant by one day, e.g, $\text{1997-12-30}_{days} + 1_{days} = \text{1997-12-31}_{days}$.

We observe that an interval of 1_{days} also represents a duration that when added to an instant at the granularity of months, could displace that instant by 0 or 1 months. In the

4. *Maybe* is neither *true* nor *false*. The semantics of '<' on indeterminate operands is described elsewhere [11].

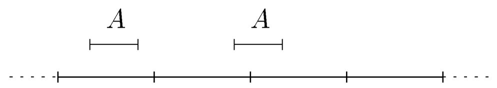


Fig. 4. Interval A can be placed within a single granule or spanning two granules.

above example, the instant could be moved from the month of December to the month of January. Note that the interval of 1_{days} could also displace an instant into the next year. Imagine taking an interval and placing it anywhere along a time-line that is partitioned into granules. Depending upon where we place the interval, it will cross more or fewer granules as shown in Fig. 4. Even the smallest interval can cross at least one granule boundary.

A second observation is that some intervals span different numbers of finer granules. For example, 1_{months} may represent anywhere from 28 to 31 days. Twenty-eight days will displace any instant in February into the next month, but 31 days are sometimes needed to displace an instant in August into the next month.

For both of these reasons, scaling an interval may result in an indeterminate interval. An indeterminate interval $a \sim b$ in granularity G implies that the length of the interval is a number between a and b of granules in G. An indeterminate interval may result when scaling from coarser to finer or from finer to coarser. Below, we give some examples to illustrate scaling an interval.

$$\begin{aligned}
scale(1_{days},\ years) &= 0_{years} \sim 1_{years} \\
scale(1_{days}, months) &= 0_{months} \sim 1_{months} \\
scale(1_{days}, days) &= 1_{days} \\
scale(1_{days}, hours) &= 24_{hours} \\
scale(1_{days}, minutes) &= 1440_{minutes} \\
scale(1_{months}, days) &= 28_{days} \sim 31_{days}.
\end{aligned}$$

The actual mechanics of scaling an interval is a variation of that for scaling an instant.

5 Piecemeal Specification of Granularities

Each granularity, G, can always be specified by giving the function $G()$ that maps indexes to granules in G, that is, to subsets of the time domain. For many granularities, however, it would be helpful if granularities could be specified with respect to other granularities rather than to the underlying time domain. For example, suppose the time domain is UTC seconds. Gregorian days, weeks, months, and years could be specified in this time domain, but each specification would have to take into account complicated leap seconds adjustments. An alternative, modular approach would be to specify days in terms of seconds, weeks, and months in terms of days, and years in terms of months. Since the purpose of the granularity specifications is to support conversions *between* granularities, for many conversions knowing the granularity to the precision of the underlying time domain is unnecessary.

In this section, we advocate specifying granularities via mappings between pairs of granularities, rather than specifying granularities directly via their index functions.

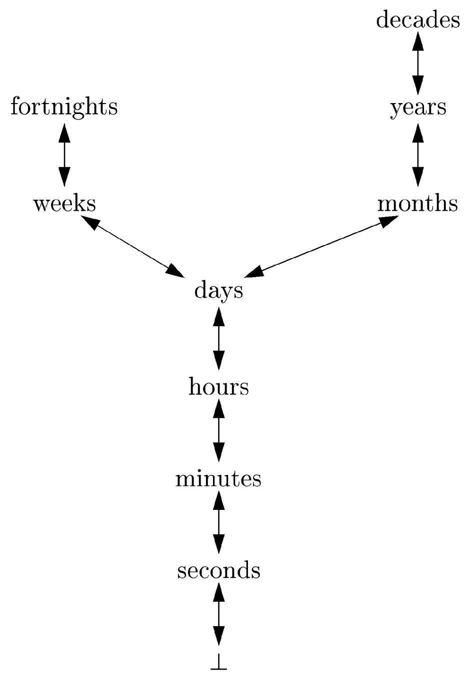


Fig. 5. A Gregorian calendar granularity graph.

From a sufficiently complete set of these mappings, the system can deduce any required granularity information.

Elaborating on our approach, the user (or database administrator) specifies granularities by providing the conversion functions (*scale* and *cast*) between some pairs of granularities. This set of conversion functions describes a directed graph called the *granularity graph*. Each node in the graph represents a granularity. An edge from G to H indicates that the user has supplied a function to convert from G to H. A granularity graph for the granularities in the Gregorian calendar is shown in Fig. 5. In the figure, all mappings have been supplied in both directions between the indicated pairs of granularities.

To allow scales and casts between arbitrary granularities, we must place some restrictions on the granularity graph. First, the graph must contain a finest or bottom granularity, denoted $\perp$. The granules in $\perp$ are called *chronons* [15]. Second, for every granularity G in $\mathcal{G}$, there must exist a path $(G = G_0, G_1, \ldots, G_k = \perp)$ from G to $\perp$ such that $G_i \overset{\triangleright}{-} G_{i+1}$ for all $0 \leq i < k$. Finally, for every granularity G in $\mathcal{G}$, there must exist a path $(\perp = G_0, G_1, \ldots, G_k = G)$ from $\perp$ to G such that $G_i \overset{\triangleleft}{-} G_{i+1}$ for all $0 \leq i < k$.

We do not require the set of granularities in the graph to form a lattice with respect to the finer-than relation. This typical assumption [31] is violated if two granularities in the calendar have different extents and, thus, no unique least upper bound (LUB). In rarer cases, two granularities may not have a unique greatest lower bound (GLB). For example, the following combination of granularities does not have a unique GLB: solar days (the day starting at midnight), civil days (the day starting at noon), hours, and Chinese calendar k'o (roughly ninety minute divisions of a day). Hours are finer than civil and solar days, as are k'o, but hours and k'o are unrelated. In either case, an "artificial" GLB or LUB could be constructed and inserted into the graph to create a lattice. In our approach, we simply do not require real-world granularities to fit neatly into a lattice. We do, however, assume the existence of a bottom within a single granularity graph ($\perp$ is finer than every granularity), for reasons discussed in Section 5.2.

Granularities in a granularity graph form a *calendar* [29]. A calendar is a software package that has two primary purposes. First, it inputs and outputs temporal values as character strings, that is, the labels of the values by translating those labels to and from granularity indexes. We discuss this mapping in Section 7. Second, a calendar provides all the functionality necessary to support granularities. In this section, we describe the necessary functionality in detail. We envision that some calendars would be provided by the DBMS vendor (an example being the calendar supporting the legacy SQL-92 granularities such as DAY), others might be provided by the database users (i.e., the database administrator's (DBA's) staff, an example being a calendar tied to the company's fiscal year vagaries), and still others might be supplied by third-party vendors (an example being an Astronomy calendar).

In this paper, we will use three calendars: the Gregorian calendar, a Business calendar, and an Astronomy calendar. We assume that the reader is already familiar with the Gregorian calendar (a variant of which is included in SQL-92 [21]). The Business calendar is a prototypical calendar for tax or payroll applications. In the Business calendar, days are the same as in the Gregorian calendar, but the Business calendar has a five day (work) week. The Business calendar year is divided into four quarters, Fall (starting on the Gregorian date October 1), Winter (starting January 1), Spring (starting April 1), and Summer (starting June 1). For tax purposes, the Business calendar year starts with the Fall quarter. The origin of the Business calendar is Fall, 1990 (the founding of the corporation supplying the calendar). The Astronomy calendar is very different from the Business calendar. Those readers familiar with Julian[5] or modified Julian dates will recognize the Astronomy calendar. The Astronomy calendar year has 365.25 days. The origin of the Astronomy calendar is noon on January 1, 4713 B.C., which is a synchronization point for various long-term celestial cycles. The Gregorian calendar date June 24, 1994 is $2449349_{astronomy\_days}$. The Astronomy calendar also has centuries, which are precisely 36525 days long.

5.1 Building the Granularity Graph

There are many methods that could be used to build the granularity graph. Perhaps the easiest is to assume that the graph is predefined; this is the approach adopted by the SQL-92 standard [22]. We feel that such an assumption is unrealistic. Instead, we outline a method for building the graph using a specification provided by the DBA. We note in passing that the MULTICAL system [29] is a concrete realization of this approach to supporting multiple calendars in a conventional DBMS.

Each calendar has a *specification file* which is parsed when the DBMS is configured by the DBA. Granularity descriptions are included in the specification file. A granularity is described as a further grouping of some other granularity using either a *regular* mapping, that is, each granule in the coarser granularity is composed of a fixed number of granules at the finer granularity (cf. Section 8.1), or as an *irregular* mapping, which is necessarily more complicated. The specification file also declares the origin and extent of each granularity.

The Gregorian calendar specification file describes the granularity of hours as a regular grouping of minutes by asserting that there are exactly sixty minutes in every hour. In the *decorated granularity graph* (Fig. 6), this information is represented by two edges: one from minutes to hours labeled '*t div 60*' and another from hours to minutes labeled '$t * 60$'. The edge label '*t div 60*' represents a *function* that converts minutes to hours. To convert minutes to hours, the time, t, is divided (integer division) by *60*. What may be surprising to some readers is that a minute in the Gregorian calendar is not always 60 seconds. Due to leap second adjustments, it may be 59 or 61 seconds. Thus, the specification file states that an irregular mapping exists between minutes and seconds. The calendar provides functions for all irregular mappings. Although the *seconds_to_minutes(t)* function is simple (we describe elsewhere how to process Gregorian calendar dates with leap seconds [10]), some irregular mappings may be quite complex. The *days_to_months(t)* mapping must accommodate months that have different numbers of days, as well as leap years. Finally, for irregular mappings, the calendar also provides functions to convert intervals, e.g., *interval_minutes_to_seconds(i)*. Hence, a calendar is a specification file that enumerates the names of the granularities and describes the mappings between them, and a collection of mapping functions, to be linked with the DBMS.

5.2 Combining Calendars

If a user wishes to compare instants in granularities that are not within the same calendar, then a larger granularity graph containing the granularities from multiple calendars must be constructed. This is done by adding mappings between granularities in the different calendars. Often, the bottom granularity in one of the calendars is finer than the bottom granularities in the other calendars. If this is the case, it is only necessary to provide enough mappings to create a path from this finest bottom to and from each of the other bottoms. This finest bottom then becomes the bottom of the large calendar. If such a granularity does not exist, then a new granularity, finer than all existing bottom granularities, must be constructed along with mappings between it and the existing bottom granularities.

For example, it is easy to combine the Astronomy, Gregorian, and Business calendars. The bottom granularity, $\perp$, of the Gregorian calendar is finer than the bottoms of the Astronomy and Business calendars. Thus, we need only provide mappings to create paths between $\perp$ and astronomy_day_hundredths and between $\perp$ and business_-days. The former is accomplished by a simple regular mapping between seconds and astronomy_day_hundredths; the latter, by a trivial mapping between days and business_days. Additional mappings may be specified to increase performance.

5. The Julian calendar we refer to here was developed in the sixteenth century by the French literary scholar Joseph Justus Scaliger, and is distinct from the Julian calendar established by Julius Caesar in 45 B.C.E. [8].

$$\begin{tikzcd}
\text{decades} \arrow[d, shift right, "t * 10"'] \\
\text{years} \arrow[u, shift right, "t\ div\ 10"'] \arrow[d, shift right, "t * 12"'] \arrow[dddddd, bend left=60, "quick\_years\_to\_seconds(t)"] \\
\text{months} \arrow[u, shift right, "t\ div\ 12"'] \arrow[d, shift right, "months\_to\_days(t)"'] \\
\text{days} \arrow[u, shift right, "days\_to\_months(t)"'] \arrow[d, shift right, "t * 24"'] \\
\text{hours} \arrow[u, shift right, "t\ div\ 24"'] \arrow[d, shift right, "t * 60"'] \\
\text{minutes} \arrow[u, shift right, "t\ div\ 60"'] \arrow[d, shift right, "minutes\_to\_seconds(t)"'] \\
\text{seconds} \arrow[u, shift right, "seconds\_to\_minutes(t)"'] \arrow[d, shift right, "t * 1"'] \\
\perp \arrow[u, shift right, "t\ div\ 1"']
\end{tikzcd}$$

Fig. 6. A partial view of the decorated granularity graph in the Gregorian calendar.

The granularity graph shown in Fig. 7 was constructed by combining the Astronomy, Gregorian, and Business calendars. Several additional mappings, e.g., astronomy_days to hours, have also been defined. Presumably, these provide improved performance for common conversions.

5.3 Casts and Scales Via the Granularity Graph

The user may want to convert from one granularity to another even when there is no direct mapping to perform the particular conversion. In this case, we use the provided mappings between granularities to implement the desired mapping. The DBMS finds a path in the granularity graph between the granularities and performs the sequence of mappings along this path. The problem is that some compositions of mappings may not result in a correct conversion.

For example, to convert astronomy_days to months, we may follow the path (astronomy_days, hours, days, months), or (astronomy_days, astronomy_years, years, months), or (astronomy_days, astronomy_day_tenths, astronomy_day_hundredths, seconds, minutes, hours, days, months), etc. But scaling along the second path will always result in an indeterminate instant (with a support of at least 12 months) even if the astronomy_day being scaled is determinate. This is incorrect since a determinate astronomy_day should scale to a determinate month. In scaling from astronomy_days to astronomy_years, we lose information about the original instant, resulting in a possibly incorrect final mapping. We need a method of identifying those paths that always result in correct mappings: the *correct paths* (generally there are several).

A correct conversion path is a path $(G = G_0, G_1, \ldots, G_k = H)$ in the granularity graph $\mathcal{G}$ such that for all instants $l_0 \sim u_0$ in G_0, $l_k \sim u_k = \mathit{scale}(l_0 \sim u_0, G_k)$, where $l_k \sim u_k$ is defined inductively by

$$l_{i+1} \sim u_{i+1} = \mathit{scale}(l_i \sim u_i, G_{i+1})$$

for all $i \in \{0 \ldots k-1\}$.

We wish to identify those paths from G to H that are correct conversion paths. A *V-path* is a path $(G_0, G_1, \ldots, G_k)$ in the granularity graph $\mathcal{G}$ such that $G_0 \trianglerighteq G_1 \trianglerighteq \cdots \trianglerighteq G_p \trianglelefteq G_{p+1} \trianglelefteq \cdots \trianglelefteq G_k$ for some $p \in \{0 \ldots k\}$. Intuitively, a V-path goes from a granularity G_0 to a finer granularity G_p, then up to a coarser granularity G_k. It is called a V-path since it appears in the shape of a V in a drawing of the granularity graph where finer granularities are below coarser ones. Note, however, that "straight-line" paths are also V-paths, with $G_p = G_0$ or $G_p = G_k$.

Since there is a path in the granularity graph from the finest granularity $\perp$ to every other granularity G through successively coarser granularities and a path from G to $\perp$ through successively finer granularities, there is at least one V-path between every pair of granularities (the one with $G_p = \perp$). There are often several V-paths between two granularities. It turns out that *all* V-paths between G and H are correct conversion paths.

Lemma 5.1. *Any V-path is a correct conversion path.*

Proof. Let $(G_0, G_1, \ldots, G_k)$ be a V-path and G_p be the finest granularity in the path. Let $l_0 \sim u_0$ be an instant in granularity G_0.

Let $l_j \sim u_j$ be the instant in G_j which is the result of the composition of mappings along the path $(G_0, \ldots, G_j)$ applied to $l_0 \sim u_0$. We want to show that $l_k \sim u_k$ is the smallest instant in granularity G_k that contains the instant $l_0 \sim u_0$. In particular, we want to show that l_k contains $\min l_0$ (the smallest time point in l_0) and u_k contains $\max u_0$. We prove the above statement for the lower support l_0. A similar argument establishes the result for the upper support u_0.

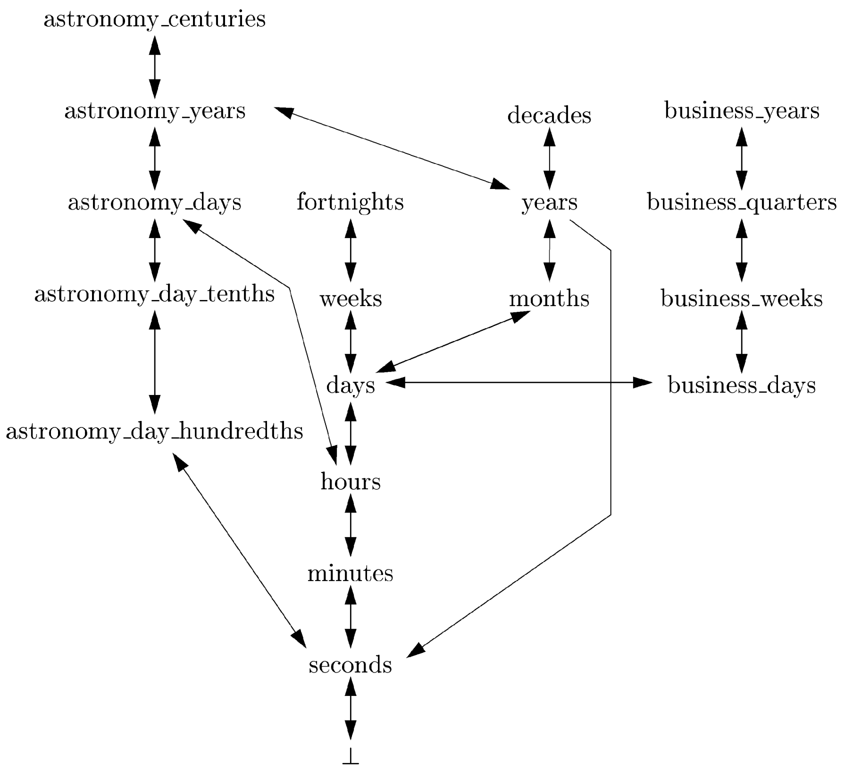


Fig. 7. A multicalendar granularity graph.

Each of the individual mappings is assumed to be correct. Thus, since G_{j+1} is finer than G_j for $j < p$, $l_{j+1} \subseteq l_j$ and, in fact, $\min l_{j+1} = \min l_j$. By induction, $\min l_p = \min l_0$.

Note that the support of the instant in G_0 equals a union of granules in the finer G_p. However, in scaling from G_j to G_{j+1} for $p \leq j < k$, from finer to coarser, we may find that some support granules in G_j are not contained in *any* granule in G_{j+1}. This means that some subset of $l_0 \sim u_0$ is not in the image of the granularity G_{j+1}. If this is the case, the mapping from G_j to G_{j+1} returns *invalid* and the composition as a whole also returns *invalid* for this path. This is correct. Since G_{j+1} is finer than G_k, the same subset of $l_0 \sim u_0$ that is not in the image of G_{j+1} is not in the image of G_k. Thus, the scale of $l_0 \sim u_0$ to G_k is indeed invalid.

If all support granules in G_j are in the image of G_{j+1} for all $p \leq j < k$, then, in particular, $l_j \subseteq l_{j+1}$. By induction, $l_p \subseteq l_k$, and $\min l_0 = \min l_p \in l_k$, which shows that the lower support is correct. □

It is possible that paths other than V-paths are correct. For instance, the user may specify a function that converts directly from granularity G to granularity H even though G and H are not comparable. This simple case of a single-edge path is easy to detect and, by assumption, is correct. We consider these paths when calculating correct conversion paths between granularities. We do not consider other paths from G to H (W-paths, etc.) because these paths are generally incorrect even though they may provide correct conversions in special cases.

6 Accommodating Mixed Granularities in SQL

To this point, we have focused on the formal underpinnings of granularities and on how to effectively specify a collection of granularities. In this section, we propose a concrete query language syntax and semantics to support mixed granularities. Implementation is discussed in the next section. The proposed support for mixed granularities is based on the SQL-92 language standard [21], [22].

6.1 Column Definitions

In SQL-92, the `CREATE TABLE` statement defines relation schemas. Columns (attributes) of determinate instants (`DATE`, `TIME`, and `TIMESTAMP`) or interval values (`INTERVAL`) can can be defined, but lacking from these column definitions is a general way to specify the granularity in the presence of user-defined granularities (SQL-92 provides only a fixed set of granularities: year, month, day, hour, minute, second, and fractions of a second). We propose to allow the user to specify an extent and granularity in a column definition. Like SQL-92, we assume that a column definition establishes a data-type that is the same for every value in that column, so all values in a column are stored to the same granularity. In our example database, all flight times are stored to the granularity of a minute rather than some being stored to finer granularities, such as a second or millisecond. Individual times known to coarser granularities (e.g., a day) can be stored by making the indeterminacy explicit (e.g., the flight leaves between the first and last minutes during that day).

6.2 Temporal Literals

Instant, period, and interval literals are syntactically delimited by single quotation marks, '$\cdots$'. A calendar (either the default Gregorian calendar or one provided by the user) translates whatever comes between the delimiters into an instant, period, or interval value [29]. We assume that the calendar also decides the granularity of that value unless the granularity is explicitly specified with the literal.

6.3 Granularity in Operations

As discussed in Section 4, support for mixed granularity operations rests on the ability to translate instants between granularities.

SQL-92 adopts the following semantics in each indicated context.

Mismatch. Give a mismatched granularity error. In SQL-92, this semantics is employed by all variants of comparison between temporal values (e.g., =, <) [21, Subclause 8.2, Syntax Rule 2, p. 169], `BETWEEN` [21, SC 8.3, SR 3, p. 172], `IN` [21, SC 8.4, SR 4, p. 173], `ANY`, `ALL`, and `SOME` [21 SC 8.7, SR 2, p. 180], and `OVERLAPS` [21, SC 8.11, SR 2, p. 186].

Left-operand semantics. Perform the operation at the granularity of the first operand. In SQL-92, this semantics is employed by $\langle \text{datetime} \rangle + \langle \text{interval} \rangle$ *and* $\langle \text{datetime} \rangle - \langle \text{interval} \rangle$ [21, SC 6.14, SR 3, p. 132].

Right-operand semantics. Perform the operation at the granularity of the second operand. In SQL-92, this semantics is employed by $\langle \text{interval} \rangle + \langle \text{datetime} \rangle$ [21, SC 6.14, SR 3, p. 132], retaining the symmetry of '+' in the presence of multiple granularities, since $\langle \text{datetime} \rangle + \langle \text{interval} \rangle$ has an identical semantics as $\langle \text{interval} \rangle + \langle \text{datetime} \rangle$: the operation is performed to and the result is given in the granularity of $\langle \text{datetime} \rangle$.

Finer semantics. Perform the operation to the finer granularity. In SQL-92, this semantics is employed by $\langle \text{interval} \rangle + \langle \text{interval} \rangle$ and $\langle \text{interval} \rangle - \langle \text{interval} \rangle$ [21, SC 6.15, SR 3c, p. 135]. This approach retains the symmetry of '+' on intervals. SQL-92 does not support indeterminacy; it (somewhat arbitrarily) uses the first granule in the support.

To support these semantics, we add two operations, paralleling *scale* and *cast* that convert temporal values. The syntax of these new operations is as follows. (We note in passing that SQL-92 already includes a `CAST` operation; our proposal is a slight extension of this existing construct.)

$$\texttt{SCALE}(\langle \text{operand} \rangle \texttt{ AS } \langle \text{granularity} \rangle)$$
$$\texttt{CAST}(\langle \text{operand} \rangle \texttt{ AS } \langle \text{granularity} \rangle)$$

The first argument to each is an operand. A temporal operand can either be a literal, a column variable, or an expression. If the operand is a literal, the granularity of that literal is given by the calendar that parses the literal. If the operand is a column variable, the granularity of that variable is given by the column definition (specified in the schema). Finally, if the operand is an expression, the granularity of the operand is the granularity of the result of that expression determined during semantic analysis. The second argument is the target granularity. For example,

```
SCALE(Flight_Departures.At_Time AS DAY)
```

scales the instants in the `At_Time` column of the `Flight_Departures` table from minutes to days.

As discussed in Section 4, SQL-92 does not permit comparison operators to be applied to values of different granularities. For other operations, SQL-92 uses, inconsistently, left operand semantics, right operand semantics, and finer semantics. However, a user who desires a different temporal semantics can explicitly insert scale or cast operations. User-specified granularity conversions supersede the implicit conversion operations indicated by the SQL-92 semantics.

When SQL-92 needs to implicitly convert to a different granularity (e.g., for $\langle \text{datetime} \rangle + \langle \text{interval} \rangle$, which uses left operand semantics), the default is to use `CAST`. The default translation operations may be globally overridden by a `SET` statement. To change the default operation from cast to scale, one would use the following.

```
SET SCALE AS DEFAULT;
```

6.4 Processing the Example Query

We use the SQL constructs proposed here to process the example query given in Section 2. First, the query is rewritten to the following query, which effects the coarser operand semantics (as mentioned above, SQL returns an error for the original query).

```
SELECT *
FROM Vacations, Flight_Departures
WHERE Vacation = 'Thanksgiving' AND
  SCALE(Flight_Departures.At_Time AS DAY)
  OVERLAPS
        (Vacations.From_Time,
         Vacations.To_Time);
```

The scaled dates of flight # 200 and of flight # 653 overlap the period of November 24 through November 28, so only these two tuples appear in the answer.

In summary, with these extensions the original SQL-92 semantics is retained, both for the predefined SQL-92 granularities and for user-defined granularities, while allowing the user to define a specific semantics for an expression via appropriately placed `CAST` and `SCALE` operations, and extending the semantics to support temporal indeterminacy.

7 Timestamp Encoding

We now describe representations for each of the three basic modeling entities: instants, periods, and intervals. We focus on the determinate formats in this paper; the indeterminate representations are presented elsewhere [11]. These indeterminate representations are more complex, as they in some cases avoid storing both indexes.

An instant timestamp specifies a granule containing the time of an occurrence. The SQL-92 instant timestamp, specifically, the `TIMESTAMP` format, is a record that has separate fields for the year, month, day, hour, minute, second, and "fractional seconds" of an instant [21]. The format can store an instant known to these granularities

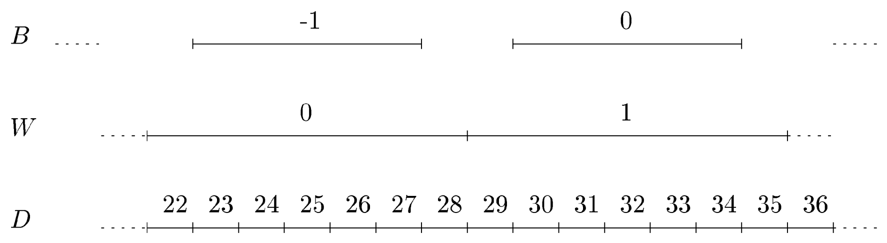


Fig. 8. Regular mapping between granularities.

only; it cannot store an instant known to the granularity of, say, weeks, or to a non-Gregorian granularity such as astronomy_day_tenths. Furthermore, common operations on such a representation are inefficient [9]. To add an interval to an instant, each field must be added separately with carries performed between the fields.

We propose replacing the DATE, TIME, and TIMESTAMP formats with a simpler format. The format has two fields: a *flags* field and an *index* field. The flags field is three bits in size, and is used to differentiate determinate instants from indeterminate instants and special values such as *beginning* and *forever* [26]. The index field stores the granule index, a signed integer value, which is a count, in granules, of the distance from the granularity origin to the instant. Since all values in a column have the same granularity (as specified in the SQL semantics), the granularity is stored with the schema rather than stored with the value, thus eliminating the need for a field to store the name of the granularity.

It is possible for the granules (as subsets of the time domain) in two granularities to be identical, but to be associated with different indexes in the two granularities, if granularities are allowed to have different origins. $23_{business\_days}$ might not be the same granule as 23_{days}; instead it might be 728316_{days} (assuming that the Gregorian day origin is January 1, 1 C.E. and the Business day origin is January 1, 1994 C.E.).

Such congruent granularities can be used to limit the size of timestamps. A user who wants to store times in the current decade to the granularity of a second can use a one word format by using a granularity, congruent with seconds, but with an origin at the current decade boundary. The extent is then only ten years, which can be stored in a one-word timestamp. By relocating the origin via congruent granularities, the user can employ one-word timestamps for most applications.

A period may be encoded by its delimiting timestamps, and an interval may be encoded as simply a count (positive or negative) of granules. This approach finesses an awkward distinction in SQL-92 between *year-month* intervals and *day-time* intervals [22]. This distinction arose because SQL-92 intervals contain a range of fields (e.g., year, month, day). So, year-month intervals can contain only a year value, only a month value, or both, and a day-time interval can contain only day, hour, minute, and second values. Because we don't know how many days there are in a month, an interval of "3 years, 4 months, and 5 days" is ill-defined, and is thus disallowed in SQL-97. Month and day fields cannot be coresident in an SQL-92 interval.

In our approach, an interval is always an integral number of granules in a specified granularity. Because the nonregularity of days in month is dealt with in the conversions (see Section 6.3), rather than in the values, there is no longer any need to make this distinction between two classes of intervals.

8 Evaluating Cast and Scale

There are two problems that arise in performing a cast or scale operation: determining a correct and efficient conversion path in the granularity graph between the source and the destination granularity (generally done during semantic analysis of the query), and performing the cast or scale on a particular instant. We first identify a class of efficient mappings. We then discuss algorithms that find the cheapest correct path and, finally, present how to use these paths effectively.

8.1 Regular Mappings

In some cases, the conversion between granularities G and H ($G \trianglelefteq H$) is particularly simple. Suppose each granule in H contains the same number of granules of G. For example, every week contains seven days, or every business week contains five days. Furthermore, suppose this uniformity is *periodic*. If day a starts a new (business) week then every seventh day after day a also starts a new (business) week. This relation between granularities permits a simple conversion between G and H that we call a *regular mapping*. (If there is a regular mapping from G to H, with $G \trianglelefteq H$, then G *groups periodically into* H [5].)

Before describing the conditions for the existence of a regular mapping in detail, it may help to consider Fig. 8. This figure shows a portion of three granularities: D (days), W (weeks), and B (business weeks). The granule $D(i)$ (the ith granule in D) is contained in the granule $W(\lfloor\frac{i-22}{7}\rfloor)$. In the granularity B, $D(i)$ is contained in $B(\lfloor\frac{i-30}{7}\rfloor)$ if $(i-30) \bmod 7 < 5$, otherwise it is *invalid*. In both cases, the conversion is accomplished by a subtraction and an integer division that rounds down.

In converting from coarser to finer, $W(i)$ equals the instant

$$D(7i+22) \sim D(7i+28),$$

while $B(i)$ equals $D(7i+30) \sim D(7i+34)$. Again, these are simple functions involving a single addition and multiplication.

The key to these conversions is knowing the *period size of* H in G (seven for weeks in days and business weeks in days), the *group size of* H in G (seven for weeks in days and five for business weeks in days), and the *anchor of* H in G (the index of the granule in G containing the first instant of the origin in H, 22 for weeks in days and 30 for business weeks in days).

For granularities G and H, if there exist integers p (period size of H in G), s (group size of H in G), and a (anchor of H in G) such that for all $i \in \mathrm{domain}(H)$,

$$H(i) = \bigcup_{j=p\cdot i+a}^{p\cdot i+a+s-1} G(j),$$

then there exists a *regular mapping* between G and H. Note that the definition implies $G \trianglelefteq H$. The actual mappings between G and H are then:

$$\begin{aligned} scale(i_G, H) &= cast(i_G, H) \\ &= \begin{cases} \lfloor \frac{i-a}{p} \rfloor & : \text{ if } (i-a) \bmod p < s \\ invalid & : \quad \text{otherwise,} \end{cases} \end{aligned}$$

$$\begin{aligned} cast(l_G \sim u_G, H) &= cast(l_G, H) \sim cast(u_G, H) \\ scale(l_G \sim u_G, H) &= scale(l_G, H) \sim scale(u_G, H) \end{aligned}$$

$$\begin{aligned} cast(i_H, G) &= p \cdot i + a \\ scale(i_H, G) &= p \cdot i + a \sim p \cdot i + a + s - 1 \\ &= cast(i_H, G) \sim cast(i_H, G) + s - 1 \\ cast(l_H \sim u_H, G) &= cast(l_H, G) \sim cast(u_H, G) \\ scale(l_H \sim u_H, G) &= cast(l_H, G) \sim cast(u_H, G) + s - 1. \end{aligned}$$

As described, regular mappings require G and H to share the same extent. A slight generalization of these definitions to account for granularities with different extents is possible. Essentially, $invalid$ must be returned if the input granule is not in $extent(G) \cap extent(H)$.

8.2 Computing the Anchor

A regular mapping between G and H ($G \trianglelefteq H$) requires the three parameters p, s, and a. These parameters are specific to the pair $\{G, H\}$. Hence, the calendar specification includes the p, s, and a parameters for all pairs of granularities connected by a regular mapping. However, it turns out that it is possible to derive the a parameter (the anchor of G in H) from the origin of each granularity. This origin is specified by providing the index of a granule in another granularity that starts at the same time as the origin. Allowing granularities to have individual origins, as opposed to requiring all granularities to share the same origin, also permits a smaller encoding, as was discussed in Section 7. Typically, the origin of the bottom granularity in a calendar is implicit. For example, the user may specify the origin of the week's granularity in terms of hours. To convert between days and weeks requires the anchor of weeks in days. So, the origin of weeks (an index in the granularity of hours) must be converted to days, and so on. This computation of the anchor for each pair of granularities is not guaranteed to terminate, even for simple granularity graphs. Consider the graph in Fig. 9. We wish to cast an instant in hours to the granularity of minutes. For this, a regular mapping, we need the anchor of hours in minutes. To compute that, we need the anchor of days in minutes. For that, we need the anchor of days in hours, and the anchor of hours in minutes, which is what we started attempting to determine.

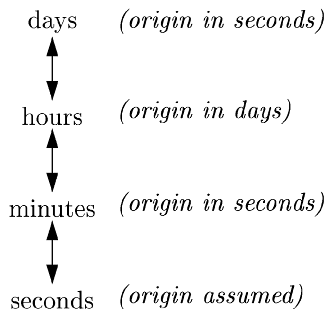


Fig. 9. A problematic granularity graph.

The question then is: What constraint(s) must be placed on the origin specification to ensure that the anchor can be computed for every regular mapping? It turns out that specifying the origin in any strictly finer granularity is sufficient.

Theorem 8.1. *If each origin of a granularity H is specified in a strictly finer granularity F ($F \triangleleft H$), then the calculation of the anchor for any pair of granularities will terminate.*

Proof. To convert between G and H ($G \trianglelefteq H$) requires the anchor of H in G. If, instead, the user supplied the anchor of H in F, then the supplied anchor (in granularity F) must be converted to G. This new conversion involves a pair of granularities $\{F, G\}$ whose "coarseness" is less than the coarseness of the original pair $\{G, H\}$ (since $F \triangleleft H$ and $G \trianglelefteq H$). Thus, the number of conversions involved in obtaining the anchor of H in G is finite. □

The conversions that are performed to calculate the anchor of H in G are simple $cast$ operations. Note that each conversion involves a mapping along the finer-than path from H through G to chronons. While the anchor of G in H is relevant only for regular mappings, it is still possible for the computation of that parameter to utilize irregular mappings.

8.3 Adding Mapping Costs to the Granularity Graph

There are usually many correct conversion paths in the granularity graph between any pair of granularities. To help determine the most *efficient* correct path, each edge in the granularity graph is annotated with the *cost* of the mapping. An initial approach, which requires no effort on the part of the specifier, is to associate a weight of one to regular mappings and a large weight to irregular mappings since applying an irregular mapping requires at least an expensive function call and return. A more refined weighting scheme might use timings from sample runs of the functions to determine their relative execution time.

8.4 Determining an Efficient Path

We present two algorithms for determining efficient V-paths. The first algorithm takes as input the finer-than relation $\trianglelefteq$, the weighted granularity graph $\mathcal{G}$, and two granularities G and H. It outputs a shortest V-path from G

```
Distinctly number the granularities from 1 to $V$ such that if granularity number $a$ is strictly finer
than granularity number $b$ then $a < b$. (Topological sort on the $\trianglelefteq$ relation.)
let $Icost(a, b) = \infty$ for $1 \leq a < b \leq V$
let $Icost(a, a) = 0$ for $1 \leq a \leq V$
for $d = 1$ to $V - 1$
    for $b = 1$ to $V - d$
        $a = b + d$;
        $Icost(a, b) = \min_{x \in F(a)} w(a, x) + Icost(x, b)$
for $s = 2$ to $2V$
    for $b = 1$ to $V$
        $a = s - b$;
        if $1 \leq a \leq V$
            $Vcost(a, b) = \min\left\{Icost(a, b), Icost(b, a), \min_{x \in F(a)} w(a, x) + Vcost(x, b)\right\}$
```

Fig. 10. Algorithm to calculate $Icost$ and $Vcost$.

to H. The second algorithm takes as input the finer-than relation $\trianglelefteq$ and the weighted granularity graph $\mathcal{G}$. It outputs shortest V-paths in $\mathcal{G}$ for all pairs of granularities. As we will see, the all-paths algorithm is significantly slower than the single-path algorithm which must nonetheless be called repeatedly if mappings between many pairs of granularities are desired.

These algorithms return only V-paths. An independent procedure checks for a direct mapping from G to H in the cases when G and H are incomparable.

8.4.1 Single-Pair Shortest V-Path

The single-pair shortest V-path algorithm performs two single-source shortest path computations, one from G in the graph $\mathcal{G}_{\trianglerighteq}$ and one from H in the graph $\mathcal{G}^R_{\trianglelefteq}$. The graphs, $\mathcal{G}_{\trianglerighteq}$ and $\mathcal{G}^R_{\trianglelefteq}$, have the same vertex set as $\mathcal{G}$, but the edges of $\mathcal{G}_{\trianglerighteq}$ are $\{(X, Y) \mid (X, Y) \in \mathcal{G} \text{ and } X \trianglerighteq Y\}$, while the edges of $\mathcal{G}^R_{\trianglelefteq}$ are $\{(Y, X) \mid (X, Y) \in \mathcal{G} \text{ and } X \trianglelefteq Y\}$ (here, 'R' denotes "reversed").

A shortest V-path from G to H is the concatenation of a shortest path from G to X in $\mathcal{G}_{\trianglerighteq}$ with the reversal of a shortest path from H to X in $\mathcal{G}^R_{\trianglelefteq}$. The definitions of $\mathcal{G}_{\trianglerighteq}$ and $\mathcal{G}^R_{\trianglelefteq}$ make any such concatenation a V-path in $\mathcal{G}$. Choosing the X that minimizes the length of this concatenated path gives the shortest V-path from G to H.

The running time of this algorithm is twice that of the single-source shortest path algorithm plus $O(V)$ to minimize over X where V is the number of vertices in $\mathcal{G}$. We may assume that $\mathcal{G}_{\trianglerighteq}$ and $\mathcal{G}^R_{\trianglelefteq}$ are acyclic; the only cycles in these graphs involve granularities that are congruent and may be treated as a single granularity for the purposes of these shortest path calculations. Note that $\mathcal{G}$ may have cycles of nonisomorphic granularities, but $\mathcal{G}_{\trianglerighteq}$ and $\mathcal{G}^R_{\trianglelefteq}$ do not. A single-source shortest path computation in a directed acyclic graph can be done in time $O(V + E)$ where E is the number of edges in $\mathcal{G}$. Thus, the total running time of the single-pair shortest V-path algorithm is $O(V + E)$.

8.4.2 All-Pairs Shortest V-Path

The all-pairs shortest V-path calculation uses a dynamic programming approach. Distinctly number the granularities from 1 to V where V is the total number of granularities in $\mathcal{G}$. The numbering should have the property that, if granularity number a is strictly finer than granularity number b, then $a < b$. We will refer to granularities by number for the remainder of this section.

Let $F(a) = \{x \mid (a, x) \in \mathcal{G} \text{ and } x \leq a\}$ be the set of granularities adjacent from a and finer than a. Let $w(a, b)$ be the cost of the edge from a to b in $\mathcal{G}$, that is, the relative time to convert a granule in a to a granule in b (or vice versa). Define the cost of a direct path from a to b as

$$Icost(a, b) = \begin{cases} \infty & \text{if } a < b \\ \min_{x \in F(a)} w(a, x) + Icost(x, b) & \text{otherwise,} \end{cases}$$

and the cost of a V-path from a to b as

$$Vcost(a, b) = \min\left\{Icost(a, b), Icost(b, a), \min_{x \in F(a)} w(a, x) + Vcost(x, b)\right\}.$$

The algorithm to calculate the two tables $Icost$ and $Vcost$ is shown in Fig. 10.

Notice, in the case of the $Icost$ table, we start with $Icost(a, a) = 0$ on the main diagonal, then fill in $Icost(a, b)$ where $a - b = 1$, then fill in $Icost(a, b)$ where $a - b = 2$, etc. To calculate $Icost(a, b)$, where $a - b = d$, we only need $Icost(x, b)$ for $x < a$. Since $x < a$, $x - b < d$ and, because of the order in which we fill the table, $Icost(x, b)$ has already been calculated (if $x - b$ is negative, then $Icost(x, b) = \infty$). This implies that $Icost(a, b)$ is well-defined.

In the case of the $Vcost$ table, we cannot follow the same order. Rather, we fill in the $Vcost$ table in the order of the increasing *sum* of a and b. We know $Vcost(1, 1) = 0$. To calculate $Vcost(a, b)$ where $a + b = s$, we only need $Vcost(x, b)$ where $x + b < s$ which has already been calculated. This implies that $Vcost(a, b)$ is well-defined.

The cost to fill in the tables is $O(V^2)$ plus the time to do the two minimizations. Each edge (a, x) appears at most V times in each minimization since a particular granularity a appears at most V times within each loop. Thus, the running time is $O(V^2 + VE) = O(VE)$ where E is the number of edges in $\mathcal{G}$ and is at least $V - 1$.

8.4.3 Choosing the Method

To compare these algorithms, we implemented both, as well as some variants, and used the ATOM tool [28] to measure the number of processor cycles spent in computing an optimal path [19]. We ran a series of tests on a multi-calendar granularity graph similar to that shown in Fig. 7, with 18 granularities and 20 edges, over randomly selected pairs of granularities. Each test was repeated 50 times.

As might be expected, the single-pair shortest V-path algorithm, at 16,684 cycles, was faster by almost a factor of ten, than the all-pairs dynamic programming approach, at 156,555 cycles. However, at current processor speeds, even the slower algorithm is quite practical: On a DEC 2000/233 workstation (with an Alpha 21064 processor), the slower algorithm, on this graph, takes less than a millisecond to compute all paths.

The algorithm was shown above to be quadratic in the number of granularities. In a database setting, this is not an issue even if the granularities number in the hundreds. Path selection is done during query *analysis*, not query *evaluation*. And, the all-pairs algorithm can be run when the database is configured by the DBA or each time a user (or DBA) defines a new granularity. A second or two to compute all paths will not be noticeable.

Where the path computation time may make a difference is when multiple granularities are used in an application. Adding this computation to each invocation of the application is less desirable. In that situation, the dynamic programming approach can be modified to do just enough searching to find the best path between two specified granularities with future requests using the intermediate paths computed as a side-effect [19]. The first call requires about a third of the time of the all-pairs computation with subsequent calls taking even less. If only a few granularities are used by the program, this approach is best; if many granularities are used, there is little difference.

8.5 Evaluating the Path

After the most efficient conversion path has been determined, it must be evaluated. To evaluate a cast, the V-path is traversed, applying each mapping in turn. Regular mappings utilize the p, s and a parameters as discussed in Section 8.1; irregular mappings invoke user-supplied functions. Integers (granule indexes) are passed to and returned by these functions. The conversion completes when $invalid$ is returned by a mapping, or when the final mapping returns a result.

To evaluate a cast on an indeterminate value, the cast is applied to both supports. If the lower and upper supports of the result are identical, then the result is determinate.

Section 8.1 also showed how to evaluate a scale over a regular mapping in terms of the p, s and a parameters; this turns out to require one or more casts and some arithmetic. In the general case, a scale over an irregular mapping on an indeterminate value can be implemented with two scales on determinate values:

$$scale(l \sim u, G) = l^l \sim u^u,$$

where $l^l \sim l^u = scale(l, G)$ and $u^l \sim u^u = scale(u, G)$. It may be preferable, in terms of efficiency, to supply two scale functions for each irregular mapping, one taking a determinate value and the other, an indeterminate value.

8.6 Query Optimization

The impact of granularity on the optimization phase of an SQL compiler can be observed in the running example query. In that query, the WHERE clause has an overlap between a flight departure instant, given in minutes, and a vacation period, given in days. In Section 6.4, it was shown that to perform the overlap using coarser operand semantics the flight departure time is first scaled from minutes to days and, only then, compared with the vacation time. But this is not the only way to effect coarser operand semantics. An alternative, semantically-equivalent query is given below.

```
SELECT *
FROM Vacations, Flight_Departures
WHERE Vacation = 'Thanksgiving' AND
        Flight_Departures.At_Time OVERLAPS
         (CAST(Vacations.From_Time
         AS MINUTE),
          CAST((Vacations.To_Time + INTERVAL
           '1' DAY) AS MINUTE) - INTERVAL
           '1' MINUTE);
```

This alternative might be preferred if an index existed for the flight departure times but not for the vacation times. Other semantically-equivalent alternatives exist, such as converting both operands to a granularity that does not appear at all in the original query, as illustrated below.

```
SELECT *
FROM Vacations, Flight_Departures
WHERE Vacation = 'Thanksgiving' AND
        CAST(Flight_Departures.At_Time
           AS SECOND) OVERLAPS
         (CAST(Vacations.From_Time
             AS SECOND),
          CAST((Vacations.To_Time + INTERVAL
               '1' DAY) AS SECOND) - INTERVAL
               '1' SECOND);
```

This alternative might be preferred if the query were nested inside a larger SELECT and, further, extensive comparisons were to be made of both times at the granularity of seconds.

8.7 Global Optimizations

The goal of global query optimization is to determine which semantically-equivalent, alternative, query execution plan is the cheapest to evaluate. In general, there are a large number of such alternatives. For instance, a query involving n OVERLAP operations in a system that supports m granularities has $O(m^n)$ alternatives. However, some simple heuristics can substantially reduce the size of the search space.

First, times that are indexed, hashed, or clustered should not be converted. A query should take advantage of these performance-enhancing data structures. If such times were converted, it may well be the case that the data structure becomes less effective. Second, only alternatives that involve conversions to granularities actually present in the original query should be explored. So, for instance, converting flight departure and vacation times to Chinese lunar months would be unlikely to lead to an optimized query. Finally, all else being equal, a scale or cast from a coarser to a finer granularity is preferred over the opposite conversion. This "local" optimization technique is presented in detail below.

8.8 Local Optimizations

Local optimizations can be applied to improve the speed of a single conversion between a pair of granularities.

The conversion operations are performed in the "inner-loop" of query processing, potentially done many times during a query. Each regular mapping costs (possibly) one addition and one "expensive" suboperation. The expensive suboperation is a division for a regular mapping from a finer to a coarser granularity. On some machines (e.g., Sun-4s) division is microcoded as repeated subtraction, typically costing much more than addition or multiplication. For a regular mapping from a coarser to a finer granularity, the expensive suboperation is a multiplication. For an irregular mapping, it is a C function invocation, which probably uses (at least) a division or a multiplication. In this section, we present four optimization strategies that are designed to minimize the cost of the "expensive suboperation." The important point here is that, by incorporating the operations in the DBMS, it can make these optimizations. In the previous approach, such optimizations are not possible because they are in application code.

The first optimization is an *algebraic simplification* of composed casts. For certain compositions (e.g., of regular mappings with equivalent origins, such as scaling from minutes to days), an expensive suboperation (a division or multiplication) can be eliminated. For example, to cast from minutes to days, we can algebraically simplify $(t\ \mathbf{div}\ 60)\ \mathbf{div}\ 24$ to $t\ \mathbf{div}\ 1440$.

The second optimization exploits the fact that on many machines multiplication is much cheaper than division. This optimization applies only to temporal comparisons, but we anticipate that comparisons will be the most common kind of temporal operation. All comparisons, including OVERLAPS, are expressed as formulas involving the *Before* relation (the $<$ relation on integers) and logical connectives [2]. Consider evaluating g *Before* h where $g \in G$ and $h \in H$ are determinate instants. If $G \trianglelefteq H$, then g *Before* h is equivalent to $scale(g, H)$ *Before* h and is also equivalent to g *Before* $scale(h, G)$. So, a temporal comparison with a scale on one operand can be transformed into a comparison with a scale on the other operand. This *program transformation* trades a scale from a finer to a coarser granularity (a division) for a scale from a coarser to a finer granularity (a multiplication). The query processor can choose the cheaper operation (in this case, the coarser to finer operation), but must factor into the decision how many times the operation is executed. If g and h are column variables and there are far fewer distinct values in g's column, then the transformation will not improve performance since many more scales of h will be performed than scales of g.

A third optimization is to introduce a *direct link* into the granularity graph for a highly optimized mapping function. For example, if the database implementor knows that casting years to seconds will be a common operation, a direct link with the name of the optimized mapping function can be inserting into the granularity graph during its construction. In casting years to seconds, the run-time engine can use this direct link rather than the composition of years to months, months to days, days to hours, hours to minutes, and minutes to seconds which costs three regular and two irregular mappings in total. Elsewhere, we show that casting years to seconds, even in the presence of leap days, requires only eight microseconds on a Sun-4 IPC [9].

The final optimization is to use a *lazy caching* strategy to avoid recomputing previously cast times. The caching strategy is based on the observation that times in a column are often clustered rather than distributed uniformly over the entire time-line (random sampling could be used to detect the clustering). Consequently, there are probably many cases where several instants at the finer granularity cast to the same instant at the coarser granularity. For example, instants in a column of employee birth dates will be clustered between 1938 and 1978 (most employees are between twenty and sixty years old). Consider a query in which these birth dates, stored to the granularity of days, are compared to a column at the granularity of years (e.g., in computing a bar graph of employee ages). In a large corporation, it is probably the case that several employees were born in the same year. To avoid recomputing the cast of years to days (introduced by a previously discussed optimization), we can cache previously computed casts using a small array. As another example, we saw in the previous section that scaling indeterminate values can be expressed as a pair of scales on determinate values, potentially increasing the cache hit probability. The viability of the caching strategy is a trade-off between the cost of building and maintaining the cache and the cost of cache misses.

To quantify the actual cost of supporting queries on mixed granularities, we programmed the example query, under a variety of optimization strategies, as a series of calls in the MULTICAL system [29]. The call sequences are shown in Fig. 11. The variables `f`, `v.from`, and `v.to` are the column variables for the `Flight_Departures.At_Time`, `Vacations.From_Time`, and `Vacations.To_Time`, respectively. The `unpack` operations parse the timestamp flags to distinguish determinate from indeterminate and special instants. We ignored the "Thanksgiving" selection and coded the OVERLAPS as a conjunction of *Before* operations with no short circuit evaluation. The first sequence (from left to right) is an overlap with no support for mixed granularities. Testing this sequence will give us a base cost against which we can compare the cost of modeling and using information at mixed granularities. The second sequence scales minutes to days, using the algebraic optimization. We also tested the unoptimized sequence; that code is not shown. The third sequence combines the algebraic simplification with the program transformation that trades a scale down to minutes for a cast up to days. Here, the overlap is manipulated into *Before* on the lower support of $scale(\texttt{v.from}, minutes)$ and on the upper support of $scale(\texttt{v.to}, minutes)$.

We compiled all four tests using the GNU C compiler, version 2.4.5, with compiler optimizations fully enabled. We then ran the tests several million times on a dedicated Sun-4 IPC (a twelve "mips" machine). The results we obtained are

```
{ The same-granularity sequence }
    unpack_event(f);
    unpack_period(v);
    c1 := Before(v.from,f);
    c2 := Before(f,v.to);
    if (c1 and c2) then
        include f,v in result
```

```
{ The algebraic optimization }
    unpack_event(f);
    f := f div 1440;
    unpack_period(v);
    c1 := Before(v.from,f);
    c2 := Before(f,v.to);
    if (c1 and c2) then
        include f,v in result
```

```
{ The program transformation }
    unpack_event(f);
    unpack_period(v);
    v.from := v.from × 1440;
    v.to := (v.to×1440)+1439;
    c1 := Before(v.from,f);
    c2 := Before(f,v.to);
    if (c1 and c2) then
        include f,v in result
```

Fig. 11. MULTICAL calls for example queries.

as follows: Each predicate evaluation took approximately 10 microseconds with no granularity conversion, 48 microseconds with one unoptimized scale, 27 microseconds with algebraic optimization, and 14 microseconds with algebraic optimization and program transformation. While these differences may not seem important, the microseconds quickly add up. If we assume that the *Vacations* and *Flight_Departures* relations have a modest number of tuples, 50 and 5,000, respectively, then the total cost of the OVERLAPS would be 2.5 seconds with no granularity conversion, 12 seconds with one unoptimized scale, 7 seconds with algebraic optimization, and only 3 seconds with algebraic optimization and program transformation.

These results show that modeling times at different granularities does carry a cost; for the example query it adds an overhead of between 40 percent and 380 percent. The results also show that the optimizations significantly improve performance. Note, however, that there are many other components to query evaluation, such as disk reads and writes; the additional cost of granularity conversions over the entire query execution will be relatively slight. Also, note that a user who does not want the extra modeling capability of mixed granularities can simply specify that all columns have an identical granularity, thereby incurring no added cost.

9 Related Work

Our work can be viewed as an extension of Anderson's pioneering research on a model of time [3]. Anderson pointed out the need to model times at multiple granularities. Clifford and Rao further developed Anderson's framework by adding a "granularity chain" (a complete ordering of granularities) and "finer" granularity conversions between times [7]. Wiederhold, Jajodia, and Litwin made Clifford and Rao's theoretical work more concrete by proposing a specific semantics for temporal comparisons at mixed granularities [33]. Their proposed semantics is similar to the finer granularity semantics mentioned in this paper. Recently, Wang, Jajodia, and Subrahmanian generalized the "granularity chain" to a lattice and proposed semantics for moving times "up" and "down" the lattice [32]. The specific requirements given here for granularities (total ordering of granules, ordering of granules consistent with ordering of their indexes, contiguity of granule index in a granularity, existence of an origin for each granularity, and existence of a bottom granularity) are similar to those specified for time units. Unlike time units though, our approach permits noncontiguous granules and negative granule indexes. Also, our granularities are not required to form a lattice under the finer-than relation or to share a common origin, thus permitting a more space efficient representation.

Wang et al. also proposed *temporal modules* and *extended temporal modules* that provide access to temporal relations via windowing functions, each in terms of a different time unit [31], [32]. As such, they consider how to map data defined over one granularity (e.g., annual salary) into data over another granularity (e.g., monthly salary). The present paper does not address data conversion, though our approach might well apply to granularity mappings performed during data conversion. Finally, their calculus-based federated query language allows comparisons between instants of different time units. However, only finer granularity semantics, at the granularity $\perp$, is employed, with explicit cast or scale operations not permitted.

The present paper elaborates on the theoretical framework of [5] by showing how values in particular time units (i.e., granularities) can be converted to other time units semiautomatically via user-provided conversion functions.

Barbic and Pernici discuss relative, absolute, periodic, and imprecise times at different Gregorian granularities for office information systems in the context of constraint triggers [4]. They recommend converting operands to the coarsest granularity during a temporal (comparison) operation to avoid creating information. Barbic and Pernici also advocate a "signed integer" timestamp format which is the gist of our format. Montanari et. al. investigated a slightly different problem, that of extending the granularity chain to cover macroevents, i.e., events with duration [23]. We only consider instantaneous events (instants) in this paper. They also propose finer and coarser granularity conversions (effectively a scale via a regular mapping only); the issue of indeterminacy in finer conversions is not explicitly addressed.

Terenziani's temporal formalism [30] is an adaptation of that proposed earlier [18] in which a granularity is defined as a set of intervals over the (discrete) set of Reference Time points (e.g., days). Leban et al., Terenziani, and [6] all include notations for deriving new granularities (sometimes confusingly called calendars) from other granularities (e.g., first Mondays in April). One could envision an extension to our approach whereby regular mappings could be specified in such terms, elaborating on the more restrictive sense adopted in the present paper.

None of the above papers address the integration of granularities from multiple calendars into SQL; they also lack indeterminacy, and impose a single semantics for comparison operations, typically finer granularity semantics. (To be fair, these papers weren't attempting to solve these particular problems; their foci were on other aspects of granularity.) In contrast, we treat all instants as indeterminate. When two instants located in the same hour are scaled to a finer granularity, two similar indeterminate instants result. But, each indeterminate instant retains the semantics of the original instant in that it records that the instant is located sometime during the hour (with the upper and lower supports expressed in the finer granularity). We also support several different semantics for every kind of temporal operation (not just comparison operations). It is our position that indeterminacy is necessary to support finer granularity conversions and to correctly model instants. Further, an important difference between our work and all of the above presentations is that we focus also on practical issues. We are interested in engineering a database to support mixed granularities and so we designed mechanisms to effect this support, attempting to simplify as much as possible the task of the calendar specifiers.

The practical focus of this paper on implementation is shared by Lorentzos who advocated a scheme for storing and querying nonmetric data types [20]. The SQL-92 timestamp format is one example of a nonmetric data type; it has separate fields for years, months, days, hours, minutes, and seconds. Lorentzos allows only coarser conversions in a granularity "chain." These conversions consist of removing various fields, e.g., scaling from months to years removes the months field. Although the granularity conversion operation (between fields that are in the nonmetric data type) is fast, we previously empirically determined that the execution cost of other more common temporal operations severely increases, as does the space cost [10]. Hence, we advocate that timestamp formats have as few separate internal fields as possible.

Goralwalla et al. also used a tuple of integers to denote an instant [13], [14]. This provides an increase in expressive power for intervals (for example, "one month and five days" must be mapped to a number of days (from 33 to 36) or an indeterminate interval $33 \sim 36$ in our model), but not for instants or periods. They also support calendars, in a similar way as defined here, but restrict calendars to contain granularities for which a total order is defined. They support regular mappings and obtain some of the generality of irregular mappings through a variety of calendric functions. These are especially effective for converting intervals to different granularities though at an increase in complexity for the calendar specifier. And, as with the other approaches, conversion of instants are effected by converting to the finest granularity which they term "global real time," which is a dense model represented with floating point numbers. In contrast, in our model, we go only as far down the hierarchy as is needed and always deal with integers. Finally, these papers also support temporal indeterminacy, but via a set of times rather than a lower and an upper support; they do not support an associated probability function. We argue that indeterminacy and granularity are intimately related, serving as different perspectives of a single phenomenon.

Finally, Gauthier has advice on how to implement calendars in Ada, taking into account important details such as very precise compile-time type checking [12].

10 Summary

This paper demonstrates that granularity and indeterminacy are related features of temporal data. Granularity is the unit of measure for a temporal datum while indeterminacy represents partial information about finer units of measure. For example, an instant known to the granularity of an hour has an hour-long support. For this instant, we only know the hour during which it is located, we cannot ascertain with certainty the minute during which it is located. Such is the nature of "real-world" temporal data.

In this paper, we use a common model of a granularity as a segmentation of the time-line. Granularities are related in that some granularities are finer or coarser with respect to others. The conversion functions, *scale* and *cast*, move times between granularities. The scale operation does not create information; rather it exploits the relationship between granularity and indeterminacy to refine the information content of a temporal value. A determinate instant stored at a particular granularity becomes indeterminate when scaled to a finer granularity. Support for indeterminacy permits conversions between granularities which some have considered incomparable, such as weeks and months. Judicious use of scale and cast can implement a variety of semantics for temporal operations. We examine various semantics and show that they can be effected by inserting cast or scale operations.

The conventional way to specify a granularity is to provide an invertible function that maps an index to a granule, which is a subset of the time domain. We propose that granularities instead be specified via regular or irregular mappings between granularities. Regular mappings are specified with three parameters; irregular mappings are associated with arbitrary functions which may be invoked by the DBMS. This specification is easier to define than the original approach: most of our mappings are regular, whereas most index-to-granule functions are complex. Efficiency is also gained in that a conversion from one granularity to another need only go through a common, finer granularity rather than all the way down to the time domain and back up. Our approach also supports modular specification of granularities, via calendars which collect together a set of granularities and their associated mappings.

Finally, we explore the cost of our framework. We present four optimizations that can be easily applied during semantic analysis and show that for the example query, these optimizations reduce the predicate evaluation overhead to a reasonable level.

Our conclusion is that full database support for temporal granularities is not only a desirable goal, but, by using a realistic design that addresses theoretical concerns, language extensions, and implementation details, is an attainable one.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation grants ISI-9302244, IRI-9632569, and IIS-9817798, by IBM contract 1124, and by a grant from the AT&T Foundation. The authors would like to thank John Hartman and Udi Manber for their comments and Marshall Freiman for initial development of the granularity conversion code.

REFERENCES

[1] M. Adiba, N. Bui Quang, and J. Palazzo de Oliveira, "Time Concept in Generalized Data Bases," *Proc. ACM Ann. Conf.,* pp. 214–223, Oct. 1985.

[2] J.F. Allen, "Maintaining Knowledge about Temporal Intervals," *Comm. ACM,* vol. 26, no. 11, pp. 832–843, Nov. 1983.

[3] T.L. Anderson, "Modeling Time at the Conceptual Level," *Proc. Int'l Conf. Databases: Improving Usability and Responsiveness,* P. Scheuermann, ed., Jerusalem, Israel: Academic Press pp. 273–297, June 1982.

[4] F. Barbic and B. Pernici, "Time Modeling in Office Information Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* S. Navathe, ed., pp. 51–62, May 1995.

[5] C. Bettini, C.E. Dyreson, W.S. Evans, R.T. Snodgrass, and X.S. Wang, "A Glossary of Time Granularity Concepts," *Temporal Databases: Research and Practice,* O. Etzion, S. Jajodia and S. Sripada, eds., Springer-Verlag, 1998.

[6] R. Chandra, A. Segev, and M. Stonebraker, "Implementing Calendars and Temporal Rules in Next Generation Databases," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 264–273, 1994.

[7] J. Clifford and A. Rao, "A Simple, General Structure for Temporal Domains," *Proc. Conf. Temporal Aspects in Information Systems (AFCET),* pp. 23–30, May 1987.

[8] N. Dershowitz and E.M. Reingold, *Calendrical Calculations.* Cambridge Univ. Press, 1997.

[9] C.E. Dyreson and R.T. Snodgrass, "Timestamp Semantics and Representation," *Information Systems,* vol. 18, no. 3, pp. 143–166, 1993.

[10] C.E. Dyreson and R.T. Snodgrass, "Efficient Timestamp Input/Output," *Software–Practice and Experience,* vol. 24, no. 1, pp. 80–109, 1994.

[11] C.E. Dyreson and R.T. Snodgrass, "Supporting Valid-time Indeterminacy," *ACM Trans. Database Systems,* vol. 23, no. 1, Mar. 1998.

[12] M. Gauthier, "The Avatars of a Package for Calendars in Ada," *Software–Practice and Experience,* vol. 25, no. 4, pp. 403–427, Apr. 1995.

[13] I.A. Goralwalla, Y. Leontiev, M.T. Özsu, and D. Szafron, *Modeling Time: Back to Basics,* Technical Report TR 96-03, Dept. Computer Science, Univ. of Alberta, Feb. 1996.

[14] I.A. Goralwalla, Y. Leontiev, M.T. Özsu, and D. Szafron, "Modeling Temporal Primitives: Back to Basics," *Proc. Int'l Conf. Information and Knowledge Management (CIKM),* pp. 24–31, 1997.

[15] C.S. Jensen, C.E. Dyreson, M. Böhlen, J. Clifford, R. Elmasri, S.K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M.R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A. Tansel, R. Tiberio, and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts," *Temporal Databases: Research and Practice,* O. Etzion, S. Jajodia, and S. Sripada, eds.,Springer-Verlag, pp. 39, 1998.

[16] W.H. Inmon, *Building the Data Warehouse,* second ed., John Wiley, 1996.

[17] N. Kline, J. Li, and R.T. Snodgrass, "Specifying Multiple Calendars, Calendric System, and Field Tables and Functions in TimeADT," TIMECENTER Technical Report 41, May 1999.

[18] B. Leban, D.D. McDonald, and D.R. Forster, "A Representation for Collections of Temporal Intervals," *Proc. Nat'l Conf. Artificial Intelligence,* pp. 360-366, Aug. 1986.

[19] H. Lin, "Efficient Conversion Between Temporal Granularities," Master's thesis, Dept. Computer Science, Univ. of Arizona, TIMECENTER Technical Report TR-19, June 1997.

[20] N. Lorentzos, "DBMS Support for Nonmetric Measuring Systems," *IEEE Trans. Knowledge and Data Eng.,* 1992.

[21] J. Melton, ed. *Database Language—SQL, ANSI X3.135,* 1992.

[22] J. Melton and A.R. Simon, *Understanding the New SQL: A Complete Guide.* San Mateo, Ca.: Morgan Kaufmann, 1993.

[23] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto, "Dealing with Time Granularity in the Event Calculus," *Proc. Int'l Conf. Fifth Generation Computer Systems (ICOT),* pp. 702–712, June 1992.

[24] M. Niezette and J. Stevenne, "An Efficient Symbolic Representation of Periodic Time," *Proc. First Int'l Conf. Information and Knowledge Management (CIKM),* Nov. 1992.

[25] N. Sarda, "HSQL: A Historical Query Language," *Temporal Databases: Theory, Design, and Implementation.* Chap. 5, Benjamin/Cummings, pp. 110–140, 1993.

[26] R.T. Snodgrass, I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Kafer, N. Kline, K. Kulkanri, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, and S.M. Sripada, *The TSQL2 Temporal Query Language.* Kluwer Academic, 1995.

[27] R.T. Snodgrass, M.H. Böhlen, C.S. Jensen, and A. Steiner, "Adding Valid Time to SQL/Temporal," Change proposal, ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/ WG3 DBL MAD-146r2, Nov. 1996.

[28] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 196–205, June 1994.

[29] M.D. Soo, R.T. Snodgrass, C.E. Dyreson, C.S. Jensen, and N. Kline, "Architectural Extensions to Support Multiple Calendars," TempIS Technical Report 32, Computer Science Dept., Univ. of Arizona, revised May 1992.

[30] P. Terenziani, "Integrating Calendar Dates and Qualitative Temporal Constraints in the Treatment of Periodic Events." *IEEE Trans. Knowledge and Data Eng.,* vol. 9, no. 5, pp. 763–783, Sept./Oct. 1997.

[31] X. Wang, "Algebraic Query Languages on Temporal Databases with Multiple Time Granularities," *Proc. Int'l Conf. Information and Knowledge Management (CIKM),* 1995.

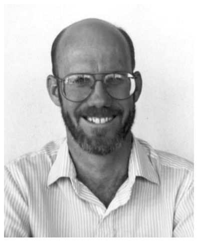
[32] X. Wang, S. Jajodia, and V. Subrahmanian, "Temporal Modules: An Approach Toward Temporal Databases," *Information Sciences,* vol. 82, no. 1/2, pp. 103–128, Jan. 1995.

[33] G. Wiederhold, S. Jajodia, and W. Litwin, "Dealing with Granularity of Time in Temporal Databases," *Proc. Third Nordic Conf. Advanced Information Systems Eng.,* May 1991.

Curtis E. Dyreson received the PhD degree from the University of Arizona, in 1994. He has been working in the field of temporal databases. After several years of toiling at James Cook University in tropical Queensland, he traveled to Aalborg University in northern Denmark to continue researching temporal databases.

William S. Evans received the BSc degree in computer science from Yale University in 1987 and the PhD degree in computer science from the University of California at Berkley in 1994. He then spent two years as a NSERC Canada International Postdoctoral Fellow at the University of British Columbia. He is now on the faculty at the University of Arizona. He is a member of the IEEE.

Hong Lin received the MS degree in physics from Northwestern University in 1993, and the MS degree in computer science from the University of Arizona in 1995. She is a software engineer at IBM Global Services.



Richard T. Snodgrass received his PhD degree from Carnegie Mellon University in 1982 and joined the University of Arizona in 1989, where he is a professor of computer science.

He is chair of ACM SIGMOD. He is a fellow of the ACM and a senior member of the IEEE. He is an associate editor of the *ACM Transactions on Database Systems* and is on the editorial board of the *International Journal of Very Large Databases*. He chaired the program committees for the 1994 ACM SIGMOD Conference and the 1993 International Workshop on an Infrastructure for Temporal Databases. He also was a vice-chair of the program committees for the 1993 and 1994 International Conferences on Data Engineering and will chair the American program committee for the 2001 International Conference on Very Large Databases.

He chaired the TSQL2 Language Design Committee, edited the book, *The TSQL2 Temporal Query Langauage*, published by Kluwer Academic Press, and is now working closely with the ISO SQL3 Committee to add temporal support to that language. He initiated the SQL/Temporal part of the SQL3 draft standard. He is a coauthor of *Advanced Database Systems*, published by Morgan Kaufmann, a coeditor of *Temporal Databases: Theory, Design, and Implementation* published by Benjamin/Cummings and author of *Developing Time-Oriented Database Applications in SQL*, published by Morgan Kaufmann. He codirects TIMECENTER, an international center for the support of temporal database applications on traditional and emerging DBMS technologies.

His research interests include temporal databases, query language design, query optimization and evaluation, storage structures, database design, and software development databases.