EFFICIENT CONVERSION BETWEEN TEMPORAL GRANULARITIES

HONG LIN

July 8, 1997

TR-19

A TimeCenter Technical Report

Title	EFFICIENT CONVERSION BETWEEN TEMPORAL GRANULARITIES
	Copyright © 1997 HONG LIN. All rights reserved.
Author(s)	HONG LIN
Publication History	None

EFFICIENT CONVERSION BETWEEN TEMPORAL GRANULARITIES

HONG LIN, M.S.
The University of Arizona, 1997

Director:

A *temporal granularity* is a unit of measuring time, e.g., second, day, week. A *granularity graph* is a directed graph showing the relationship among the granularities. Efficiently and correctly converting time values within the granularity graph is critical for supporting multiple time granularities in an application program or a database management system. The research involves finding an optimally efficient path in the granularity graph for any pair of granularities and developing an algorithm to perform the conversion operation between the two granularities for anchored time related values, to correctly convert a granule from a specified granularity to another granularity. The research also evaluates several strategies to improve the performance of temporal operations at mixed granularities.

STATEMENT BY AUTHOR

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor and mentor, Professor Richard Snodgrass, for his persistent inspiration, guidance and support. Without his encouragement and endless patience, I could not have written this thesis. I wish to thank Professor William Evans and Professor John Hartman for their valuable comments and advice, and Professor Curtis Dyreson and Kristian Torp for their constant help.

Special thanks go to my parents for their continuing encouragement and support, and to my husband, Xiaoguang, and son, Daniel, for their understanding, sacrifice and patience.

Contents

List of Figures

Abstract

A *temporal granularity* is a unit of measuring time, e.g., second, day, week. A *granularity graph* is a directed graph showing the relationship among the granularities. Efficiently and correctly converting time values within the granularity graph is critical for supporting multiple time granularities in an application program or a database management system. The research involves finding an optimally efficient path in the granularity graph for any pair of granularities and developing an algorithm to perform the conversion operation between the two granularities for anchored time related values, to correctly convert a granule from a specified granularity to another granularity. The research also evaluates several strategies to improve the performance of temporal operations at mixed granularities.

Chapter 1

INTRODUCTION

Supporting multiple calendars in database applications is a highly desirable feature. Currently, the Gregorian calendar (with a fixed number of granularities: year, month, day, hour, and second) is the single calendar available in SQL-92 (Structured Query Language) for representing and manipulating time-related data. In the real world, there are many applications that require a wide variety of calendar support. The usage of a calendar depends on the cultural, legal and business aspects of the user. For example, the Eastern world commonly uses a lunar calendar, the US government uses a business calendar with the financial year starting in October, and universities generally use an academic calendar with years consisting of semesters or quarters. Today's database systems must support conversion among these calendars.

There has been considerable research in incorporating multiple calendars into a database system. But most of the previous research has focused on theoretical aspects. For mixed granularities in multiple calendars, no practical algorithm has been proposed that allows mappings to be composed automatically.

Based on the architecture first devised by Curtis Dyreson and Richard Snodgrass, with initial coding by Marshall Freiman, our work develops an efficient algorithm and provides a software module to support conversion operations for mixed granularities, i.e., converting a granule in one granularity to a granule in another granularity. The objective of this work is to provide an efficient and correct algorithm for supporting multiple time granularities within an application program or a database.

This thesis is organized as follows. Chapter Two gives an overview of the related work on the mixed granularities, including introducing the granularity graph and the initial model of converting time values from one granularity to another granularity. Chapter Three covers the granularity module interface and shows how to integrate different calendars into a single granularity graph. This chapter also points out what the interface needs to check to ensure the granularities declared by the user are properly defined. We examine the paths for granule conversion in Chapter Four. We then present two recursive algorithms for finding an optimal path and prove their correctness. We begin by giving an example to illustrate non-termination in conversion operation. We then propose a reasonable constraint to ensure termination. Finally, we present the algorithm for the conversion operation is presented. Chapter Six provides an empirical performance analysis, and examine the efficiency of path caching and the granularity origin offsets caching. We summarize our work in Chapter Seven. We include the external data structures and the detailed description of the functions provided by the external module in Appendix A. In addition, Appendix B gives the internal data structures.

Chapter 2

RELATED WORK

Since the very early days of computers, applications have had a need to represent times in stored data and to manipulate the information. But there is no standard; every computer system invented its own convention to handle time related data. This is clearly unacceptable. There have been several languages fully implemented to support the time related data available on commercial database management systems (DBMSs). The best known of these is SQL. SQL was first designed and implemented at IBM Corporation as the interface for an experimental relational database system called system R. SQL was first standardized in 1986 and was revised significantly to form the standard SQL-92 [Melton & Simon 1993]. SQL-92 includes date and time data types, and supports a single calendar, the Gregorian calendar. Recently, the Object Database Management Group defined the ODMG-93 standard for object database management to provide for object databases what SQL has provided for relational database [Cattell 1994]. But the time support in ODMG-93 is similar to SQL-92. Generally, the existing database software has ignored the issue of the mixed granularities or have assumed the use of a single calendar.

Anderson [Anderson 1982] first pointed out the need to support mixed granularities. Clifford and Rao [Clifford & Rao 1987] then proposed a theoretical model of complete ordering of granularities. Wiederhold, Jajodia and Litwin [Wiederhold et al. 1991] further developed this model by adding a specific semantics for temporal comparisons.

Temporal granularities, e.g., seconds, days, weeks, months, were initially formalized as partitions of some base time lines composed of indivisible time units, called *chronons* (usually denoted as $\perp$), e.g., microseconds. We slightly generalize Wang et. al.'s definition of time unit [Wang et al. 1993] to the following.

Definition 2.1 *A granularity α is a set of nonoverlapping and contiguous granules. Each granule has an integer index, with the ordering of integers. We use an integer subscripted with the granularity to identify the granule. The granularity contains 0_α termed anchor. Here,*

$$\alpha = \{\cdots, -1_\alpha, 0_\alpha, 1_\alpha, \cdots\}.$$

The granularity chronons ($\perp$) is the smallest granularity. Each granule at a granularity corresponds to an contiguous set of chronons. To distinguish between a granule (an integer) and the sequence of chronons that comprise a granule, we use **CHR** to represent the set of chronons in a given granule.

Definition 2.2 **CHR**$(i_\alpha) = \{c_\perp | c_\perp \ is \ in \ i_\alpha\}$ *, where i_α is the i^{th} granule in granularity α and $c_\perp$ is a chronon.*

The above definitions imply the following properties.

- for chronons $c_\perp$, $c'_\perp$ and granules i_α, j_α, $c_\perp \in$ **CHR**(i_α), $c'_\perp \in$ **CHR**(j_α) and $c_\perp < c'_\perp$ implies $i_\alpha \leq j_\alpha$.
- for chronons $c_\perp$, $c'_\perp$ and $c''_\perp$, $c_\perp < c'_\perp < c''_\perp$, $c_\perp \in$ **CHR**(i_α) and $c''_\perp \in$ **CHR**(i_α) implies $c'_\perp \in$ **CHR**(i_α) .
- For granules i_α and j_α , $i_\alpha \neq j_\alpha$ implies **CHR**$(i_\alpha) \cap$ **CHR**$(j_\alpha) = \emptyset$.

The first property says that chronons and granules are totally ordered. The second one requires that a granule contains a contiguous set of chronons and the third one ensures that the different granules do not overlap. We differ from Wang's definition in not requiring $0_\perp \in$ **CHR**(0_α) (Business calendar has an anchor in Gregorian date October 1, 1990), and in allowing gaps, i.e., some chronons may not map to any granule of a particular granularity, e.g., semesters with no summer coverage.

Cast, which converts a granule of one granularity to a granule of another granularity, is the basic operation on granule. Other operations (for example, *Scale* , *Plus*) generally can be defined in terms of the *Cast*. Following is the formal definition for the *Cast*.

Definition 2.3 *Cast*$(i_\alpha, \alpha, \beta) \rightarrow j_\beta$ *, where α and β are granularities, i_α is the i^{th} granule in α and j_β is the j^{th} granule in β.*

Given the definition of granularity, clearly, there is a "finer than" relation between granularities. For example, days is finer than months, months is finer than years, etc. Note that months is not a further partitioning of weeks, or vice-versa. *A complete lattice* is a partially ordered set in which every pair of elements have a unique least upper bound and a unique greatest lower bound [Vinogradov et al. 1988]. It has been shown that a collection of granularities (or time units) can form a complete lattice with respect to a "finer than" relationship [Wang et al. 1993]. By relating an arbitrary time unit (i.e., granularity) to the smallest time unit (i.e., chronons), the *Cast* operation is easily definable (the latter because the existence of a bottom in the lattice ensures that there is a path from a granularity to all other granularities, and the fact that the granule-chronon mapping is invertible). This model is sufficient from a mathematical point of view, but does not present a practical solution. First, most calendar users do not know what the smallest time unit it is; they usually build a new calendar based on a well-known calendar. Second, given leap seconds and various arbitrary aspects of human-designed calendars, the mapping functions from granularities to the smallest time unit are generally complex. Third, if a set of granularities do not form a complete lattice, one or more artificial granularities have to be added in order to form a complete lattice. These extra granularities "may sometimes be very hard to compute and counter-intuitive to real-life concepts of time units" [Wang et al. 1993]. Finally, this model has not considered the anchor difference between a pair of granularities. It assumes that all granularities have a same anchor. In reality, this assumption is impractical. At the physical level, time values are stored in fixed-size data structures called timestamps. For example, if we pick midnight, January 1, A.D. 1 as the anchor-point for all granularities, then representing a time value for Business day (starting on the Gregorian date October 1, 1990) would require many storage bits. Using different anchors can significantly reduce the storage requirement.

Although Wang's model provides a mathematical framework for mixed granularities, they do not present calendars. Recently, Kraus et al. propose a very interesting approach to represent time in a calendar [Kraus et al. 1996]. Most theoretical models including Wang's model reference time with respect to integers (granules), however, human being, as well as many applications specify time, not as integers, but as "dates" to a particular calendar (i.e., Monday, Tuesday, January in the Gregorian calendar). Referring to this mismatch between the theory and application, Kraus et al. provide a new definition of a calendar. They define a time unit as a time-value set with a linear order. For example, the time unit `month` consists of the 12 months of the year, i.e., January, February etc. In this model, they present time instances and time intervals in terms of constraints with respect to a given calendar. In case of multiple calendars, they also show how to integrate those calendars into a single, unified calendar. This framework offers the advantage that the user can work with his own calendar representation of time, which is more nature than representing time as integers. This work relates to ours because they introduce a new way to represent time and provide a new technique to integrate calendars. On the other hand, they do not discuss the conversions between the time units, nor do they cover the conversions among calendars.

Kraus et al. is not the only group trying to solve the mismatch of time representations between internal data structure and the external calendar specification. Dershowitz and Reingold [Dershowitz et al. 1990] provide Lisp functions for converting time points between the specifications in different calendars, namely the Gregorian, Julian, Islamic, Hebrew, and ISO (International organization for Standardization) calendars, and integers. Later, Reingold, Dershowitz and Clamen add the Mayan, French Revolutionary, and Old Hindu calendars [Reingold et al. 1993]. With this approach, a time point specified in one of the above calendar can easily be converted to an integer and an integer can also be converted back to a desired calendar representation. This approach also makes the internal conversion among the calendars possible.

TSQL2 (Temporal Structured Query Language) [Snodgrass 1995] which is a temporal extension to SQL-92 provides many capabilities not available in SQL-92. In particular, TSQL2 supports mixed granularities and multiple calendars. Although TSQL2 also chooses integers to represent time points, by taking Reingold's approach, it provides a nice Input/Output interface for converting between the internal form of a timestamp (i.e. integer), and various external forms, mainly character strings in a specific underlying calendars [Dyreson & Snodgrass 1994B]. With the I/O interface, each user can define his own calendar and deal with his calendar representation of time; at physical level, communication among those calendars is actually going on via the timestamp representation. TSQL2 supports conversion between a pair of granularities only if the user provides the direct mapping function for the conversions. As we stated before, most of the mapping functions are complex; in addition, in a large multicalendar system, it is impossible to provide all mapping functions for every pair of granularities. Based on the general architecture of the multicalendar system in TSQL2, our research tries to realize TSQL2 by providing an practical algorithm to allow the system to perform the conversions dynamically.

Dyreson and Snodgrass [Dyreson & Snodgrass 1994A] present a model for granularities in temporal operations which offers a practical solution to convert time values between a pair of granularities. They observe that the interactions between most granularities, e.g., `hours` and `minutes`, `days` and `weeks`, are regular: one is a further partitioning of the other, and so a granule represented by an integer can be converted to another by a simple multiply or divide, with an anchor adjustment. They defined the granularity graph explicitly, as mappings between granularities. Each node in the graph is a granularity, and each edge represents a relationship between a pair of granularities. An arrow from g to h indicates that g is finer granularity than h. The graph in Figure 2.1 [Dyreson & Snodgrass 1994A] shows a multicalendar granularity graph comprised of the Gregorian, Business, and Astronomy calendars. Mappings can be *regular mappings*, e.g., between `hours` and `minutes`, with a conversion constant, *irregular mappings* (granules can not be converted by a simple multiply or divide), e.g., between `months` and `days`, or *congruent mappings* (granularities with identical granules, but perhaps different anchors), e.g., between Gregorian `days` and Business `days`. In the granularity graph, a directed thin line is a regular mapping while a directed thick line is an irregular mapping. A congruent mapping is denoted by an undirected line labeled with a conversion constant 1. Irregular mappings are associated with two C functions provided by the granularity designers: one is for mapping "*upward*" (from finer to coarser granularity), the other is for mapping "*downward*" (from coarser to finer granularity). For a system supporting multiple calendars, there will be hundreds of granularities. However, without the immediate functional mappings from granules to chronons, the system must perform a *Cast* to do conversion among granularities. Dyreson and Snodgrass proposed a method to perform a *Cast* operation. The first step is to find a correct path, then execute each portion of the path applying the appropriate mapping. They conjecture that all *V-paths* down to a common

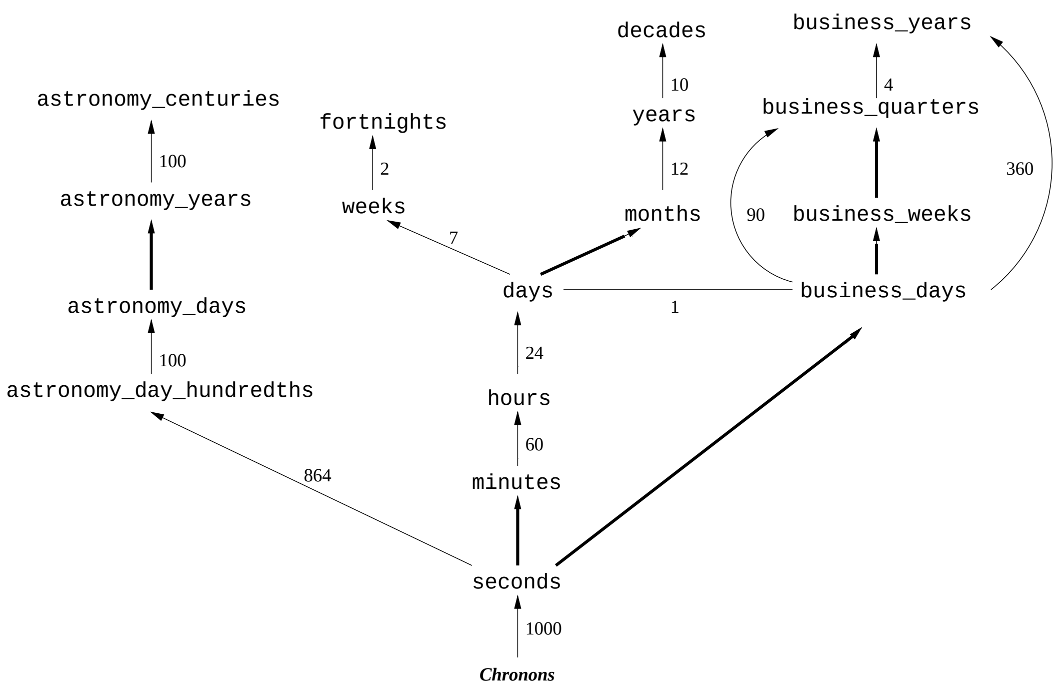


Figure 2.1: A multicalendar granularity graph

ancestor then back up (due to the shape, these paths are termed V-paths) yield equivalent results, but paths differ in computation cost, in terms of the number of user-defined functions that must be invoked.

The proposed work is to implement this model and provide an efficient and correct module to support both temporal DBMSs and application programs that handle anchored time values.

Chapter 3

THE GRANULARITY MODULE INTERFACE

This chapter summarizes the module interface provided to perform the conversion operations for mixed granularities. The SQL-92 standard only supports a single calendar, the Gregorian calendar. Our module remains consistent with SQL-92 and also provides support for multicalendar system.

In this research, our goal is efficient and correct conversions between temporal granularities. For example, to convert a time in Gregorian days to the same time in Chinese lunar days, the user is unlikely to provide functions to do the conversion; instead, the database must be able to convert Gregorian days to lunar days dynamically from the user-supplied relationships.

Calendars define granularities. We envision that the DBMS vendor will provide some common calendars (for example, Gregorian calendar), and the database user can define his own calendars (for example, a company's business calendar). Different calendars are woven together to form the granularity graph. The user can declare many granularities (consider fiscal years, academic semesters, and lunar years and months etc.), each with a calendar-id which identifies the calendar that supports this granularity. An anchor granularity and an anchor point must be given for each granularity. For the granularities in the user-defined calendar, the user also needs to give the conversion constants for regular mappings and define the C functions for irregular mappings.

The user can integrate the calendars by simply declaring a mapping between a pair of granularities from different calendars. An example is the congruent mapping between Gregorian `days` and the `business_days` in Figure 2.1. In addition, the mapping between Gregorian `seconds` and Astronomy `astronomy_day_hundredths` is an example of weaving different calendars by regular mapping.

Our module also allows the database administrator (DBA) to define additional mapping functions to improve performance. For example, in Figure 2.1, the mapping between the `business_days` and Gregorian `seconds` is an additional mapping. Gregorian calendar and Business calendar originally are linked by a congruent mapping between Gregorian `days` and the `business_days`. If the DBA knows that casting from Business `business_days` to Gregorian `seconds` will be performed repeatedly, then a direct link can be added into the granularity graph. In casting a granule from the `business_days` to Gregorian `seconds`, the direct link will be used instead of the composition of `business_days` to `days`, `days` to `hours`, `hours` to `minutes`, and `minutes` to `seconds`.

A *determinate* timestamp records an instant located sometime during a particular granule. However, if the exact granule the instant is located is unknown, an *indeterminate* timestamp is used to represent the instant [Dyreson & Snodgrass 1993]. The basic operations at mixed granularities are *Cast* and *Scale*. In TSQL2, a *Cast* operation always produces a determinate timestamp by returning the first granule in the result, while a *Scale* operation may produce an indeterminate timestamp. Converting a granule from coarser to finer granularity produces an indeterminate result. For example, converting the day 01/01/1997 to `hours` yields 01/01/1997 00 - 01/01/1997 23, which is the correct scaling result. But the correct casting result for the above conversion is 01/01/1997 00 by taking the first granule. Chapter Five will describe the *Cast* operation in detail. Other operations in SQL-92 generally can be defined in terms of the *Cast*. Our module supports the *Cast* and the *Scale* basic operations, along with other standard operations in SQL-92. This module interface provides 20 different functions and has total 3000 lines of C code. The external data structures and the functions provided in the module are listed in Appendix A.

The hardest part of the module is to ensure that the granularity graph declared by the user is properly formed. In other words, when all granularities and mappings are defined, the module must respond if the granularities are ill-specified.

One possible problem is a circularity in the granularity graph. The granularity graph must be acyclic. The "finer than" relation cannot be defined if there is a cycle in the granularity graph. The module needs to make sure this will not happen.

Another problem is caused by the non-termination in the *Cast* algorithm. Upon further investigating the *Cast* algorithm, we notice that calculating the anchor offset is non-trivial. Since the anchors may be expressed using different granularities, computing the anchor offset involves recursive *Cast* operations. Chapter 5.1 gives an example illustrating the non-termination in the *Cast* algorithm. We state in Chapter 5.1 that in order to ensure the algorithm terminates, the user is limited to define the anchor of a granularity in a previously defined finer granularity. In order to avoid the non-termination in the *Cast* algorithm, when the user finishes the declaration, the module has to check if the anchor of every granularity is defined with respect to a finer granularity.

One more aspect to check is if there exists a unique bottom in the granularity graph. Without a unique bottom in the granularity graph, we can not guarantee a V-path for every pair of granularities. Notice that a granularity graph is not constrained to a complete lattice in the module, so no artificial granularities are introduced.

Our module provides a function called *Declare_Done*. When the users have finished the declarations, they call this function to check the above three requirements. The module returns an error report if any one of above requirements is not satisfied.

Chapter 4

DETERMINING THE OPTIMAL PATH

When performing the *Cast* operation, if the mapping from a source granularity to a destination granularity has not been given explicitly by the user or the DBA, the relationship between the source and destination granularities must be computed as the composition of the existing mappings in the granularity graph. In this chapter, we first examine the paths between a pair of granularities and identify the correct paths for the conversion. Then, we introduce an algorithm to find the optimal path to improve the performance and provide the proof of correctness.

4.1 The Correct Paths

Not all paths between a pair of granularities are suitable for the conversion. Because a *straight-up path* (casting a granule from a finer to a coarser granularity) loses information, any path going up then going down may yield an incorrect result. For example, to cast 02/01/1997 at granularity `days` to granularity `months`, if we choose the path: `days` up to `years` (with result 1997), then `years` down to `months`, we will get the result of 01/1997 instead of 02/1997 at granularity `months`. It is easier to see that *straight-down paths* (from coarser to finer granularity), without losing any information, always produces the correct casting results. Considering a V-path as a straight-down path followed by a straight-up path, although the straight-up path losses information, we still get the correct casting result [Dyreson & Snodgrass 1994A]. To ensure that the least amount of information is lost during the *Cast*, we add the constraint that the path can either be a *straight-line path* (straight-down path or straight-up path) or a V-path. In Figure 2.1, the path between `years` and `hours` (`years` to `months` to `days`) is a straight-line path; the path between `years` and `weeks` (`years` to `months`, `months` to `days` and `days` to `weeks`) is a V-path. To find a correct path between a source granularity α to a destination granularity β, the basic idea is to identify a common ancestor (*CA*). To find a CA, we traverse the granularity graph from both the source and destination granularities. The common granularities encountered are the CAs. For example, in Figure 2.1, the common ancestor of source granularity `years` and destination granularity `weeks` is granularity `days`. The path composes steps from α to the CA, and then from the CA to β. A straight-line path is a special V-path since it's CA is the finer granularity. From now on, we will use the term V-paths to represent both straight-line paths and V-paths. For a properly formed granularity graph, the existing unique bottom in the granularity graph guarantees at least one V-path between any pair of granularities. In the case of more than one CA, we prove that all V-paths will yield the same result.

Before we prove that all V-paths yield the same result, we first need to introduce the formal definition for the "finer" relationship by using Definition 2.1, 2.1 in Chapter Two. Second, we need to explicitly describe the mechanism of the *Cast*.

Definition 4.1.1 *("finer" relationship)*

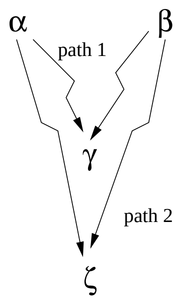


Figure 4.1: A simple granularity graph with two V-paths

*If granularity β is a further partition of granularity α, i.e., if $\forall j_\beta$, $\exists i_\alpha$ **CHR**$(j_\beta) \subseteq$ **CHR**(i_α), then granularity β is finer than granularity α, expressed as $\beta < \alpha$.*

The relation "$<$" is a partial order. For example, `weeks` and `months` are incomparable: `weeks` is not finer than `months`, and `months` is not finer than `weeks`. Furthermore, it is easily seen that the "$<$" relation is transitive due to transitivity of $subseteq$.

With the definition of the finer relation of granularities, we can describe the mechanism used to cast finer mappings and coarser mappings respectively for the *Cast* operation.

If $\alpha > \beta$ then $Cast(i_\alpha, \alpha, \beta) \rightarrow j_\beta$ such that $min(\mathbf{CHR}(i_\alpha)) = min(\mathbf{CHR}(j_\beta))$.
If $\alpha < \beta$ then $Cast(i_\alpha, \alpha, \beta) \rightarrow j_\beta$ such that $\mathbf{CHR}(i_\alpha) \subseteq \mathbf{CHR}(j_\beta)$.

Now we can prove that all V-paths yield the same casting result.

Theorem 4.1.2 *All V-paths from a source granularity to a destination granularity yield the same casting result.*

Proof: As shown in Figure 4.1, let α and β be any two granularities, and path1 and path2 be any two V-paths between α and β with different CAs: γ and ζ. Let's assume that j_β and j'_β are the results of casting i_α from α to β along path1 and path2 respectively. We must show that $j_\beta = j'_\beta$.

We begin from the *Cast* definition:
For path 1:

$$Cast(i_\alpha, \alpha, \beta) = Cast(Cast(i_\alpha, \alpha, \gamma), \gamma, \beta) = Cast(m_\gamma, \gamma, \beta) = j_\beta \ .$$

For path 2:

$$Cast(i_\alpha, \alpha, \beta) = Cast(Cast(i_\alpha, \alpha, \zeta), \zeta, \beta) = Cast(n_\zeta, \zeta, \beta) = j'_\beta \ .$$

First, let's look at the first half of the V-paths (α to the CAs) in the granularity graph. The finer *Cast* operation always returns the first granule in the granularity. Moreover, as γ and ζ are finer granularities of α, the granules i_α, m_γ, n_ζ are aligned, as shown in Figure 4.2. According to the transitivity of $=$, we have,

$$min(\mathbf{CHR}(i_\alpha)) = min(\mathbf{CHR}(m_\gamma)) = min(\mathbf{CHR}(n_\zeta)) \ .$$

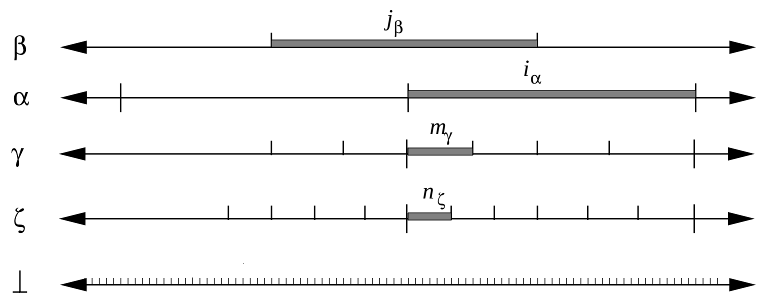


Figure 4.2: The time-lines at granularities α, β, γ, ζ and $\perp$

Now, let's turn to the second half of the V-paths (the CAs to β). Since the paths from the CAs to β are coarser paths, the source granules (m_γ and n_ζ) may not align with the resulting granules, as shown in Figure 4.2. But from the coarser *Cast* mechanism, $min(\mathbf{CHR}(i_\alpha))$ must be in the resulting granules in granularity β. According to the transitivity of $\subseteq$, we have,

$$min(\mathbf{CHR}(i_\alpha)) = min(\mathbf{CHR}(m_\gamma)) \ \in \mathbf{CHR}(j_\beta) \ .$$
$$min(\mathbf{CHR}(i_\alpha)) = min(\mathbf{CHR}(n_\zeta)) \ \in \mathbf{CHR}(j'_\beta) \ .$$

Since chronons is the partition of all granularities and granules in a granularity do not overlap, obviously we have:

$$j_\beta = j'_\beta$$

The paths we examined are arbitrary V-paths, thus we conclude that all V-paths yield the same casting result. □

If more than one CA exists, we choose the one that can be computed most efficiently. The choice of the paths is based on the computation cost: we assume that regular mappings are cheaper than irregular mappings because each irregular mapping in the path requires invoking a potentially costly user-defined function. Therefore, we choose the path with fewest irregular mappings. Of those with an equal number of irregular mappings, we choose the path with the fewest steps.

Finding the optimal path between a pair of granularities is a shortest path problem in an acyclic graph with the edge weights being 1 for irregular mappings and 0 for regular mappings in the granularity graph. But existing shortest path algorithms cannot be used directly to solve the problem because of the added constraint that the path must be a V-path. We have developed two algorithms to find the optimal V-path for a pair of granularities. In the next two sections, we first present an extension to a common shortest path algorithm to find the optimal V-path for a single pair of granularities. We then give a detailed analysis of a dynamic programming approach to solve the all pairs optimal V-paths problem.

4.2 Extending The *Dag-Shortest-Paths* Algorithm

There are various ways to compute the optimal path for a single pair of granularities. The algorithm we have developed is a simple extension to the *Dag-Shortest-Paths* algorithm [Cormen et al. 1990], which,

Algorithm 4.2.1 $Search\_Path(G, \alpha, \beta)$

```
***Input:*** A granularity graph $G$, the source granularity $\alpha$, and
          the destination granularity $\beta$.
***Output:*** The optimal V-path for $\alpha$ and $\beta$.
/* Initialize the variables */
*Dag-Shortest-Paths*$(G, \alpha) \rightarrow d[\,],\ p[\,]$;
*for all* $v \in V$ *do*
        $d_\alpha[v] = d[v]$ ;    $p_\alpha[v] = p[v]$ ;
*Dag-Shortest-Paths*$(G, \beta) \rightarrow d[\,],\ p[\,]$;
/* compute the CA for the optimal V-path */
$ca \leftarrow \emptyset$;    $min \leftarrow \infty$;
***for each*** $v \in V$ ***do***
        ***if***   $(d_\alpha[v] + d[v]) < min$   ***then***
                $min = d_\alpha[v] + d[v]$;
                $ca = v$
/* compute the optimal V-path using $p_\alpha[ca]$ and $p[ca]$ */
***return***   $\alpha \rightarrow \cdots \rightarrow ca \rightarrow \cdots \rightarrow \beta$;
```

Figure 4.3: The *Search_Path* Algorithm

given a weighted *dag* (directed acyclic graph) $G = (V, E)$, computes shortest paths from a single source in $O(|V| + |E|)$ time. Here, V is the set of nodes in the graph and E is the set of edges.

The *Dag-Shortest-Paths* topologically sorts the dag from the source vertex to get a linear ordering on the vertices. If an edge points from a vertex x to vertex y, then x precedes y in the topological sort. The *Dag-Shortest-Paths* algorithm then makes one pass over the vertices in the sorted order to compute the shortest paths from the source vertex.

The intuition for the extended algorithm is that every V-path is built up from two directed paths that meet at a granularity (CA). Since a granularity graph is a weighted dag with edges directed from coarser to finer granularities, we run the *Dag-Shortest-Paths* algorithm for the source (α) and the destination (β) granularities respectively to compute the shortest directed paths from α and β to all finer granularities. For a particular vertex v, we claim that the V-path ($\alpha \rightarrow v \rightarrow \beta$), i.e., the combination of the two directed shortest paths (α to v and β to v) is the shortest V-path between α and β going through v. If the V-path is not the shortest V-path going through v, then there must exist a shorter path either from α to v or β to v. This contradicts with the results of the *Dag-Shortest-Paths* algorithm. The optimal V-path for α and β is then computed by comparing all V-paths for different v and selecting the shortest one.

The algorithm maintains the variable $d[v]$ for each granularity v, which is the weight of the shortest path from α to v. For each granularity in the graph, a predecessor $p[x]$ that is either another granularity or Nil, is used to present the path. Figure 4.2 shows a pseudo code version of this algorithm called *Search_Path*. The algorithm finds shortest paths from α to all vertices (granularities) and from β to all vertices. It then finds the vertex whose shortest paths from α and from β have shortest total length.

The running time of the *Dag-Shortest-Paths* algorithm is $O(|V| + |E|)$. In the *Search_Path* algorithm, we run the *Dag-Shortest-Paths* algorithm twice and the computation of CAs is actually done in the second run of *Dag-Shortest-Paths*. Computing the optimal CA takes O(|V|) time, since there are at most |V| of CAs in the graph. Thus, the total running time of the extended algorithm is $O(|V| + |E|)$.

4.3 A Dynamic Programming Algorithm

The *Search_Path* algorithm can compute the optimal V-path between a single pair of granularities. An alternative is to compute all optimal paths at DBMS generation time and cache the results. This approach is quite appealing for large DBMSs. However, for an application program dealing with only a small subset of the granularities, computing the path as needed certainly provides better performance. Our approach is to allow the user to request the precomputation. For application programs, we use a lazy caching strategy (compute the optimal V-path as demanded and cache the result) to avoid recomputation.

The algorithm described in last section finds the optimal V-path for a single pair of granularities. For database applications, we have developed an algorithm to satisfy the V-path constraint and use the top-down dynamic programming method to solve the all-pairs optimal V-paths for a granularity graph. This algorithm can be used to compute the optimal path for a single pair of granularities as needed or can be used at DBMS generation time to determinate all-pair optimal V-path. Next, we will give a detail analysis of our approach and describe the algorithm, then we will argue that the algorithm can be extended to solve all-pairs optimal V-paths for a granularity graph. Finally, we will prove the correctness of the algorithm.

Dynamic programming is applicable if subproblems share subproblems. The intuition behind our algorithm is that every V-path is formed by a smaller V-path. We observe that the optimal V-path for a single pair of granularities can be solved by combining the optimal solutions to subproblems. (The formal proof is given later in this section.) Given a pair of granularities α and β, if we consider the optimal paths from α to all one step finer granularities of β as the subproblems, then the optimal V-path for α and β can be found by comparing all V-paths composed of the optimal V-paths from the subproblems and the one step finer paths. Thus, to find the optimal V-path, we traverse down the granularity graph from the source and the destination granularities respectively to solve the subproblems first. The V-path constraint is achieved by traversing the graph following the finer paths recursively. Each subproblem is computed just once and the solution is stored to avoid traversing the graph multiple times.

First, let's introduce the 2-dimensional variables (all are integer arrays) used to define a path. For convenience, we call these variables *path-determining* variables.

$first[\alpha][\beta]$ The first granularity encountered in the V-path from source granularity α to destination granularity β.

$icost[\alpha][\beta]$ The number of irregular mappings in the V-path from source granularity α to destination granularity β.

$tag[\alpha][\beta]$ Enumerated type, with disjoint tags:

- tag_c Indicates that the path from α to β is a coarse("c") path (all edges in the path from finer to coarser granularities).
- tag_f Indicates that the path from α to β is a finer ("f") path (all edges in the path from coarser to finer granularities).
- tag_b Indicates that the path from α to β is a V shape path. We use the bottom granularity to represent the finest granularity encountered in the path, so the "b" stands for "bottom".
- tag_s Indicates that the path from α to β is a congruent path. The "s" stands for "same" to distinguish from the coarser path.
- tag_u $tag[x][x] = tag\_u$ marks the granularity x as unvisited ("u"). Note that this tag is only used for diagonal entries.

Below, we use show the above variables for the granularity graph in Figure 2.1.

$first[\texttt{decades}][\texttt{days}] = \texttt{years}$	$first[\texttt{days}][\texttt{decades}] = \texttt{months}$
$icost[\texttt{decades}][\texttt{days}] = 1$	$icost[\texttt{days}][\texttt{decades}] = 1$
$tag[\texttt{decades}][\texttt{days}] = tag\_f$	$tag[\texttt{days}][\texttt{decades}] = tag\_c$
$first[\texttt{years}][\texttt{weeks}] = \texttt{days}$	$first[\texttt{weeks}][\texttt{years}] = \texttt{days}$
$icost[\texttt{years}][\texttt{weeks}] = 1$	$icost[\texttt{weeks}][\texttt{years}] = 1$
$tag[\texttt{years}][\texttt{weeks}] = tag\_b$	$tag[\texttt{weeks}][\texttt{years}] = tag\_b$
$tag[\texttt{days}][\texttt{business\_days}] = tag\_s$	$icost[\texttt{days}][\texttt{business\_days}] = 0$
$first[\texttt{days}][\texttt{days}] = \texttt{days}$	$icost[\texttt{days}][\texttt{days}] = 0$

Notice that for V-paths, we use $first[\ ][\ ]$ to represent the CA of source and destination granularities. Thus $first[\texttt{years}][\texttt{weeks}]$ is granularity `days` instead of granularity `months`. Given $first[\alpha][\beta]$ and $tag[\alpha][\beta]$, we can easily identify the path between α and β by recursively discovering all intermediate steps. If $tag[\alpha][\beta]$ equals tag_c (straight-up path) or tag_f (straight-down path), we start from the source granularity and recursively find out the intermediate steps until reach the destination granularity. For example, to find the path between Gregorian `years` and `days`, we first compute $first[\texttt{years}][\texttt{days}]$ with result: `months`; then compute $first[\texttt{months}][\texttt{days}]$ resulting in `days` to yield the path `years` to `months` to `days`. If $tag[\alpha][\beta]$ is tag_b (V-path), then $first[\alpha][\beta]$ is first computed to get the CA. We then find the path by traversing down to the CA from both source and destination granularities. For example, to find the V-path between `years` and `weeks`, $first[\texttt{years}][\texttt{weeks}]$ is initially computed to get the CA: `days`. With the resulting CA, we obtain one half of the V-path (`years` to `weeks` to `days`) by computing the straight-down path from `years` to `day`, and the other half (`weeks` to `days`) by computing $first[\texttt{weeks}][\texttt{days}]$. The final V-path is the combination of the two straight-down paths: `years` to `months`, `months` to `days`, and `days` to `weeks`.

The granularity graph in our module is represented as a collection of adjacency lists. Each node has pointers referencing a regular finer mapping list, an irregular finer mapping list and a congruent mapping list. The system builds the granularity graph as the user declares each granularity and mapping. The data structures for the granularity graph are listed in Appendix B.

Initially, we set tag to tag_u (unvisited), and the other variables to the maximum number of granularities to indicate that there is no path between any pair of granularities. To find a single-pair optimal path, the source and destination granularities are stored in a queue (top_queue). The strategy is to follow the depth-first search to visit the unmarked granularity from the *tops* (the granularities in the top_queue) to finer granularities whenever possible, and set the path-determining variables on the way down. Notice that the first pass from the top_granularity always reaches the bottom $\perp$. At a granularity g, whenever one of the g's finer granularity x has been explored, the algorithm tests whether we can improve the best V-paths from g to all other granularities (y) found so far by going through the finer granularity x and, if so, updates $first[g][y]$ and $tag[g][y]$.

The *Find_Path* algorithm is given in Figure 4.4. Figure 4.6 also shows how to update the path-determining variables between a source granularity g and the destination granularity y under different situations. Assuming x is a finer granularity of g and has been explored, if the path going through x (g to x to y) costs less than current path from g to y, then update the variables. The solid arrow is the one-step finer path from g to x; the dotted line represents the current path between g and y; and the dashed arrow indicates the optimal V-path from x to y which has been computed during the visiting of x. In Figure 4.6(a), y is coarser than x. Notice, the path from g to y becomes a V-path; so we set $tag[g][y]$ to tag_b and $first[g][y]$

```
Algorithm 4.3.1 $Find\_Path(G)$
Input: A granularity graph
Output: $first[\alpha][\beta]$
/* Initialize the variables */
for each $\alpha$ and $\beta \in G$ do
        assign $max\_gran\_num$ to $first[\alpha][\beta]$, and $icost[\alpha][\beta]$;
        $tag[\alpha][\beta] \leftarrow tag\_u$;
build the $top\_queue$;
for each granularity $g \in top\_queue$ do
        $Do\_Close(g)$;
```

Figure 4.4: Algorithm $Find\_Path$

```
Algorithm 4.3.2 $Do\_Close(g)$
Input: A granularity $g$
Output: A portion of $first[\alpha][\beta]$
/* Traverse down the tree by following $g$'s finer lists */
for each $x \in g'$s finer lists $(reg\_finer\_list, irreg\_finer\_list$ and $congruent\_list)$ do
        $first[g][x] \leftarrow x;$    $first[x][g] \leftarrow cg;$
        $tag[g][x] \leftarrow tag\_f;$    $tag[x][g] \leftarrow tag\_c;$
        if $x \in irreg\_finer\_list$ then $icost[g][x] \leftarrow 1$ else $icost[g][x] \leftarrow 0$;
        if $tag[x][x] = tag\_u$ then $Do\_Close(x);$
        for each $y \in G$ and $y \neq g$ do
                if the cost of $g \rightarrow x \rightarrow y \leq$ the cost of $g \rightarrow y$
                    /* Update the variables */
                    Assign new values to $first[g][y],$ $first[y][g],$ $tag[g][y]$, and $tag[g][y]$,
                    and update $icost$ as shown in Figure 4.6, according to values of $tag[x][y]$.
```

Figure 4.5: Algorithm Do_Close

to x. In Figure 4.6(b), y is finer than x. In Figure 4.6(c), there is a V-path between x and y so the updated path from g to x to y is also a V-path. In the final graph 4.6(d), y is congruent with x. The updated $icost$ is same for all cases and equals the sum of the $icost$s of the paths g to x and x to g, where the $icost$ of the one-step finer path (g to x) is 1 for irregular mapping and 0 for regular and congruent mappings.

The algorithm always traverses down following the finer path, this guarantees the V-path constraint. This approach examines all the possible V-paths between a pair of granularities and picks the cheaper path each time it encounters a new V-path between a pair of granularities, so the final results in $first[\alpha][\beta]$ are the optimal results. In the following we give a formal proof that this algorithm is correct and does indeed compute the optimal paths.

Lemma 4.3.1 *The updated paths represented by the variable first are V-paths.*

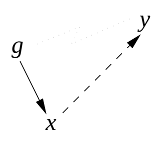
Proof: Initially, each entry of $first$ contains the maximum number of granularities to indicate that there is no path between any pair of granularities. Considering Algorithm Do_Close, the values of the $first$ array are updated only in following two cases:

1. Traversing down a one-step finer path (from g to x)

icost[g][y] = icost[y][g] ← icost[g][x] + icost[x][y];

tag_x[x][y]

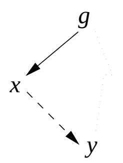
(a) tag_c:



first[g][y] = first[y][g] ← x;

tag[g][y] =tag[y][g] ← tag_b;

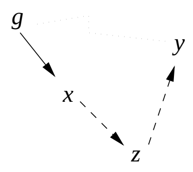
(b) tag_f:



first[g][y] ← x; first[y][g] ← first[y][x];

tag[g][y] ← tag_f; tag[y][g] ← tag_c;

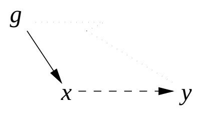
(c) tag_b:



first[g][y] = first[y][g] ← first[x][y];

tag[g][y] = tag[y][g] ← tag_b;

(d) tag_s:



first[g][y] ← x; first[y][g] ← first[y][x];

tag[g][y] ← tag_f; tag[y][g] ← tag_c;

Figure 4.6: Updating the path-determining variables for the $Find\_Path$ algorithm

$first[g][x] = x; \qquad first[x][g] = g$

The path between g and x is obviously a V-path (a straight-up, a straight-down path or a congruent path).

2. Updating the path between g and y through finer granularity x.

   This is a recursive graph problem. Due to the existence of an unique bottom in the graph, the first pass over the graph always reaches the bottom. The following passes traverse the graph from a granularity down to the finest unvisited granularity (CA) to build the smallest V-paths, as shown in Figure 4.6(a). The larger V-paths then can be built using the smaller ones as shown in Figure 4.6(c). If granularity y is finer or congruent with x, then the resulting path is a straight down path from g to y or a straight up path from y to g as in Figure 4.6(b) and (d). Depending on the relationship between x and y, which is built recursively as in Figure 4.6, the new path from g to y through finer granularity x can only be a straight-down path or a V-path.

We conclude that the updated paths represented by the variable $first$ are V-paths. In other words, the final paths are V-paths. □

Theorem 4.3.2 *If we run Algorithm $Find\_Path$ on a pair of granularities, then at termination, the paths represented by $first[x][y]$ are the optimal V-paths.*

Proof: By Lemma 4.3.1, the paths represented by *first* are V-paths when we run the algorithm. We claim that our algorithm is basically a top-down, dynamic-programming

algorithm (named *Memoization*) to find the optimal V-path between a pair of granularities. We examine two key ingredients that must exist to ensure an optimal solution [Cormen et al. 1990].

1. The optimal substructure of the optimal V-path problem.

   We say a problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems. We use the same notation in Algorithm *Do_Close* to illustrate the optimal substructure in this problem.

   Assuming the path (g to x to z to y) is the optimal V-path between granularities g and y (here, x is a one-step finer granularity of g and z is the bottom), then the subpath (x to z to y) must be the optimal path for x and y. We can proof this by contradiction. If there were a better V-path for x and y, substituting the path in g and y would produce another optimal V-path whose cost was lower than the original path: contradiction. Using the same argument, we claim that, for an optimal straight path, assuming g is the coarser granularity, then the subpath (x to y) is the optimal path for x and y. Thus, an optimal solution to an instance of the optimal V-path problem contains within it optimal solutions to subproblem instances. Note that we always traverse down from the coarser granularity. If we reach bottom z, then the optimal subpath of the straight-path (z to y) is the path: the finer granularity of y to z.

2. A recursive solution

   The second step is to define the value of an optimal solution recursively in terms of an optimal solutions to subproblems.

   Let $V\_icost[g, y]$ be the number of irregular mappings in the optimal V-path from g to y, and $I\_icost[g, y]$ be the number of irregular mappings in straight path from coarser to finer, we can define $V\_icost[g, y]$ and $I\_icost[g, y]$ recursively as follows. If $g = y$, there is no cost. To compute

$V\_icost[g,y]$ and $I\_icost[g,y]$ when $g \neq y$, we take advantage of the structure of an optimal solution from step 1. Let's assume that the optimal path for g and y is through x which is finer than and adjacent to g in the granularity graph. We have

$V\_icost[g,y] = V\_icost[x,y] + I\_icost[g,x]$, and

$I\_icost[g,y] = I\_icost[x,y] + I\_icost[g,x]$.

Where,

$$I\_icost[g,x] = \begin{cases} 1 & for\ irregular\ mapping \\ 0 & for\ regular\ mapping \end{cases}$$

The above recursive equation assumes that we know which finer granularity (x) to form the optimal V-path, which we don't. Since x is not the only finer granularity of g and the optimal V-path must be constructed by going through one of the finer granularities, we need to check them all to find the best V-path. Thus, the recursive definition for the minimum cost of the optimal V-Path becomes

$$V\_icost[g,y] \ = \min \begin{cases} I\_icost[g,y] \\ I\_icost[y,g] \\ \min\limits_{x\ \in\ g's\ finer\ lists} \{V\_icost[x,y] + I\_icost[g,x]\} \end{cases}$$

$$I\_icost[g,y] \ = \begin{cases} \infty & if\ g\ finer\ than\ y \\ \min\limits_{x\ \in\ g's\ finer\ lists} \{I\_icost[x,y] + I\_icost[g,x]\} \end{cases}$$

Algorithm *Find_Path* is a top-down algorithm based on the above recurrence to compute the optimal V-path. This algorithm is one of the ways to calculate the tables defined above. Combining with Lemma 4.3.1, we conclude that the final results in $first$ array are the optimal V-paths. □

If we search for all top-granularities (granularities with no coarser but some finer or congruent granularities) in the granularity graph and store them instead of the source and destination granularities in the top_queue, the algorithm is extended to solve the all-pairs optimal V-paths for the granularity graph. This is easy to see because if we traverse down from top-granularities, each subproblem will be encountered to solve the optimal V-paths for the entire graph.

What is the running time of the above algorithm? If there are $|V|$ granularities and $|E|$ number of mappings (edges) in the granularity graph $G(V,E)$, building the top queue takes times $O(|V|)$. Each granularity is visited at most once, and procedure Do_Close is called exactly once for each granularity in G. During the execution of Do_Close, the loop for the one-step finer granularities of g is executed $|finer\_granularities(g)|$ (the number of one-step finer granularities of g) times. Since

$$\sum_{cg \in G} |finer\_granularities(g)| = O(|E|),$$

and there is a testing loop at each granularity, the total running time of Algorithm $Find\_Path$ is $O(|V|) + O(|V||E|)$, or $O(|V||E|)$.

Chapter 5

THE *Cast* OPERATION

The $Cast$ operation is performed to correctly convert time from the source to the destination granularity. Given the optimal path, the *Cast* algorithm needs to convert a time value from one granularity to another in each step of the path. For an irregular mapping step, the algorithm simply invokes the user-defined C functions to convert the granule. For a regular mapping or a congruent mapping, the granule can be converted by multiplying or dividing by the conversion constant with an anchor adjustment.

For example, to cast an instant, the i^{th} granule, from granularity α to granularity β, assuming a regular mapping with conversion constant C, then

$$Cast(i_\alpha, \alpha, \beta) = i_\alpha \times C + \textit{anchor\_offset}$$

We defined in Chapter Two that an anchor is the 0^{th} granule of a granularity. In the specification of a granularity graph, the anchor may be defined in terms of a granularity that also needs to be converted. The user declares granularities, each with an anchor, the latter in an anchor granularity. As stated in Chapter Two, we don't want to require the user to use chronons for the anchor granularity; in other words, we want to allow non-chronon anchor granularities. For example, a student wishes to design a special calendar for his academic activities with year origin defined as the first year he was in this department. He knows the origin is Fall,1995 (the anchor of his special calendar which is defined on the Gregorian years), but most likely, he has no idea which chronon that is. We also don't want to use a same anchor for all granularities as we stated in Chapter two.

In this Chapter, we will first describe the problem caused by computing the anchor offset, and the solution we come up with, then we present the algorithm for the *Cast* operation and provide the proof of correctness.

5.1 Computing The Anchor Offset

To compute the anchor offset, we need to call the *Cast* algorithm recursively. Unfortunately, this will sometimes cause the algorithm to never halt if the granularity graph declared by the user is not properly formed. We give the following example to illustrate the termination problem.

Figure 5.1 is a granularity graph with three user-declared granularities (α, β, γ) as well as the bottom granularity (chronons). In this particular example, α and γ anchors on chronons and β anchors on α. Note that α is a coarser granularity than β. When we say β's anchor is the 4th granule of α, we really mean that β's anchor is the first chronon in the 4th granule of α. The problem is that the software has no information about what β's anchor is in chronons. Suppose we are to cast the i^{th} granule from β to γ, we need to know the anchor offset of β and γ. The following are the steps needed to compute the anchor offset.

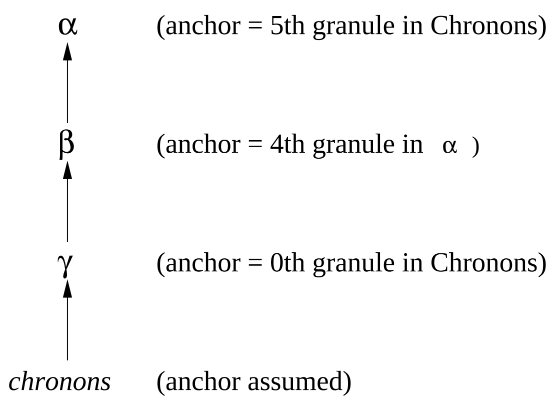

α
(anchor = 5th granule in Chronons)
β
(anchor = 4th granule in α)
γ
(anchor = 0th granule in Chronons)
chronons
(anchor assumed)

Figure 5.1: A granularity graph with nontermination problem

1. We need to calculate anchor of β expressed on granule of γ.
2. Given the anchor of β on α (the fourth granule of α), we need to call the *Cast* function to cast the 4th granule from α to γ.

   $Cast(i_\beta, \beta, \gamma) = i_\beta \times C + Cast(4_\alpha, \alpha, \gamma)$

   The only path we can have in the granularity graph is: $\alpha \rightarrow \beta \rightarrow \gamma$
3. Convert the fourth granule in α to granule in β. The result is obviously the 0th granule in β.
4. Then convert the 0th granule in β to γ, and we go back to step 1.

This example shows that a user, when defining granularities and anchors, can indeed get into trouble, even when the granularity graph is clearly acyclic. To avoid this problem, we will require that the anchor granularity be finer than the granularity being defined. This is a reasonable constraint. It will be checked at DBMS generation time; the module will report an error if this constraint is not satisfied. Given the constraint, we can prove that the *Cast* algorithm always terminates. We call the computation of anchor offset as "$anchoring$" to differentiate it from the real *Cast* operation. In the next two sections, we first present the *Cast* algorithm, then prove the algorithm terminates.

5.2 The *Cast* Algorithm

Suppose we are to cast a granule g_α from granularity α to β and a granule g_β from granularity β to α. Assuming that α is coarser than β,

$$Cast(g_\alpha, \alpha, \beta) = \lfloor g_\alpha \times C^{\alpha \rightarrow \beta} + \Delta^\alpha_\beta \rfloor$$
$$Cast(g_\beta, \beta, \alpha) = \lfloor g_\beta \times C^{\beta \rightarrow \alpha} - \Delta^\alpha_\beta \times C^{\beta \rightarrow \alpha} \rfloor$$

where,

$C^{\alpha \rightarrow \beta} = \frac{1}{C^{\beta \rightarrow \alpha}}$ is the conversion constant from α to β, and
Δ^α_β is the anchor of α expressed in granule of a finer granularity β.
To compute Δ^α_β, we use Δ^α_π, which is α's anchor defined in the finer granularity π. Then,

```
**Algorithm 5.2.1** $Cast(g, from, to)$
***Input:*** The granule to be converted ($g$), the source granularity ($from$), and
        the destination granularity ($to$).
***Output:*** The granule in the destination granularity ($result$).
***if*** $from = to$ ***then*** ***return*** $g$;
/*find the path between the source and the destination granularities */
$path \leftarrow Find\_Path(from, to)$;
$result \leftarrow g$;
***while*** ($path$ is not null)
    ***switch*** $(path.mapping\_type)$
        ***case*** $irreg\_finer\_mapping$:
            $result \leftarrow f_{irregular\_finer\_mapping}(result, path.from, path.to)$;
        ***case*** $irreg\_coarser\_mapping$ :
            $result \leftarrow f_{irregular\_coarser\_mapping}(result, path.from, path.to)$;
        ***case*** $reg\_finer\_mapping$ ***or*** $congruent\_mapping$ :
            $anchor\_v$ is the anchor value of ($path.from$);
            $anchor\_granularity$ is the anchor granularity of ($path.from$);
            $result \leftarrow \lfloor result \times (path.C) +$
                $Cast(anchor\_v, anchor\_granularity, path.to) \rfloor$;
        ***case*** $reg\_coarser\_mapping$ :
            $anchor\_v$ is the anchor value of ($path.to$);
            $anchor\_granularity$ is the anchor granularity of ($path.to$);
            $result \leftarrow \lfloor result \times (path.C) -$
                $Cast(anchor\_v, anchor\_granularity, path.from) \times path.C \rfloor$;
    $path \leftarrow path.next$;
***return*** $result$;
```

Figure 5.2: The $Cast$ algorithm

$$\Delta^{\alpha}_{\beta} = Cast(\Delta^{\alpha}_{\pi}, \pi, \beta)$$

Note that for congruent mapping, either one of the above *Cast*s can be used to compute the anchor offset.

To perform the *Cast* operation from a source to a destination granularities, we need to find the optimal path. The optimal path is stored in a linked list named $path$, of steps. The structure also contains the source ($from$) and the destination (to) granularities, the mapping types ($mapping\_type$), and for regular mapping, the conversion constant (C), for irregular mapping, two pointers pointing to the user-defined C mapping functions. Figure 5.2 gives the *Cast* algorithm.

The *Cast* algorithm in Figure 5.2 actually does the path finding and recursive anchoring operation at each step of the conversion. This is considered inefficient for both the application programs and databases applications. In either case, the previously computed optimal V-paths and anchor offsets are cached to avoid the recomputation. This will be further discussed in next chapter. Recalling that we do not consider the recursive anchor offset computing when we derive the $Find\_Path$ algorithm, the main reason of ignoring the anchor offsets is that an anchor offset is computed only when it is first encountered. The computed anchor offset is stored so the value is simply looked up each subsequent time it appears. With the caching strategy, the anchor offset computing won't affect the choice of an optimal V-path.

Notice that in Algorithm 5.2.1, for any determinate instance, the *Cast* always returns a determinate result. For any anchored indeterminate instance, the start and the end of the instance are cast separately to get the indeterminate result. Our module provides functions to perform all the standard SQL-92 operations for a multicalendar system. The functions are listed in Appendix A.

5.3 Proof Of Termination

The proof of termination involves constructing a bipartite graph for the *Cast* operation, and using a partial order to argue the nonexistence of cycles in the graph.

The directed bipartite graph (G_b) is constructed based on the granularity graph (G_g). To distinguish the bipartite graph and the granularity graph, we use the subscript b to indicate the bipartite graph.

Let $G_b = (V_b, E_b, U_b)$ be the bipartite graph. The vertices are partitioned into two disjoint subsets (V_b and U_b) such that there is no edge connecting two vertices from the same subset.

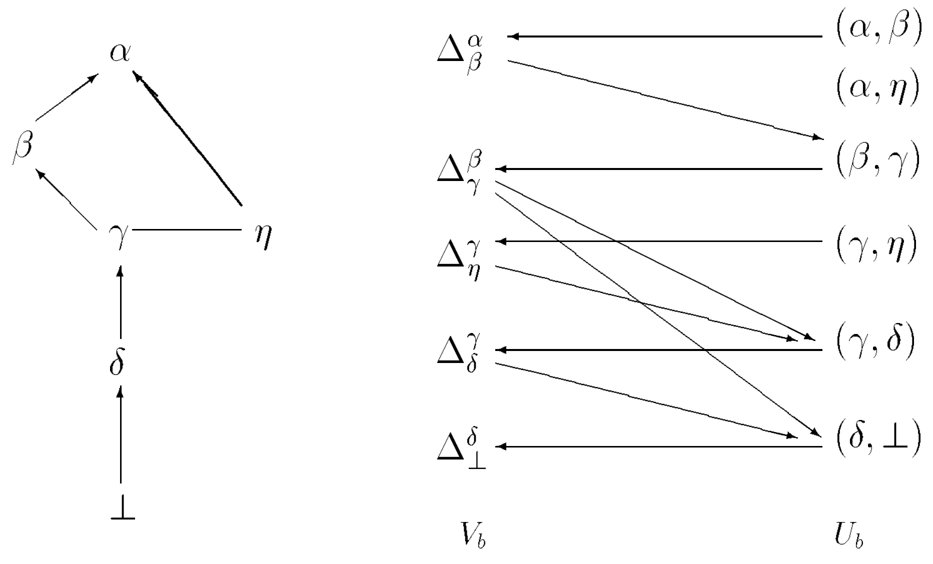
V_b is a set of anchor offsets needed to be computed in the granularity graph G_g. A vertex in V_b is denoted by Δ_y^x, where x is a coarser or a congruent granularity of y in G_g and the mapping between x and y is either regular or congruent. Remember there is no need to compute the anchor offset for an irregular mapping.

U_b is the other set of vertices in which each vertex represents an one-step mapping in the granularity graph. A vertex in U_b is denoted by (x, y) to indicate the one-step mapping between x and y in the granularity graph.

E_b is a set of directed edges connecting vertices between U_b and V_b. The edges are constructed by following the processes taken to compute an anchor offset. An edge points from a vertex $u_b = (x, y)$ in the set U_b to a vertex $v_b = \Delta_y^x$ in the set V_b if the mapping (x, y) requires computing the anchor offset Δ_y^x. Note that only regular and congruent mappings require to compute the anchor offsets; for an irregular mapping, there is no edge coming out of the node in U_b. Given the anchor of x is in π, the path p from π to y is needed to compute the anchor offset (Δ_y^x). An edge points from an vertex $v_b = \Delta_y^x$ in V_b to a vertex u_b in U_b if the one-step path in u_b is contained in the corresponding path p.

Figure 5.3 shows a simple granularity graph G_g and the corresponding bipartite graph G_b. In the granularity graph, the anchor of α is in γ, the anchors of β, γ and δ are in the chronons $\perp$ and the anchor of η is in δ. Note that the mapping between α and η is an irregular mapping and the mapping between γ and η is a congruent mapping. Let's follow the anchor offset computation of Δ_β^α to construct the edges for the bipartite graph. Given α's anchor in γ, the casting path is $\gamma \to \beta$. Since this is a coarser mapping, the required anchor offset is Δ_γ^β. The anchor of β is defined in $\perp$, so to compute Δ_γ^β, the needed path is $\perp \to \delta \to \gamma$. Since the path is composed of two steps, there are two edges from Δ_γ^β, pointing to ($\perp, \delta$) and (δ, γ). Following the anchor offset computation of Δ_η^γ will give us the rest of edges in the bipartite graph. For this particular example, starting from Δ_β^α and Δ_η^γ yields the complete bipartite graph. Note that there is no edge coming out of $\Delta_\perp^\delta$ in V_b and (α, γ) in U_b. The former is because the anchor offset is given for the granularity graph; the latter is because the mapping between α and γ is irregular. Since the *Cast* algorithm effectively follows the edges in the graph, if there is an cycle in the graph G_b, the *Cast* algorithm will not terminate.

To ensure the termination of the *Cast* algorithm, we add the constraint that the anchor of a granularity should be defined in a finer granularity. We claim that the *Cast* algorithm will always terminate for a granularity graph satisfying the above constraint. Before we give the formal proof of the claim, let's define a partial order for the vertices (the one-step paths) in set U_b. We use a subscript p on this partial order to distinguish the coarser relation for the one step paths in U_b from the coarser relation for granularities.



(a) G_g (b) G_b

Figure 5.3: (a) A simple granularity graph. (b) The corresponding bipartite graph.

Definition 5.3.1 *coarser ("$>_p$") relationship*
Given two vertices in set U_b in the bipartite graph: $u_1 = (x_1, y_1)$ and $u_2 = (x_2, y_2)$,
$u_1 >_p u_2$ ***iff***

(1) $(max(x_1, y_1) > max(x_2, y_2)) \quad \vee$
(2) $((x_1 \sim y_1 \sim x_2) \wedge (y_2 < x_2)) \quad \vee$
(3) $((x_1 \sim y_1 \sim y_2) \wedge (x_2 < y_2))$

In above definition, max returns the coarser granularity and chooses an arbitrary granularity if the two granularities are congruent. The first condition in Definition 5.3.1 is straight forward: the path in U_b with the coarser granularity is the coarser path. The second and third conditions are for special cases: we define a congruent path u_1 to be coarser than a path composed of a granularity congruent with granularities in u_1 and a finer granularity. Applying the definition on Figure 5.3(b), we have

$$(\alpha,\beta) >_p (\beta,\gamma) >_p (\gamma,\eta) >_p (\gamma,\delta) >_p (\delta,\perp) \quad \text{and}$$

$$(\alpha,\eta) >_p (\beta,\gamma) >_p (\gamma,\eta) >_p (\gamma,\delta) >_p (\delta,\perp) .$$

Since the definition of the "$>_p$" is based on the relation between granularities, as in the finer relation for granularities, the "$>_p$" is a partial order. In Figure 5.3(b), (α,β) is not coarser than (α,γ), and (α,γ) is not coarser than (α,β). The "$>_p$" also has the following two properties, which follow from its definition:

- Transitivity: if $u_i >_p u_j$ and $u_j >_p u_k$ then $u_i >_p u_k$.
- Inreflexivity: $u_i \not>_p u_i$.

With Definition 5.3.1 and the assumption about the finer anchor granularity, we argue that starting from any vertex in the G_p built on a granularity graph satisfying the constraint about the finer anchor granularity, and following the direction of edges to compute an anchor offset, the encountered vertices in the set U are in descending order with respect to the "$>_p$" relation.

Lemma 5.3.2 *Given a granularity graph, if the anchor of each granularity is defined in a finer granularity, then following an arbitrary edge in the corresponding bipartite graph, for any two consecutive vertices u_i and u_{i+1} in U_b, we have $u_i >_p u_{i+1}$:*

Proof: Let u_i be an arbitrary one-step path (α, β) in U_b and α's anchor is in γ. Assuming β is finer than α, an edge in the bipartite graph points from u_i to Δ^{α}_{β}. The outward edges from Δ^{α}_{β} should point to each step of the path p from γ to β in order to compute the anchor offset. The path p must be either a straight-up path, a straight-down path or a V shape path in the granularity graph. Let u_{i+1} be any consecutive vertex in the bipartite graph, then the path p can be expressed as $\gamma \cdots \beta$.

1. For a straight-up path, due to the transitivity of granularities, we have

   $$\gamma < \cdots < \beta \ .$$

   Along with the assumption that $\beta < \alpha$, it's straight forward to see that all granularities in the path p is finer than α. By Definition 5.3.1 (1), we prove that $u_i >_p u_{i+1}$.

2. For a straight-down path,

   $$\beta < \cdots < \gamma \ .$$

   Given $\gamma < \alpha$ from the constraint about the finer anchor granularity, as in step 1, all granularities in the path p are finer than α resulting in $u_i > u_{i+1}$.

3. For a V shape path, the CA is finer than β and γ by the CA definition. From step 1 and step 2, we immediately have $u_i >_p u_{i+1}$.

If α is congruent with β, because of the assumption of the finer anchor granularity, the path p from γ to β can only be a straight-up path or a V-path. Applying step 1 and step 3, and plus Definition 5.3.1 (2) and (3), we also find $u_i >_p u_{i+1}$. Combining the all steps, we have proved the Lemma. □

Now we can further prove that given the constraint, the *Cast* algorithm terminates.

Theorem 5.3.3 *Given a granularity graph, if the anchor of each granularity is defined in a finer granularity , then the Cast algorithm on any pair of granularities always terminates.*

Proof: This is proved by contradiction.

Suppose the algorithm does not terminate. Since each step of the *Cast* algorithm traverses an edge of G_b, and since there are finite number of nodes in this graph, there must be a cycle in the bipartite graph G_b. The cycle contains two vertices u_i and u_j in U_b such that

$$u_i \rightarrow \cdots \rightarrow u_j \rightarrow \cdots \rightarrow u_i \ .$$

Then according to Lemma 5.3.2 and the transitivity of "$<_p$", we have $u_i >_p u_i$.

This is in contradiction with the inreflexivity of "$<_p$", so we have proved Theorem 5.3.3. □

Chapter 6

PERFORMANCE

Performance is an important issue. Our goal is to provide a package to support the multiple time granularities within both application programs and databases. A scalable solution, capable of handling hundreds of granularities, is desired. We use several strategies to improve the performance.

The first strategy is to compose the path for regular mappings and congruent mappings with the same anchor. We define $acost$ as the cost of the anchor adjustments in the path. For V-paths between a pair of granularities, the $icost$ (the number of irregular mappings) is the major factor to choose which path is better. If two granularities have the same anchor and the same anchor offset, then the $acost$ of these two granularities is zero. In the case of the same $icost$, the path with smaller $acost$ will be selected. The $acost$ is also used to improve the performance further. For a straight-line path (either finer or coarser path), if the $acost$ is zero in each step, and there is no irregular mapping in the path, then the path can be combined into a single step with a new conversion constant to reduce the mappings. For example, in Figure 2.1, the mappings from `days` $\rightarrow$ `hours` $\rightarrow$ `minutes` can be combined as `days` $\rightarrow$ `minutes` with a new conversion constant 1440.

Since the number of the anchor offsets needed to be computed in a granularity graph is a fixed number, the number of edges, we use a lazy caching strategy to improve the performance. An anchor offset is computed on demand and stored in the front of the cache, which is a linked list. Each subsequent time this anchor offset is encountered, the value in the list is used and the anchor offset is moved to the front of the list. When the cache is full, the values at the back of the list are freed to give space for the newly computed anchor offsets. We envision that if there aren't a great many edges, the cache can be made large enough to hold all anchores.

The performance of computing the optimal V-paths is more complicated. We emphasize that our module is designed to support both application programs and databases. As we stated in Chapter 4.3, for database applications, during the graph specification the optimal V-paths computation can be very slow, but the *Cast* must be fast at query-time. If the set of granularities is small, the optimal V-path between any two pair of granularities can be determined at DBMS generation time, as the declaration is done. However, as there are $O(|V|^2)$ optimal V-paths in a given granularity graph $G(V, E)$, for many application programs that cast and scale between a small subset of granularities, computing all optimal V-paths is overkill. Our solution is to allow precomputation optionally, as a separate user command. This would be appropriate for database applications. For application programs, we use the lazy caching approach: the optimal V-path is calculated on demand and cached after the computation.

To quantify the actual cost of computing an optimal V-path, we ran a series of tests on the multicalendar granularity graph shown in Figure 2.1, with 18 granularities and 20 edges. We start with a sequence of pairs of randomly selected granularities. As the length of the sequence increases, we expect that the average cost per optimal V-paths (termed the *V-cost*) will drop for the lazy caching approach. We then vary the degree of randomness and the cache size to determine the effect of the lazy caching strategy. To precisely measure the V-cost, we ran the tests on a DEC 2000/233 workstation (with an Alpha 21064 processor running at 233MHz) and use the atom tool [SE94] to measure the number of cycles spent in computing an optimal path. Each test is repeated 50 times (each time with a random sequence of pairs) to get the average results. The V-cost is computed by dividing the total cost of searching the paths by $50*$ sequence length. The following is the parameters used in the tests:

Algorithms The algorithms used to compute the optimal V-paths:

DP-ALL The dynamic programming approach (DP), using Algorithm 4.3.1 to precompute the variable $first$ for the all-pairs (ALL) optimal V-paths.

DP-S The dynamic programming approach (DP), using Algorithm 4.3.1 to compute the variable $first$ as needed for a single (S) pair of granularities .

EDSP-S The extended *Dag-Shortest-Paths* Algorithm (EDSP), to compute the optimal V-path as needed for a single pair (S) of granularities without caching.

EDSP-SC The extended *Dag-Shortest-Paths* Algorithm (EDSP), to compute the optimal V-path as needed for a single pair (S) of granularities with the lazy caching strategy (C).

Sequence Length The length of the call sequences with values 1, 2, 4, 8, 16, 32, 64, 128 and 256.

Cache Size The cache size for EDSP-SC with values 1KB, 2KB, 3KB, 4KB, 5KB, 6KB, 7KB and 8KB.

Degree of Randomness The number of the unique pairs of granularities needed to compute the optimal V-paths, with values 1, 2, 4, 8, 16, 32, 64, 128 and 256. For n unique pairs of granularities, the pairs are generated randomly, then pairs are randomly chosen to form a series of calls. A value of 1 specifies all pairs are identical; a values of 256 results in a totally random sequence of pairs.

Figure 6.1 shows the V-cost versus sequence length with randomly generated sequences for different algorithms. We choose the cache size 1KB for EDSP-SC to match the size of the variables ($first$, tag and $cost$) in Algorithms DP-ALL and DP-S. For DP-ALL, precomputing the $first$ variable costs 156555 cycles. Figure 6.1 shows the V-cost with the precomputation and without the precomputation. As one would expect, the V-cost for EDSP-S (16684 cycles) is much higher (8 times higher) than that for DP-ALL (2331 cycles not including the precomputation). As shown in the graph, for EDSP-SC, the 1KB cache has no effect on the cost because of the randomly generated sequences. Figure 6.2 is a log scaled version of Figure 6.1. We observe that for shorter sequences (sequence length ≤ 8), the V-costs for DP-ALL (including the precomputation) and DP-S are higher than those for EDSP-SC and EDSP-S. This makes sense — Algorithm DP-ALL and DP-S use a dynamic programming approach, thus computing the optimal V-paths for subproblems. For longer sequences, the V-costs for DP-ALL and DP-S reduce quickly below the cost for EDSP-S since the previously computed optimal V-paths are simply looked up in the $first$ variable. Note that, for longer sequences, the cost for DP-ALL drops even faster than that for DP-S.

Figure 6.3 gives the result for different degrees of randomness for the DP-S approach. The curves show that a lower degree of randomness yields a smaller V-cost. The interesting point is that the V-cost for different randomness are not dramatically different. At longer sequences, the costs are quite similar for different randomness.

Figure 6.4 and Figure 6.5 show the performance for Algorithm EDSP-SC. We wish to determinate the right cache size for different randomness. In Figure 6.4, we measure the V-cost versus sequence length with a fixed degree of randomness of 8 for different cache sizes. As shown in the figure, clearly, the 2KB cache is sufficient for the sequences with 8 unique pairs. The V-cost drops rapidly as the sequence length increases.

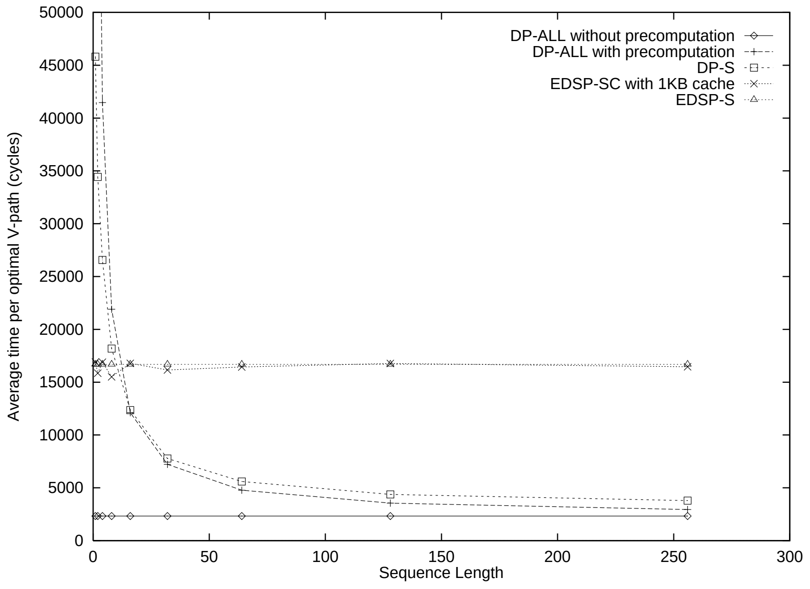


Figure 6.1: The V-cost versus Sequence Length.

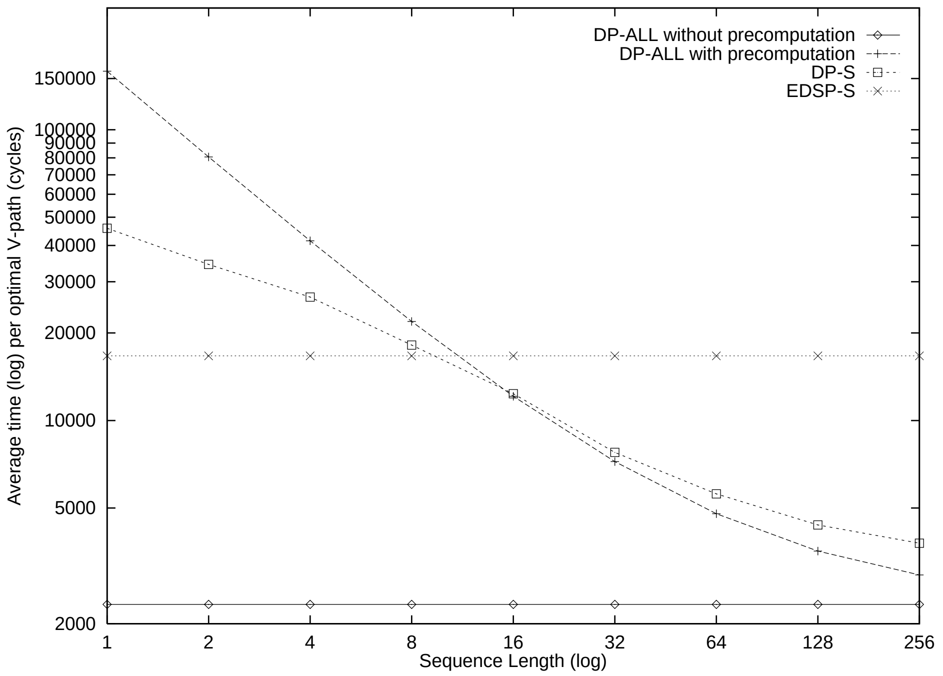


Figure 6.2: The V-cost versus Sequence Length.

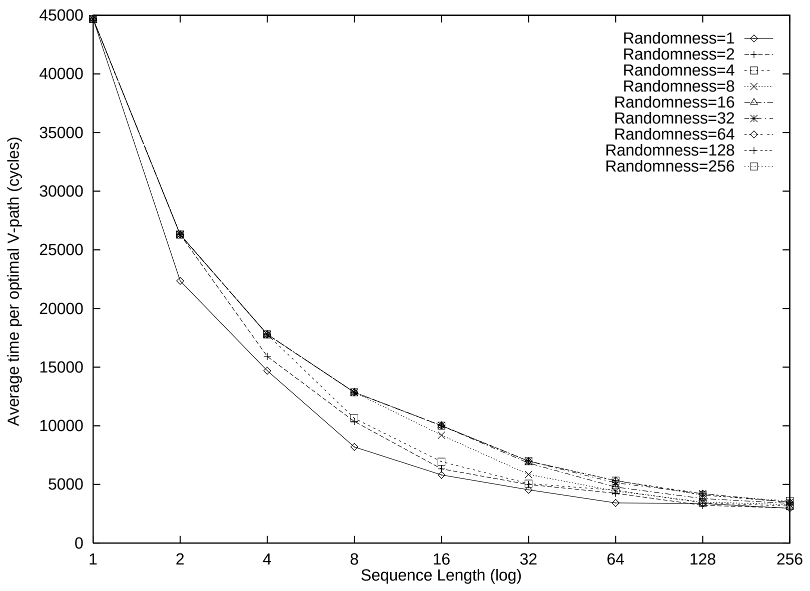


Figure 6.3: The V-cost versus Sequence Length for DP-S.

For sequences longer than 64 pairs, the V-cost drops below that (2331 cycles) for DP-ALL in Figure 6.1, which shows that for a degree of randomness of 8 and a sequence length ≥ 64, EDSP-SC provides the best performance.

To further evaluate the effect of the lazy caching strategy, we ran further tests with different randomness for EDSP-SC. Figure 6.5 shows the results. This figure indicates which cache size is sufficient for different degree of randomness. For example, 3KB is enough for Randomness=16 and 5KB for Randomness=32. Surprisingly, we find that with 3KB cache, Algorithm EDSP-SC can achieve the best performance (the cost per optimal V-path is less than 500 cycles comparing with 2231 cycles for DP-ALL) for sequences up to 16 unique pairs.

These analysis shows that different algorithms do affect the cost of the *Cast* operations. Although the tests ware done only on one granularity graph, that of Figure 2.1, the results suggest that for a user, performing the *Cast*s with higher degree of randomness and shorter sequence length, EDSP-S works fine; for lower degree of randomness (≤ 16), EDSP-SC with small cache size (about 3KB for this particular graph) provides surprisingly good results; for higher degree of randomness and longer sequence length, precomputation (DP-ALL and DP-S) are the best choices.

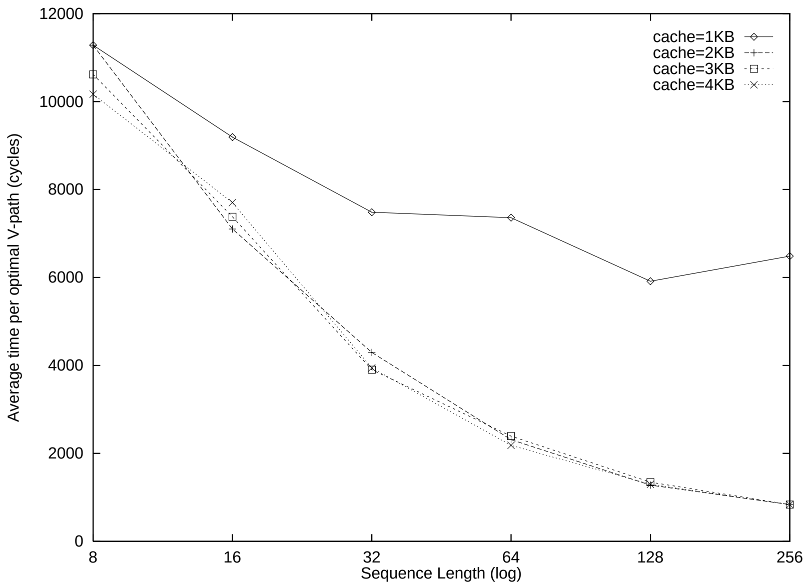


Figure 6.4: The V-cost versus Sequence Length with Randomness=8 for EDSP-SC.

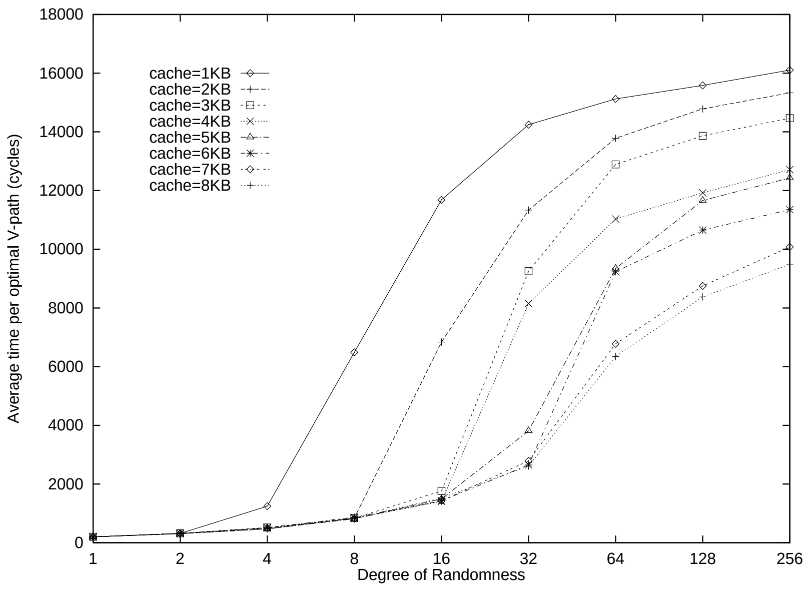


Figure 6.5: The V-cost versus Degree of Randomness with Sequence Length=256 for EDSP-SC.

Chapter 7

SUMMARY

This thesis involves elaborating a software architecture for supporting multiple time granularities within an application program or a database. First, we described a granularity as partitioning of the time-line, and the mapping relationship for a pair of granularities, which may be either a regular mapping, an irregular mapping or a congruent mapping. We showed how to weave granularities from different calendars into a single granularity graph. In Chapter Three we also pointed out the three requirements for a properly formed granularity graph. When the graph is declared, the module will check the graph and report any errors. We described why the V-paths are the correct paths to perform the *Cast* operation and proved that all V-paths yield the same result. We then extended the shortest path algorithm to compute the optimal path for a single pair of granularities and also presented an dynamic programming algorithm to solve the all-pairs optimal V-paths. We pointed out the nontermination problem in the computation of anchor offset and proved that given a granularity graph, if the anchor of each granularity is defined in a finer granularity, then the *Cast* algorithm on any pair of granularities always terminates. Chapter Five gave the correct *Cast* algorithm for casting the anchored time values. Finally, we presented our strategies to improve the performance. We emphasized that our module is to support both application programs and database management systems and we also described the different performance requirement for application programs and DBMSs. The lazy caching strategy is recommended to avoid computing previously computed results. The external module interface is listed in Appendix A. This work will serve as an important part of a comprehensive library, supporting both temporal DBMSs and application programs that handle time values.

There are some future work can be done. First, this thesis assumes that the costs for all irregular mappings are the same by assigning a cost of 1 to all edges. Such an approach has a drawback, however. It does not distinguish the edges with different execution time, which may affect the computation of the optimal path. If we know the execution costs of irregular mappings, our module interface can easily be extended to provide potentially faster V-paths. Second, this thesis does not deal with *intervals*, i.e., unanchored durations of time. An interval is an amount of time with a known length but no specific starting or ending instants [Dyreson & Snodgrass 1994A]. For example, an interval two months has a duration of two months, but can be any block of two consecutive months, which could be a wide ranges of days due to the irregular mapping between months and days. Given leap granules and various arbitrary aspects of calendars, computing the duration of an interval in a finer granularity is generally complex. The computation involves checking the every possible duration in the entire time line of the interval granularity, which is practically impossible. Therefore, how to compute intervals in the granule operations is still an open question in temporal databases.

Appendix A

THE EXTERNAL INTERFACE

```
/* The internal timestamp with maximum 96 bits.  */
typedef int polymorphic_int;

/* The mappings types */ typedef enum {
     reg_finer,
     reg_coarser,
     conguent,
     irreg_finer,
     irreg_coarser
} mapping_type;


/* The possible error types */
typedef enum {
      gran_OK,
      gran_not_found,
      gran_other_not_found,
      gran_exists,
      gran_result_not_available,
      gran_overflow,
      gran_cross_calender_operators,
      gran_too_many_granularities,
      gran_lattice_error
} gran_error_type;


/* granularity is a global notion.  Each is supplied by a calendar.  */
typedef unsigned char granularity;


/*
* Each calendar also has, known to it, local identifiers, which are
* integers, of the granularities it supports.  When any of the mapping
* or other calendar-specific functions are called, the granularities
* are always referred to by their local identifiers, which should be
* known to the calendar.
*/
/*
* There is a distinguished granularity, 0, with anchor 0, in relation
* to which all other granularities are defined.  The distinguished
* granularity is the base-line second.
*/
static const granularity second_granularity=0;

extern granularity gran_default_granularity;
```

```
/* USED FOR INITIALIZATION */
/*
* Routine:  declare_granularity
*
* Description:  Declare a granularity, and associate an anchor with it.
*
* Arguments:  id -- (IN): granularity being associated with the anchor
*             calendar_id -- (IN): identifies the calendar that
*                           supports this granularity
*             local_id -- (IN): the local identifier for the
*                        granularity, within the calendar
*             anchor -- (IN): position of anchor
*             anchor_gran -- (IN): the granularity of the anchor
*
* Return Value:  Error Code
*
* Errors:  gran_exists
*
* Side Effects:  Granularity id, if not previously declared,
*                is now defined.
*/
gran_error_type declare_granularity(granularity id, int calendar_id,
     int local_id, polymorphic_int anchor, granularity anchor_gran);


/*
* Routine:  declare_congruent
*
* Description:  Declare two granularities to be identical partitionings
*               of the underlying time line (with possibly different
*               anchors).  This relationship is reflexive, symmetric
*               and transitive.
*
* Arguments:  one -- (IN): one of the congruent granularities
*             two -- (IN): the other congruent granularity
*
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if one not found
*          gran_other_not_found:  if two not found
*
* Side Effects:  The two granularities, and any previously congruent
*                granularities, and considered to be congruent.
*/
gran_error_type declare_congruent (granularity one, granularity two);


/*
* Routine:  declare_irregular_mapping
*
* Description:  Declare an irregular mapping from one granularity to
*               another.  The mapping consists of three functions
*               that are available for converting a value in one
*               granularity to a value in a different granularity.
*               Mappings are allowed only between granularities
*               supported by the same calendar.
*
* Arguments:  from -- (IN): the source granularity
*             to -- (IN): the destination granularity
*             cast -- (IN): user-defined function that yields the
*                           closest determinate value in the destination
*                           granularity where cast has the following
*                           parameters:
```

```
*                  from -- (IN): local id of the source granularity
*                  to -- (IN): local id of the destination granularity
*                  this_lower -- (IN): value to be converted
*                  result -- (OUT): value in the to granularity
*                  and returns an error code (gran_not_found or
*                                   gran_other_not_found)
*              scale_determinate -- (IN): user-defined function that
*                      yields a possibly indeterminate value (in the
*                      destination granularity) of a determinate value
*                      in the source granularity where scale_determinate
*                      has the following parameters.
*                  from -- (IN): local id of the source granularity
*                  to -- (IN): local id of the destination granularity
*                  this_lower -- (IN): value to be converted
*                  result_lower -- (OUT): lower support in the to
*                                     granularity
*                  result_upper -- (OUT): upper support in the to
*                                     granularity
*                  and returns an error code (gran_not_found or
*                                     gran_other_not_found)
*              scale_indeterminate -- (IN): function that yields a
*                      possibly indeterminate value (in the destination
*                      granularity) of an indeterminate value in the source
*                      granularity where scale_indeterminate has the
*                      following parameters
*                    from -- (IN): local id of the source granularity
*                    to -- (IN): local id of the destination granularity
*                    this_lower -- (IN): value to be converted
*                    this_upper -- (IN): value to be converted
*                    result_lower -- (OUT): lower support in the to
*                                         granularity
*                    result_upper -- (OUT): upper in the to granularity
*                    and returns an error code (gran_not_found
*                                         or gran_other_not_found)
*
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if from not found
*          gran_other_not_found:  if to not found
*
* Side Effects:  Mapping is recorded for use in conversion operations.
*/
gran_error_type declare_irregular_mapping(granularity from,
     granularity to,
     gran_error_type (*cast_finer)(),
     gran_error_type (*cast_coarser)(),
     gran_error_type (*scale_determinate)(),
     gran_error_type (*scale_indeterminate)());


/* * Routine:  declare_regular_mapping
*
* Description:  Declare an irregular mapping from one granularity to
*                 another.  A regular mapping is simply an integral
*                 to conversion.  An example is from hours to minutes
*                 by multiplying by 60.  An anchor adjustment may also
*                 be required.  Regular mappings can be defined between
*                 granularities supported by different calendars.
*
* Arguments:  from -- (IN): the source (coarser) granularity
*               to -- (IN): the target (finer) granularity
*               conversion -- (IN): the multiplicative conversion factor
*
* Return Value:  Error Code
```

```
*
* Errors:  gran_not_found:  if from not found
*           gran_other_not_found:  if to not found
*
* Side Effects:  Mapping is recorded for use in conversion operations.
*/
gran_error_type declare_regular_mapping(granularity from,
     granularity to, int conversion);


/*
* Routine:  declare_e_plus_s
*
* Description:  Declare a function to do this arithmetic operation
*                 in the indicated span granularity.
*
* Arguments:  s_gran -- (IN): granularity of the event
*               e_plus_s -- (IN): function which can perform the operation
*                       where e_plus_s has the following parameters
*                   e -- (IN): source event
*                   e_gran -- (IN): local granularity of the event
*                   s -- (IN): source span
*                   s_gran -- (IN): local granularity of the span
*                   result -- (OUT): the event resulting from the,
*                           addition in the e_gran
*                   and returns an error code (gran_not_found or
*                           gran_other_not_found)
*
* Return Value:  Error Code
*
* Errors:  gran_not_found
*
* Side Effects:  The function pointer is stored, or overwritten if
*                  previously present, for later use.
*
*
*
*gran_error_type declare_e_plus_s(granularity s_gran,
*     gran_error_type (*e_plus_s)(polymorphic_int e,
*     granularity e_gran, polymorphic_int s,
*     polymorphic_int* result));
*/
gran_error_type declare_e_plus_s(granularity s_gran,
     gran_error_type (*e_plus_s)());


/*
* Routine:  declare_s_plus_e
*
* Description:  Declare a function to do this arithmetic operation in
*                 the indicated span granularity.
*
* Arguments:  s_gran -- (IN): granularity of the span
*               s_plus_e -- (IN): function which can perform the operation
*                         where s_plus_e has the following parameters
*                   s -- (IN): source span
*                   e -- (IN): source event
*                   e_gran -- (IN): local granularity of the span
*                   result -- (OUT): the event resulting from the
*                             addition in the s_gran
*                   and returns an error code (gran_not_found)
*
* Return Value:  Error Code
*
```

```
* Errors:  gran_not_found
*
* Side Effects:  The function pointer is stored, or overwritten if
*                 previously present, for later user
*
*gran_error_type declare_s_plus_e(granularity s_gran,
*     gran_error_type (*s_plus_e)(polymorphic_int s,
*     polymorphic_int e, granularity e_gran,
*     polymorphic_int *result));
*/
gran_error_type declare_s_plus_e(granularity s_gran,
     gran_error_type (*s_plus_e)());


/*
* Routine:  declare_e_minus_e
*
* Description:  Declare a function to do this arithmetic operation with
*                the indicated result granularity.
*
* Arguments:  result_gran -- (IN): desired granularity of the result
*              e_minus_e -- (IN): function which can perform the operation
*                     where e_minus_e has the following parameters:
*                   left -- (IN): source event
*                   left_gran -- (IN): local granularity of left
*                   right -- (IN): other source event
*                   right_gran -- (IN): local granularity of right
*                   result -- (OUT): the span resulting from the
*                              subtraction, in the result_gran
*                   and returns an error code (gran_not_found or
*                              gran_other_not_found)
*
* Return Value:  Error Code
*
* Errors:  gran_not_found
*
* Side Effects:  The function pointer is stored, or overwritten if
*                 previously present, for later use.
*
*gran_error_type declare_e_minus_e(granularity result_gran,
*     gran_error_type (*e_minus_e)(polymorphic_int left,
*     granularity left_gran,
*     polymorphic_int right,
*     granularity right_gran,
*     int *result));
*/
gran_error_type declare_e_minus_e(granularity result_gran,
     gran_error_type (*e_minus_e)());


/*
* Routine:  declare_extract
*
* Description:  Declare an extraction function requesting a specified
*                granularity
*
* Arguments:  right_gran -- (IN): the specified granularity
*              extract -- (IN): a function which performs the requested
*                          operation.
*                     where extract has the following parameters:
*                   right_gran--(IN): the specified granularity
*                   right -- (IN): an event
*                   left_gran -- (IN): the requested granularity
*                   result -- (OUT): the value of the requested
*                              granularity
*                  and returns an error code (gran_not_found, gran_overflow)
*
* Return Value:  Error Code
*
* Errors:  gran_not_found
*
* Side Effects:  None.
*
*gran_error_type declare_extract(granularity right_gran,
*      gran_error_type (*extract)(granularity right_gran,
*      polymorphic_int right,
*      granularity left_gran,
*      int *result) );
*/
gran_error_type declare_extract(granularity right_gran,
     gran_error_type (*extract)());


/*
* Routine:  extract
*
* Description:  Extracts the indicated component of the argument event
*                 or span.  As an example, extracting the year from the
*                 event "April 19, 1955" yields 1955.  Extracting the
*                 month yields 4, and extracting the day yields 19.
*                 Of course, the specific values returned are
*                 calendar-dependent.
*
* Arguments:  left_gran -- (IN): the granularity to be extracted
*              right -- (IN): an event or span, from which the value
*                             is to be extracted
*              right_gran -- (IN): the granularity of right
*              result -- (OUT)
*
* Return Value:  Error Code
*
* Errors:  gran_not_found
*            gran_other_not_found
*            gran_cross_calender_operators
*            gran_overflow -- the integer returned is too big to
*                          fit in an int
*
* Side Effects:  A calendar_supplied function is called.
*/
gran_error_type extract(granularity left_gran, polymorphic_int right,
     granularity right_gran, int *result);


/*
* Routine:  scale_determinate
*
* Description:  Yields a possibly indeterminate value (in the
*                 destination granularity) of a determinate value
*                 in the source granularity
*
* Arguments:  original -- (IN): value to be converted
*              from -- (IN): the source granularity
*              to -- (IN): the target granularity
*              result_lower -- (OUT): lower support in the to
*                             granularity
*              result_upper -- (OUT): upper support in the to
*                             granularity
*
```

```
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if from not found
*           gran_other_not_found:  if to not found
*
* Side Effects:  One or more calendar_supplied functions may be called.
*/
gran_error_type scale_determinate(polymorphic_int original,
     granularity from, granularity to,
     polymorphic_int* result_lower,
     polymorphic_int* result_upper);


/*
* Routine:  scale_indeterminate
*
* Description:  Yields an indeterminate value (in the destination
*                 granularity) of a determinate value in the source
*                 granularity.
*
* Arguments:  this_lower -- (IN): lower support of value to be converted
*              this_upper -- (IN): upper support of value to be converted
*              from -- (IN): the source granularity
*              to -- (IN): the target granularity
*              result_lower -- (OUT): lower support in the to
*                           granularity
*              result_upper -- (OUT): upper support in the to
*                           granularity
*
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if from not found
*           gran_other_not_found:  if to not found
*
* Side Effects:  One or more calendar_supplied functions may be called.
*/
gran_error_type scale_indeterminate(polymorphic_int this_lower,
     polymorphic_int this_upper, granularity from,
     granularity to, polymorphic_int *result_lower,
     polymorphic_int* result_upper);


/*
* Routine:  cast
*
* Description:  Yields the first determinate value in the destination
*                 granularity of a value, either determinate, or lower
*                 support if indeterminate
*
* Arguments:  this_lower -- (IN): input event
*              from -- (IN): source granularity
*              to -- (IN): target granularity
*              result -- (IN): *
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if from not found
*           gran_other_not_found:  if to not found
*
* Side Effects:  A calendar_supplied function is called.
*/
gran_error_type cast(polymorphic_int this_lower, granularity from,
     granularity to, polymorphic_int* result);
```

```
/*
* Routine:  e_plus_s
*
* Description:  Adds an event and a span, of mixed granularities,
*                 yielding an event.
*
* Arguments:  e_gran-- (IN): granularity of the event
*             e -- (IN): the source event
*             s_gran -- (IN): granularity of the span
*             s -- (IN): the source span
*             result -- (OUT): The result is in e_gran
*
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if e_gran not found
*            gran_other_not_found:  if s_gran not found
*            gran_cross_calender_operators
*
* Side Effects:  A calendar_supplied function is called.
*/
gran_error_type e_plus_s(granularity e_gran, polymorphic_int e,
     granularity s_gran, polymorphic_int s,
     polymorphic_int* result);


/*
* Routine:  s_plus_e
*
* Description:  Adds an event and a span, of mixed granularities,
* 3.2cmyielding an event.
*
* Arguments:  s_gran -- (IN): the granularity of the span
*             s -- (IN): the source span
*             e_gran -- (IN): the granularity of the event
*             e -- (IN): the source event
*             result -- (OUT): in the s_gran granularity
*
* Return Value:  Error Code
*
* Errors:  gran_not_found:  if s_gran not found
*            gran_other_not_found:  if e_gran not found
*            gran_cross_calender_operators
*
* Side Effects:  A calendar_supplied function is called.
*/
gran_error_type s_plus_e(granularity s_gran, polymorphic_int s,
     granularity e_gran, polymorphic_int e,
     polymorphic_int* result);


/*
* Routine:  e_minus_e
*
* Description:  Subtracts two events, in mixed granularities,
*     yielding a span, in a specified granularity.
*
* Arguments:  left_gran -- (IN): granularity of left
*             left -- (IN): one of the source events
*             right_gran -- (IN): granularity of right
*             right -- (IN): the other of the source events
*             result_gran -- (IN): the desired resulting granularity
*             result -- (OUT): *
* Return Value:  Error Code
*
```

```
* Errors:  gran_not_found:  if from not found
*            gran_other_not_found:  if to not found
*            gran_cross_calender_operators
*
* Side Effects:  A calendar_supplied function is called.
*/
gran_error_type e_minus_e(granularity left_gran,
     polymorphic_int left, granularity right_gran,
     polymorphic_int right, granularity result_gran,
     polymorphic_int* result);
```

Appendix B

THE INTERNAL DATA STRUCTURE

```
/* The linked-list structure for regular mappings.  It contains the
   * destination granularity, the conversion constant and a pointer
   * referencing the next finer mapping.
   */
   typedef struct reg_map_list_struct {
         int conversion;
         granularity destination;
         struct reg_map_list_struct *next_ptr;
   } reg_map_list_type;


   /* The linked-list structure for irregular mappings.  It contains the
   * destination granulairty, the irregular mapping funtions and a
   * pointer referencing the next irregular mapping.
   */
   typedef struct irreg_map_list_struct {
         granularity destination ;
         gran_error_type (*cast)(); /*user-defined function*/
         gran_error_type (*scale_det)(); /*user-defined function*/
         gran_error_type (*scale_indet)(); /*user-defined function*/
         struct irreg_map_list_struct *next_ptr;
   } irreg_map_list_type;


   /* The structure for the granularity graph.  It contains the
   * user_id given by the user, the internal id:  local_id,
   * the calender_id which the granularity belongs to, the
   * anchor and the anchor granularity.  It also contains pointers
   * referencing the regular mapping list, the irregular mapping
   * list and thecongruent mapping list.
   */
   typedef struct gran_lattice_struct {
         granularity user_id;
         int calender_id;
         int local_id;
         polymorphic_int anchor;
         granularity anchor_gran;
         reg_map_list_type *reg_map_coarser_list;
         reg_map_list_type *reg_map_finer_list;
         reg_map_list_type *cong_map_list;
         irreg_map_list_type *irreg_map_coarser_list;
         irreg_map_list_type *irreg_map_finer_list;
   } gran_lattice_type;


   /* The structure for the optimal path.  It contains the source
   * granularity, the destination granularity, the mapping type,
* the conversion constant for the regular mapping and mapping
* functions for irregular mappings.  It also has a pointer
* referencing the next step in the path.
*/
typedef struct super_path_struct {
      granularity from;
      granularity to;
      double conversion;
      mapping_type map_type; /*mapping type*/
      gran_error_type (*cast)();
      gran_error_type (*scale_det)();
      gran_error_type (*scale_indet)();
      struct super_path_struct *next_ptr;
} super_path_type;
typedef super_path_type *super_path_type_ptr;
```

Bibliography

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

[Cattell 1994] Cattell, R. G. G., “The Object Database Standard: ODMG-93” San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1994.

[Clifford & Rao 1987] Clifford, J. and A. Rao., “A Simple, General Structure for Temporal Domains,” in *Proceedings of the Conference on Temporal Aspects in Information Systems*. AFCET. France: May 1987, pp. 247–265.

[Cormen et al. 1990] Cormen, T. H. and R. Rivest, “Introduction To Algorithms” London, England: The MIT Press, 1990.

[Dershowitz et al. 1990] Dershowitz N. and E. Reingold, “Calendrical calculations,” in *Software—Practice and Experience*, 20, No. 9(1990), pp. 899–928.

[Dyreson & Snodgrass 1993] Dyreson, C. E. and R. Snodgrass, “Valid-time Indeterminancy,” in *Proceedings of the International Conference on Data Engineering*. 1993, pp. 335–343.

[Dyreson & Snodgrass 1994A] Dyreson, C. E. and R. Snodgrass, “Temporal Granularity and Indeterminacy: Two Sides of the Same Coin.” Technical Report TR94-06. Computer Science Department, University of Arizona. Feb., 1994.

[Dyreson & Snodgrass 1994B] Dyreson, C. E. and R. Snodgrass, “Efficient Timestamp Input and Output,” in *Software-Practice and Experience*, 24, No. 1(1994), pp. 80–109.

[Kraus et al. 1996] Kraus, S., Y. Sagiv and V. Subrahmanian, “Representing and Integrating Multiple Calendars.” Submitted for publication.

[Melton & Simon 1993] Melton, J. and A. R. Simon. “Understanding the New SQL: A Complete Guide” San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.

[Reingold et al. 1993] Reingold E. M., N. Dershowitz and S. Clamen, “Calendrical Calculations, II: Three Historical Calendars,” in *Software—Practice and Experience*, 23, No. 4(1993), pp. 383–404.

[Snodgrass 1995] Snodgrass, R., “The TSQL2 Temporal Query Language” Dordrecht, The Netherlands: Kluwer Academic Publishers, 1995.

[SE94] Srivastava, A. and A. Eustace, ATOM: A System for Build Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN ’94 Conference on programming Language Design and Implementation*, pp. 196–205, June 1994

[Vinogradov et al. 1988] Vinogradov, I. M., “Encyclopaedia Of Mathematics” Dordrecht, The Netherlands: Kluwer Academic Publishers, 1988, V2, pp. 255.

[Wang et al. 1993] Wang, X., S. Jajodia and V. Subrahmanian, “Temporal Modules: An Approach Toward Temporal Databases,” in *Proceedings of the ACM SIGMOD International Conference* on Manegement of Data. 1993, pp. 227–236.

[Wiederhold et al. 1991] Wiederhold, G., S. Jajodia and W. Litwin., “Dealing with Granularity of Time in Temporal Databases,” in *Proceedings Nordic Conference on Advanced Information Systems Engineering*. Trondheim, Norway: May 1991.