VALID-TIME INDETERMINACY

by

Curtis E. Dyreson

Copyright © Curtis E. Dyreson 1994

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 4

VALID-TIME INDETERMINACY

Curtis E. Dyreson, Ph.D.
The University of Arizona, 1994

Director: Richard T. Snodgrass

In *valid-time indeterminacy*, it is known that an event stored in a temporal database did in fact occur, but it is not known exactly *when* the event occurred. We extend a tuple-timestamped temporal data model to support valid-time indeterminacy and outline its implementation. This work is novel in that previous research, although quite extensive, has not studied this particular kind of incomplete information. To model the occurrence time of an event, we introduce a new data type called an *indeterminate instant*. Our thesis is that by representing an indeterminate instant with a set of contiguous chronons and a probability distribution over that set, it is possible to characterize a large number of (possibly weighted) alternatives, to devise intuitive query language constructs, including schema specification, temporal constants, temporal predicates and constructors, and aggregates, and to implement these constructs efficiently. We extend the TQuel and TSQL2 query languages with constructs to retrieve information in the presence of indeterminacy. Although the extended data model and query language provide needed modeling capabilities, these extensions appear to carry a significant execution cost. The cost of support for indeterminacy is empirically measured, and is shown to be modest. We then show how indeterminacy can provide a much richer modeling of *granularity* and *now*. Granularity is the unit of measure of a temporal datum (e.g., days, months, weeks). Indeterminacy and granularity are two sides of the same coin insofar as a time at a given granularity

is indeterminate at all finer granularities. *Now* is a distinguished temporal value. We describe a new kind of instant, a *now-relative indeterminate instant*, which has the same storage requirements as other instants, but can be used to model situations such as that an employee is currently employed but will not work beyond the year 1995. In summary, support for indeterminacy dramatically increases the modeling capabilities of a temporal database without adversely impacting performance.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of the manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: ___________________________

ACKNOWLEDGMENTS

This work was supported in part by NSF grants ISI-8902707 and ISI-9302244, and IBM contract #1124. I wish to thank my parents for all their love and encouragement throughout my life, I could not have written this dissertation without them. I am also very much indebted to my advisor and mentor, Richard Snodgrass. Quite simply, Rick was a great advisor; he gave freely of his time, patiently encouraged me to think rigorously, and had an inexhaustible supply of energy and enthusiasm. I also wish to thank Peter J. Downey for insightful comments and contributions.

TABLE OF CONTENTS

LIST OF FIGURES 9

LIST OF TABLES 12

ABSTRACT 13

CHAPTER 1: INTRODUCTION 14

CHAPTER 2: A TAXONOMY OF INCOMPLETE INFORMATION 22
- 2.1 Definitions 22
- 2.2 Attribute Incompleteness 26
  - 2.2.1 Common Incomplete Attribute Values 26
  - 2.2.2 Truncation Incompleteness 32
  - 2.2.3 Complete Information Null Values 32
- 2.3 A Taxonomy of Incomplete Information Data Models 33
  - 2.3.1 Value vs. Temporal Incompleteness 34
  - 2.3.2 Alternative vs. Possible 34
  - 2.3.3 Unweighted or Weighted 35
  - 2.3.4 The Taxonomy Tree 35
  - 2.3.5 Valid-time Indeterminacy 37
- 2.4 Indeterminacy and Related Work 39
- 2.5 Summary 44

CHAPTER 3: TIME MODEL 45
- 3.1 The Ontology of Time 45
- 3.2 A Discrete View of Time 46
- 3.3 Modeling Instants 47

3.4 Modeling Intervals 48
3.5 Modeling Spans 49
3.6 Impact of the Model on the Semantics of Timestamp Operations 49
3.7 Summary 49

CHAPTER 4: EXTENSIONS TO TQUEL 51
4.1 Extending the Data Model with Indeterminacy 51
4.1.1 Indeterminate Instants 51
4.1.2 Indeterminate Intervals 54
4.1.3 Indeterminate Spans 55
4.1.4 Indeterminate Tuples 55
4.1.5 Other Kinds of Indeterminacy 57
4.2 Review of TQuel 57
4.3 Syntactic Extensions to Support Valid-time Indeterminacy 60
4.4 Semantic Extensions to TQuel 63
4.4.1 Supporting Ordering Plausibility 63
4.4.2 Supporting Range Credibility 69
4.4.3 Coalescing 71
4.4.4 Semantics of the Example Query 72
4.4.5 Query Reducibility 73

CHAPTER 5: IMPLEMENTATION OF INDETERMINACY 76
5.1 Indeterminate Timestamp Formats 77
5.1.1 Instant Timestamp Formats 77
5.1.2 Indeterminate Intervals and Spans 82
5.1.3 Design Decisions 83
5.2 Implementation of Operators 84
5.2.1 The *Before* Operation 84
5.2.2 The *Shrink* functions 94
5.2.3 *Set_Before* 97
5.2.4 *Reduce′* 98

5.2.5 Impact of Indeterminacy on the Determinate Implementation . . 98
5.3 Empirical Analysis of the Implementation 99

CHAPTER 6: INDETERMINACY IN TSQL2 107
6.1 Syntactic Extensions to Support Temporal Indeterminacy 107
6.2 Semantic Extensions to TSQL2 110
6.2.1 Semantics of Arithmetic Operations 110
6.2.2 Semantics of Aggregate Operations 112
6.2.3 Input and Output of Indeterminate Temporal Constants 113
6.3 Summary 113

CHAPTER 7: INDETERMINACY AND GRANULARITY 114
7.1 Motivation 115
7.2 A Discrete Image of Time 116
7.2.1 Modeling Instants at Various Granularities 120
7.2.2 Indeterminacy and Granularity 120
7.3 A Proposal for Accommodating Mixed Granularities 121
7.3.1 Building the Lattice 121
7.3.2 Temporal constants 124
7.3.3 Column Definitions 125
7.3.4 Granularity in Operations 127
7.3.5 Scale 128
7.3.6 Scaling Intervals and Spans 132
7.3.7 Scaling Mass Functions 133
7.3.8 Cast 134
7.3.9 Enforcing SQL-92 Semantics 135
7.3.10 Defaults 136
7.3.11 Processing of Example Query 136
7.4 Implementation 137
7.4.1 Impact of Granularity on Timestamp Formats 138
7.4.2 Scale and Cast 139

7.4.3 Empirical Results . . . 141
7.5 Related Work on Granularity . . . 142
7.6 Chapter Summary . . . 144

CHAPTER 8: *NOW* AND INDETERMINACY . . . 146
8.1 *Now* in Valid-time . . . 146
8.2 *Now* Remodeled . . . 150
8.3 Timestamp Representation . . . 153
8.4 Summary . . . 155

CHAPTER 9: CONCLUSIONS . . . 156

APPENDIX A: TSQL2 BNF FOR INDETERMINACY . . . 162
A.1 New or Modified Syntax for Indeterminacy Constructs . . . 162
A.1.1 Section 5.2 <token> . . . 162
A.1.2 Section 5.3 <literal> . . . 162
A.1.3 Section 6.1 <data type> . . . 163
A.1.4 Section 6.3 <table reference> . . . 165
A.1.5 Section 7.6 <where clause> . . . 165
A.1.6 Section 11 <schema definition> . . . 166
A.1.7 Section 12.5 <SQL procedure statement> . . . 166
A.2 New or Modified Syntax for Granularity . . . 167
A.2.1 Section 5.3 <literal> . . . 168
A.2.2 Section 6.8 ⟨*datetime value function*⟩ . . . 168
A.2.3 Section 6.10 ⟨*cast specification*⟩ . . . 168
A.2.4 Section 6.14 <datetime value expression> . . . 169
A.2.5 Section 6.15 <interval value expression> . . . 170
A.2.6 Section 10.1 <interval qualifier> . . . 171
A.2.7 Section 11.10 <alter table statement> . . . 172
A.2.8 Section 12.5 <SQL procedure statement> . . . 172

LIST OF FIGURES

1.1 A valid-time database 18
1.2 Answers to example queries 19

2.1 The taxonomy subtree for value/alternative incompleteness 36
2.2 The taxonomy subtree for value/possible incompleteness 37
2.3 The taxonomy subtree for temporal/alternative incompleteness 38
2.4 The taxonomy subtree for temporal/possible incompleteness 38
2.5 The situation modeled by schools in value incompleteness 39
2.6 The situation modeled by schools in temporal incompleteness 40

4.1 A "probably early" distribution 53
4.2 Examples of value incompleteness 57
4.3 An example `retrieve` statement 58
4.4 An example query 62
4.5 Result of the example query 63
4.6 Table of **Pr**$[\alpha < \beta]$ for the indeterminate instants in *Received* 65
4.7 A pictorial representation of the instants in *Received* 65
4.8 Ordering the events in *Received* depends on γ 66
4.9 *Shrink_s*$(\delta, (|1 \sim 20|, Uniform))$ for several values of δ 69
4.10 Demonstrating Query Reducibility 74

5.1 The standard instant format 78
5.2 The indeterminate formats 80
5.3 Interface to *Before* 85
5.4 The approximated uniform mass function with $P = 3$ and $C = 8$ 87
5.5 The tree storing the approximated uniform mass function with $P = 3$ and $C = 8$ 88

5.6 The pruned tree storing the approximated uniform mass function with $P = 3$ and $C = 8$ 89
5.7 A rod counting operation 92
5.8 The pivoting algorithm 95
5.9 The rods within the dotted lines are the undercount for the pivot 96
5.10 The *Shrink_s* algorithm 97
5.11 Worst-case performance of *Before* 101
5.12 Sliding one instant relative to another performance of *Before* 102
5.13 The cost of comparing ten instants randomly placed in a chronon space of varying size 103
5.14 The average cost of comparing ten instants randomly placed in a chronon space of varying size 104
5.15 The average cost of the example query per combination of tuples 105
5.16 MULTICAL calls for example query 106

6.1 An example query 110

7.1 A flight database 116
7.2 The time-line at a granularity of days 119
7.3 A Gregorian calendar granularity lattice 119
7.4 The Gregorian calendar granularity lattice with mappings between granularities 123
7.5 Span A can be placed within a single granule or spanning two granules . . 133
7.6 Flight times scaled to days 137
7.7 MULTICAL calls for example queries 143

8.1 Jane's employment tuple 147
8.2 Jane's employment tuple with a large right boundary 148
8.3 Meaning of Jane's tuple, if today is July 9 and the bound is 3 days 149
8.4 Jane has a fixed-term appointment, represented with valid-time indeterminacy 150

8.5 Jane's employment will be for at least two months 150
8.6 Assuming a mandatory retirement 150
8.7 No guaranteed minimum term 151
8.8 The situation as of July 9 151
8.9 The situation as of July 10 152
8.10 Using a now-relative indeterminate instant 153
8.11 The now-relative indeterminate instant format 154

LIST OF TABLES

5.1 Encodings in the indeterminate formats 79
5.2 Timings on indeterminate operations 100
6.1 The mass function that results from an arithmetic operation 111
7.1 Sizes of some common instant timestamps 140

ABSTRACT

In *valid-time indeterminacy*, it is known that an event stored in a temporal database did in fact occur, but it is not known exactly *when* the event occurred. We extend a tuple-timestamped temporal data model to support valid-time indeterminacy and outline its implementation. This work is novel in that previous research, although quite extensive, has not studied this particular kind of incomplete information. To model the occurrence time of an event, we introduce a new data type called an *indeterminate instant*. Our thesis is that by representing an indeterminate instant with a set of contiguous chronons and a probability distribution over that set, it is possible to characterize a large number of (possibly weighted) alternatives, to devise intuitive query language constructs, including schema specification, temporal constants, temporal predicates and constructors, and aggregates, and to implement these constructs efficiently. We extend the TQuel and TSQL2 query languages with constructs to retrieve information in the presence of indeterminacy. Although the extended data model and query language provide needed modeling capabilities, these extensions appear to carry a significant execution cost. The cost of support for indeterminacy is empirically measured, and is shown to be modest. We then show how indeterminacy can provide a much richer modeling of *granularity* and *now*. Granularity is the unit of measure of a temporal datum (e.g., days, months, weeks). Indeterminacy and granularity are two sides of the same coin insofar as a time at a given granularity is indeterminate at all finer granularities. *Now* is a distinguished temporal value. We describe a new kind of instant, a *now-relative indeterminate instant*, which has the same storage requirements as other instants, but can be used to model situations such as that an employee is currently employed but will not work beyond the year 1995. In summary, support for indeterminacy dramatically increases the modeling capabilities of a temporal database without adversely impacting performance.

CHAPTER 1

INTRODUCTION

The relational database model [Codd 1970] has proved a fruitful paradigm for database research and commercial applications. The relational model blends conceptual simplicity with a solid theoretical foundation, and has an efficient implementation [Date 1990]. But, as originally formulated, the relational model could not represent or query *incomplete* information. In the last two decades, the relational model has been extended to provide support for many kinds of incomplete information, including fuzzy, imprecise, indefinite, indeterminate, missing, partial, possible, probable, uncertain, unknown, vague, and disjunctive information. (Related work is presented in Chapter 2.) This interest in supporting incomplete information in relational databases is fueled by the realization that incomplete information is pervasive in large bodies of data.

Temporal information is also ubiquitous; time is an attribute of most "real-world" data. For example, a personnel relation is commonly a history of employee promotions and demotions, a building relation records different blueprint versions, and an inventory relation charts sales over time. Despite the prevalence of temporal information, there is no special support for it in the relational model. Temporal database researchers have added special support for temporal information; extensions to the relational model to support time are manifold [Kline 1993, Soo 1991]. This support is divided along two orthogonal temporal dimensions: valid time and transaction time [Jensen et al. 1994, Snodgrass & Ahn 1986]. Valid time is the "real-world" time associated with a fact while transaction time is the time when the fact was stored in the database. The two time dimensions can be used to partition databases into four categories: *snapshot* databases which have no support for either time dimension, *transaction-time* databases which support only transaction time, *valid-time* databases which support only valid time, and *bitemporal*

databases which support both time dimensions. This taxonomy of temporal databases is sufficient; in the six years since it was formulated, no new taxonomies have supplanted it.

A valid-time database records the history of an enterprise [Jensen et al. 1994]. The history is a sequence of events. The term “history” is intuitive, but somewhat misleading, since the events could be in the future. It associates with event a *timestamp* indicating when that event occurred. Often a user knows only approximately when an event happened. For instance, she may know that it happened “between 2 PM and 4 PM,” “sometime last week,” or “around the middle of the month.” These are examples of *valid-time indeterminacy*. Information that is valid-time indeterminate can be characterized as “don’t know when” information, or more precisely, “don’t know *exactly* when” information. This kind of information has various sources, including the following.

- *Granularity mismatch* — In perhaps most cases, the granularity with which data is recorded is finer than the precision to which the occurrence time of an event is known. For example, an occurrence time known to within one day recorded on a system with timestamps in the granularity of a microsecond, happened sometime *during* that day, but during which microsecond is unknown.

- *Dating techniques* — Many dating techniques are inherently imprecise, such as Carbon-14 dating.

- *Uncertainty in planning* — Projected completion dates are often inexactly specified, e.g., the project will complete three to six months from now.

- *Unknown or imprecise event times* — In general, occurrence times could be unknown or imprecise. For example, perhaps we do not know when an individual was born. The individual’s date of birth could be recorded in the database as either unknown (they were born between the beginning and the end of time) or imprecise (they were born between now and 70 years ago).

- *Clock measurements* — Every clock measurement has some imprecision [Petley 1991].

Temporal database management systems should provide support for valid-time indeterminacy. In particular, timestamps should include a representation for valid-time indeterminacy, users should be able to control, via query language constructs, the amount of indeterminacy present in derived information, and the query evaluator should accommodate valid-time indeterminacy in its processing. Query evaluation efficiency should remain high in the presence of valid-time indeterminacy, and it should be unaffected by its absence.

This dissertation presents a *valid-time indeterminate* data model and its implementation. The model adds valid-time indeterminacy to TQuel [Snodgrass 1987]. TQuel is a strict superset of Quel, the query language for Ingres [Stonebraker et al. 1976]. TQuel has a complete, formal semantics which we extend to support valid-time indeterminacy. We also extend TSQL2 [Snodgrass et al. 1994], which is a consensus temporal extension of SQL-92 [Melton & Simon 1993], to support temporal indeterminacy. The addition of indeterminacy to two, distinct temporal query languages demonstrates that the techniques we describe are not limited to a particular query language, but instead, are generally applicable. Below we give a short example requiring the storage of temporally indeterminate information.

A valid-time database is shown in Figure 1.1. This database models a single company with two warehouses and one airplane factory. The warehouses supply parts to the factory. Each warehouse keeps a *Sent* relation, which records when parts were shipped from the warehouse to the factory. The factory maintains the *In_Production* relation, which is a production history of airplanes built by the factory. For every relation we assume an underlying timestamp granularity of one day.

Valid-time indeterminacy naturally arises in both base relations and derived relations. It may surprise the reader to note that the *In_Production* base relation is a valid-time indeterminate relation. This is because the granularity of the *In_Production* relation is a month while that of the *Sent* and *Received* relations are just a single day. A month is an indeterminate value that represents a set of possible days. We know that production on an airplane started on some day in the indicated month, but we cannot be sure which one. For

this example, we assume that production is equally likely to have started or ended during any day in an indicated month, although, in general we allow nonuniform likelihoods.

The *Received* relation is not maintained by either the factory or a warehouse; rather it is a derived relation, the product of inference. Parts are shipped by truck from a warehouse and arrive at the factory no earlier than four and no later than twenty-four days after they leave a warehouse. The *Received* relation is computed from each warehouse's *Sent* relation by adding a 4–24 day "fudge factor" to the valid-time attribute. The valid times in the *Received* relation are indeterminate; that is, we know roughly when the parts were received, but we do not know exactly which day they were received. We will assume that each day in the recorded range of days is equally likely. For example, the batch of engines received from the Trump warehouse arrived on one of the days in the set {June 8, June 9, ..., June 27}, but we have no reason to favor one day over another.

In a database that supports valid-time indeterminacy, queries can make use of indeterminate information. Suppose that a few of the Centurion airplane owners report a faulty wing strut. Naturally, we would like to query the database to determine which warehouse(s) supplied the defective parts and, specifically, which lots are implicated (we give such a query in Section 4.3). In TQuel (or TSQL2) with valid-time indeterminacy, we could query to determine which shipment of wing struts "overlaps" the production of a Centurion airplane. Overlap is the operation of temporal intersection.

There are two well-defined limits on an answer to a query in an incomplete information database: the *definite* answer and the *possible* answer [Lipski 1979]. Very roughly, the definite answer is the information that satisfies the query in *all* possible extensions of the database while the possible answer is the information that satisfies the query in *some* possible extension of the database. For example, consider a temporal selection on the *Received* relation that selects those tuples received prior to June 10. Even though the exact date the shipment of `Lot_Num 23` from the Trump warehouse arrived is unknown, it is clear that this shipment arrived before June 10 (the shipment arrived on some day in the set {May 10, May 11, ..., May 29}). This tuple, and no other, is in the definite answer to the query. `Lot_Num 30` from the Griffin warehouse is in the possible answer to the query. It is possible that this shipment arrived prior to June 10 (and also possible that

Sent_by_Trump(Lot_Num,Part)

Lot_Num	Part	Valid time (at)
23	wing strut	May 6
24	engine	June 4

Sent_by_Griffin(Lot_Num,Part)

Lot_Num	Part	Valid time (at)
30	wing strut	May 26
31	wing strut	June 9

In_Production(Model, Serial_Num)

Model	Serial_Num	Valid time (from)	(to)
Centurion	AB33	March	June
Cutlass	Z19	June	July
Centurion	AB34	June	August
Caravan	FA2K	April	May

Received(Warehouse, Lot_Num, Part)

Warehouse	Lot_Num	Part	Valid time (at)	
Trump	23	wing strut	May 10 ~ May 29	e_1
Griffin	30	wing strut	May 30 ~ June 18	e_2
Trump	24	engine	June 8 ~ June 27	e_3
Griffin	31	wing strut	June 13 ~ July 2	e_4

Figure 1.1: A valid-time database

The Definite Answer

Lot_Num			Valid time
Warehouse		Part	(at)
Trump	23	wing strut	May 10 ∼ May 29

The Probable Answer

Lot_Num			Valid time
Warehouse		Part	(at)
Trump	23	wing strut	May 10 ∼ May 29
Griffin	30	wing strut	May 30 ∼ June 18

The Possible Answer

Lot_Num			Valid time
Warehouse		Part	(at)
Trump	23	wing strut	May 10 ∼ May 29
Griffin	30	wing strut	May 30 ∼ June 18
Trump	24	engine	June 8 ∼ June 27

Figure 1.2: Answers to example queries

it did not). Similarly, `Lot_Num 24` from the Trump warehouse possibly arrived prior to June 10. The first shipment from the Trump warehouse is also in the possible answer because a definite answer is also a possible answer, but not vice-versa.

Between the possible and definite limits lie other answers. For instance, assume that it is equally likely that `Lot_Num 24` from the Trump warehouse arrived on each day in the set $\{$June 8, June 9, $\ldots,$ June 27$\}$. For the shipment to have arrived prior to June 10, it had to arrive on either June 8 or June 9. If all the days are considered to be equally likely, then there is a probability of only 0.10 (2 chances out of 20) that the the shipment was received prior to June 10. So it is *improbable* that `Lot_Num 24` arrived prior to June 10. However, it is *probable* that both `Lot_Num 30` (0.55 probability, 11 chances out of 20) and `Lot_Num 23` did arrive (1.00 probability). The definite, possible, and "probable" answer to the temporal selection are portrayed in Figure 1.2. If the query language can make use of a probability distribution over the possible times associated with an indeterminate instant, a "richer" query language results, one not restricted to the definite and possible answers. The richness of the query language, however, must not compromise efficient implementation nor detract from the intuitiveness of the language.

There are two stages to determining an answer to a query. The first stage retrieves the data that is relevant to the query. The second stage constructs an answer that satisfies the constraints specified in the query. We provide separate controls on the indeterminacy for each stage.

Range credibility changes the information available to query processing. For instance, given a uniform distribution assumption, it is unlikely that production on the Centurion serial number AB33 had begun early in March, but more likely that it had started by late March, since the likelihood that production had started by a given date is the sum of the probabilities that production begin on each previous day. A typical user might be interested in only those production times that are likely, late March to early June for the Centurion, ignoring those that are unlikely. In TQuel or TSQL2 with indeterminacy the user can express this preference by selecting an appropriate range credibility value. The chosen range credibility potentially modifies every interval in a valid-time relation, restricting the range of each interval. Effectively, non-credible starting and terminating times are eliminated to the chosen level of credibility during query processing, allowing the user to control the quality of the information used in the query.

Ordering plausibility controls the construction of an answer to the query using the pool of credible information. For instance, a Centurion owner could query which shipment of wing struts plausibly arrived during production of his or her plane. Intuitively such a query relaxes the constraints on the relationship between the production times and the day a shipment was received from “absolutely sure of overlap?” to “is it probable that they overlap?” or perhaps to “is it even remotely possible that they overlap?”. The user selects the kind of overlap that she or he requires by setting an appropriate ordering plausibility value. It is probable that lot number 31 from the Griffin warehouse was received during production of the Centurion with serial number AB33, but one cannot be absolutely sure that it did.

In summary, there is a natural division between indeterminacy in the data and indeterminacy in the query. The support for valid-time indeterminacy that we add to TQuel and TSQL2 allows the user to control both kinds of indeterminacy. Range credibility massages the information from which a plausible answer to the query is constructed.

In the next chapter, we informally and formally define what we mean by incomplete information in relational database management systems. We also map the incomplete information landscape, pinpointing previous approaches, and clearly delimit the territory of valid-time indeterminacy. We then examine the representation of valid-time indeterminacy. We present our ontology of time and introduce the three basic indeterminate modeling entities: the indeterminate instant, interval, and span. Next, we add constructs to TQuel to permit creation of relations with indeterminate time values and to express operations on those values. After that, we explore what it means to retrieve information from a database with valid-time indeterminacy. Emphasis is placed on providing a simple and intuitive retrieval method. We extend TQuel's tuple calculus (the formalism used in TQuel's determinate semantics), to handle valid-time indeterminacy. The extended semantics reduces to the current semantics when no indeterminacy is present in a query. We then consider implementation issues. Although retrieving valid-time indeterminate information may appear to be expensive, we outline an efficient implementation. We describe the algorithms used to implement the extended semantics. These algorithms are implemented and performance figures presented. Bit layouts of indeterminate instants, intervals, and spans are presented and shown to be competitive (in terms of space) with existing database timestamp formats. The extended implementation reduces to the current implementation when no indeterminacy is used in a query. In the final chapters we add indeterminacy to TSQL2, and explore the impact of indeterminacy on the modeling of *granularity* and *now*; indeterminacy permits a richer modeling of both concepts.

CHAPTER 2
A TAXONOMY OF INCOMPLETE INFORMATION

Some database management systems can store and use information that is *incomplete*. In this chapter, we informally and formally define what we mean by incomplete information in database management systems, and we classify proposed extensions to the relational data model that can store and query incomplete information. There are many different kinds of incomplete information including information that is fuzzy, imprecise, indeterminate, indefinite, missing, partial, possible, probabilistic, unknown, uncertain, or vague. We will explore each variety of incomplete information in detail below. Because there are so many different flavors of incomplete information, there have been many proposed extensions of the relational model to support incomplete information, far too many to give a detailed account of each in the limited space of this chapter. To make sense of the multitude, we provide a taxonomy that pigeonholes each proposed extension in a general class of incomplete information models.

This chapter is organized as follows. First, we define "incomplete" and build a formal notation for describing databases that contain incomplete information. Next, we apply the formal definition to the kinds of incomplete information found in the literature. We then describe a taxonomy and classify the various research proposals. Finally, we provide a brief summary of the central concepts.

2.1 Definitions

In this section we define what we mean by "incomplete." The definition of incomplete given in *The Concise Oxford English Dictionary* is, perhaps, too concise; it defines incomplete as "not complete" [Sykes 1964]. The definition of complete is of more help; complete means *entire* or *whole*. An object, then, is incomplete with respect to an object that is entire or whole. The incomplete object is missing something from its more complete

partner. For example, an ancient Greek statue missing an arm is incomplete with respect to the statue with that arm. If the statue's arm is unearthed and reaffixed, the statue could be made complete.

If information is represented as a fact (we define what we mean by a fact below), the difference between a complete and an incomplete fact is often a matter of *precision*. For instance, a fact which states that the temperature is 56° is less precise than a fact which states that the temperature is 56.4^o. The 56° temperature fact, however, is more precise than one which asserts the temperature to be $56^o \pm 4^o$ or a fact which states that it could be any temperature.

The informal definition of incomplete information is that information is incomplete because it is missing something from more complete information, and the incompleteness can be assuaged by adding the missing information, which may be unavailable. It is very important to note that information is incomplete only with respect to more complete information (which in turn could be incomplete with respect to still other information).

We will assume that a *fact* is a *tuple* in a relation. Adopting the well-known correspondence between first-order logical models and relational databases [Ullman 1988], we will write a fact as a *ground literal*. For example, assume that we have a relation, *Employees*, with a single attribute, *Name*. Then the fact that Joe is an employee is written: *employee(joe)*. In general, there are two kinds of facts: *extensional* facts and *intensional* facts. The extensional facts are what the database has in its base relations, that is, the axioms of the first-order model. The intensional facts are what can be derived by the database (e.g., views). To simplify this discussion, we will largely ignore what can be derived in a query language (i.e., our first-order model has no proof rules).

The formal definition of incomplete information in database management systems begins with defining the meaning of a database. In this chapter, we will assume that a database, D, is a set of facts, $\{F_1, \ldots, F_m\}$. The meaning of a database, D, written $[\![D]\!]$, is denoted by a *set*, $\{[\![F_1]\!], \ldots, [\![F_m]\!]\}$, where the meaning of each fact, $[\![F_i]\!]$, depends on the fact, but in general, is a *multiset*.[1] (The reason for using a multiset will be made clear

[1] Knuth describes a multiset as "a mathematical entity which is like a set but is allowed to contain reprated elements; an object may be an element of a multiset several times, and its multiplicity of occurrences is relevant" [Knuth 1969].

when probabilistic information is introduced below.) As a notational convenience, instead of replicating identical elements in a multiset, we will associate a numerical coefficient, representing the number of such elements in the multiset, with each element. For example, we will write the multiset, $\{a, b, b\}$ as $\{1 \cdot a, 2 \cdot b\}$. One further notational note, we will drop the coefficient when it is 1. In our simplified model of a database, a database is nothing more than a collection of facts, consequently, in what follows, we will focus just on single facts rather than on the database as a whole.

In the multiset model of a fact, the meaning of a fact is a multiset. Each element in the multiset is a *set* of *literals*. For example, the meaning of the fact $emp(b)$ in our multiset model is $\{1 \cdot \{emp(b)\}\}$, or just $\{\{emp(b)\}\}$. The mappings between facts and their meanings are developed in Section 2.2.

Each fact in the database might or might not be true of the world (i.e., the database might model a fantasy world). We assume that everything that is known about the modeled world is contained in the database.

Each fact potentially has two interpretations: a *definite* interpretation and a *possible* interpretation. The definite interpretation is all the information that that fact *definitely* represents, while the possible interpretation is everything that the fact *possibly* represents. We stipulate that the definite interpretation of a fact is the subset of literals that is common to each set of literals in the meaning fact (recall that the meaning of a fact is a multiset, the elements of which are sets of literals).

Definition 2.1.1 The *definite* information in a fact, $[\![F]\!] = \{f_1, \ldots, f_n\}$, written def_F, is

$$def_F \equiv_{df} \bigcap_{f_i \in [\![F]\!]} f_i \qquad \text{where } \bigcap \text{ is set intersection.}$$

∎

The possible information in a fact is the information that is isolated to specific literals associated with a fact.

Definition 2.1.2 The *possible* information in a fact, $[\![F]\!] = \{f_1, \ldots, f_n\}$, is written $poss_F$.

$$poss_F \equiv_{df} \bigcup_{f_i \in [\![F]\!]} f_i \; - \; def_F \qquad \text{where } \bigcup \text{ is set union.}$$

∎

The formal definition of incomplete information in a fact is related to its possible and definite information. A fact can be incomplete only with respect to another fact.

Definition 2.1.3 A fact F is *incomplete* with respect to a fact F' (written $F <_i F'$) if

$$(def_F \subset def_{F'} \ \wedge \ poss_{F'} \subseteq poss_F) \ \vee \ (def_F \subseteq def_{F'} \ \wedge \ poss_{F'} \subset poss_F)$$

∎

This definition states that F is incomplete with respect to F' if it contains either less definite information or more possible information. We motivate the utility of this definition with an example. Consider the following facts:

$$F_1 = \{\{emp(b)\}\}$$
$$F_2 = \{\{emp(b)\}, \{emp(c)\}\}$$
$$F_3 = \{\{emp(b)\}, \{emp(c)\}, \{emp(d)\}\}$$

The definite and possible information in these facts is given below.

$$def_{F_1} = \{emp(b)\} \qquad poss_{F_1} = \emptyset$$
$$def_{F_2} = \emptyset \qquad poss_{F_2} = \{emp(b), \ emp(c)\}$$
$$def_{F_3} = \emptyset \qquad poss_{F_3} = \{emp(b), \ emp(c), \ emp(d)\}$$

F_2 is incomplete with respect to F_1 because it has both more possible information and less definite information. Similarly F_3 is incomplete with respect to F_2, but only because it has more possible information. We will return to these examples in the next section, because it remains unclear whether the incompleteness relationships among these facts correspond to what is traditionally considered to be incomplete information. We will demonstrate that the above relationships capture some very common kinds of incomplete information.

Incomplete information is present at the database level as well as at the fact level (a database is a set of facts). For instance, a database that contains two facts is incomplete with respect to another database which stores those two facts plus a third. Incompleteness at the database level is defined below.

Definition 2.1.4 A database, D, is *incomplete* with respect to another database, D', if

$$\forall F_i \in D\ (\exists F_c \in D'\ (F_i\ <_i\ F_c)).$$

∎

This definition stipulates that the incomplete database is missing information from its more complete partner. There is a more complete fact in the complete database for every fact in the incomplete database. We will not explore incompleteness at the database level further; it is a issue of incomplete information gathering rather than one of incomplete information within a database context. Instead, we will concentrate on incomplete information at the level of facts.

2.2 Attribute Incompleteness

In this section, we describe various kinds of incomplete information and give the meaning of each type in the context of the formal model developed in the previous section. We limit the discussion to incomplete information stored as attribute values, which we call *attribute* incompleteness to distinguish it from *tuple* incompleteness (whether or not a tuple belongs to a relation) or *query* incompleteness (whether or not a query returns a complete answer). We focus on attribute incompleteness because valid-time indeterminacy is a kind of attribute rather than tuple or query incompleteness.

2.2.1 Common Incomplete Attribute Values

An *unknown* attribute value is a value that is known to exist, but the actual value is unknown. The unknown value is assumed to be a valid attribute value, that is, some value in the domain of that attribute. This a very common kind of incomplete information. For example, in an employee database, while everyone must have an age, Joan's age may be recorded as unknown. The unknown value indicates that Joan has a valid age (i.e. it is not "purple" or "-3" or "INAPPLICABLE"), but we do not know her age. An unknown value has various names in the literature including *unknown null* [Codd 1979], *missing null* [Goldstein 1981], and *existential null* [Biskup 1981, Minker 1982].

In our formal model, we can describe an unknown attribute value as follows. The meaning of a fact, F, with an unknown attribute value over an attribute domain of cardinality N is a multiset with N members; each member is a set containing an F literal with the unknown value replaced a different value from the attribute domain. For example, assume $F = emp(@)$ (where @ represents an unknown value over a domain $\{b,\ c,\ d\}$), then the meaning of F is the meaning of example fact F_3, that is, $\{\{emp(b)\},\ \{emp(c)\},\ \{emp(d)\}\}$. Not surprisingly, F_3 is incomplete with respect to F_1 (F_1 is an unknown value over the domain $\{b\}$, or in other words, it represents a known value). This corresponds to the notion that a fact with an unknown value is incomplete with respect to a fact where that unknown value is no longer unknown, but is now known to be a specific value.

A generalization of an unknown value is an *imprecise* value. An imprecise value is an attribute value that is known to exist and known to be a single value from a *subset* of the attribute domain. For example, in an employee database, if Jill's age is recorded as somewhere between 20 and 30 years old, but her precise age is not recorded, her age is said to be imprecise. A special case of an imprecise value is an unknown value (the subset is the domain itself). Another special case is a precise or complete value (the subset is a singleton set). *Partially known values* [Grant 1979] and *set nulls* [Keller & Wilkins 1985] are imprecise values.

The meaning of an imprecise value in the multiset formulation is similar to that of an unknown value. In our example collection of facts, F_2 is an imprecise value over the subset $\{b,\ c\}$ of the domain $\{b,\ c,\ d\}$. F_2 is incomplete with respect to F_1 because it has no definite information, but not incomplete with respect to F_3 because F_2 has less possible information than F_3. F_2 is more precise than F_3.

Another generalization of an unknown fact is a *disjunctive* fact [Grant & Minker 1986], also known as *indefinite* information [Liu & Sunderraman 1990, Liu & Sunderraman 1991]. A disjunction is a logical *or* applied to literals. For example, Jill might be an employee or a manager (e.g., "emp(Jill) $\vee$ manager(Jill)"). The disjunction is *exclusive* [Ola 1992] or *inclusive* [Homenda 1991]. If it is an exclusive disjunction, one and only one disjunct is true. The meaning of an exclusive disjunctive fact is the same as that of an

imprecise value. Let F be an exclusive disjunctive fact with N disjuncts. The meaning of F is given by a multiset with N members; each member is a set containing one disjunct. For example, assume $F = emp(b) \vee emp(c)$, then the meaning of F is the meaning of example fact F_2.

The meaning of an inclusive disjunctive fact is somewhat different than that of its exclusive counterpart. Let F be an inclusive disjunctive fact with N disjuncts. The meaning of F is given by a multiset with $2^N - 1$ members; each member is a unique subset of disjuncts. For example, assume $F = emp(b) \vee emp(c)$, then $[\![F]\!] = \{\{emp(b),\ emp(c)\},\ \{emp(b)\},\ \{emp(c)\}\}$. The "empty attribute" fact, $emp()$, is not part of the meaning of an inclusive disjunctive fact since at least one alternative value must be the attribute value. The empty attribute represents the situation where a tuple exists, but does not have a particular attribute value (e.g., an unmarried employee tuple exists, but does not any value for the married name attribute).

A *maybe value* is an attribute value which might or might not exist [Gessert 1991]. If it does exist, the value is known. For instance, we could store in our employee database that Jill's phone number may be 555-5555. If Jill has a phone, then this is her number, but she may not have a phone. A *maybe tuple* is similar to a maybe value, but the entire tuple might not be part of the relation [Liu & Sunderraman 1990, Liu & Sunderraman 1991]. Maybe tuples are produced when one disjunct of an inclusive disjunctive fact is found to be true, the other disjuncts become maybe tuples.

Both kinds of maybe information are represented similarly. Let F be a fact with a maybe attribute value. The meaning of F is given by a multiset with 2 members; one containing the literal with the attribute value, the other containing the literal without the attribute value. For example, if $F = emp(jill, \text{maybe } 555 - -5555)$ then the meaning of F is $\{\{emp(jill, 555 - 5555)\}, \{emp(jill)\}\}$. If F is a fact with a maybe tuple then the meaning of F is the multiset with 2 members; one containing the tuple, the other containing the empty attribute fact ($emp()$).

A combination of inclusive disjunctive and maybe information is *open* information. An open attribute value indicates that an attribute of a particular tuple is under the open world assumption [Gottlob & Zicari 1988]. The attribute value may not exist, could be

exactly one value, or could be many values. For example, in the employee database an open value could be used for Jill's previous employment history. This value means that Jill possibly had previous employment (this could be Jill's first job), Jill might have had one previous job, or Jill might have been employeed many times previously. The open value covers all these possibilities.

Let F be a fact with an open attribute value over an attribute domain of cardinality N. The meaning of F is given by a multiset with 2^N members; each member is a unique subset of the domain. For example, assume $F = emp(@)$ (where @ represents an open value over a domain $\{b,\ c\}$), then the meaning of F is

$$\{\{emp(b),\ emp(c)\}, \{emp(b)\}, \{emp(c)\}, \{emp()\}\}.$$

A *no information* value is a combination of an open value and an unknown value [Zaniolo 1984]. The no information value restricts an open value to resemble an unknown value. A no information value might not exist, but if it does, then it is a single value which is unknown, rather than possibly many values.

The meaning of a no information value is similar to that of an unknown value. Let F be a fact with a no information attribute value over an attribute domain of cardinality N. The meaning of F is given by a multiset with $N + 1$ members; each member is a set containing a literal with the no information value replaced by a value from the domain. In addition, the empty attribute fact is also a member of the multiset. For example, assume $F = emp(@)$ (where @ represents a no information value over the domain $\{b,\ c\}$), then the meaning of F is $\{\{emp(b)\},\ \{emp(c)\},\ \{emp()\}\}$.

A generalization of open information is *possible* information [Lipski 1979] (this differs from our use of the term "possible"). Possible information is an attribute value whose existence is undetermined, but if it does exist, it could be multiple values from a *subset* of the attribute domain. For example, in an employee database, Jill's previous employment history could be narrowed to possibly two companies, she could have worked for both companies, only one, or neither. A special case of a possible attribute value is an open value (the subset is the domain itself). Another special case is a maybe value (the subset is a singleton set).

The meaning of possible information in the multiset formulation is similar to that of open information.

Probabilistic information is a variant of imprecise information. A probabilistic data value is a set of alternatives. Each alternative has an associated probability that it is *the* attribute value [Barbará et al. 1989, Cavallo & Pittarelli 1987, Gelenbe & Hebrail 1986]. For example, in the employee database, assume that we do not know Jill's salary exactly but we are 70% sure it is $30,000 and 30% sure that it is $35,000. Jill's salary is a probabilistic data value; the value exists, it is a value from a known subset of the attribute domain, it is exactly one value, and we know that some alternatives are more likely than others. In some models, one of the members of the set of alternatives could be an unknown value [Barbará et al. 1989] in which case the associated probability is distributed uniformly over the elements in the domain.

To represent a probabilistic fact using the multiset notation, the probability of an alternative is proportional to the membership ratio of that alternative in the multiset. Or in other words, the probability is the number of times that alternative appears in the multiset. So if f_i is an alternative of fact F with probability ρ, then

$$\rho \cdot \{f_i\} \in [\![F]\!].$$

For example, if $F = emp((b, 0.25)(c, 0.75))$ (indicating that b is the employee with probability 0.25 and that c is the employee with probability 0.75), then the meaning of F is

$$\{0.25 \cdot \{emp(b)\},\ 0.75 \cdot \{emp(c)\}\}.$$

Note that the weights are ignored in the definition of the definite and possible information in a fact. So the fact F has the same information content as $F' = emp((b, 0.50)(c, 0.50))$ (since $poss_F = poss_{F'}$ and $def_F = poss_{F'}$).

Another variety of weighted information is *fuzzy set* information. Fuzzy set information is similar to possible information. A fuzzy set is a set of possibilities. Each possibility is a maybe value, that is, it may belong to the set or it might not. The possibility that is does belong is given by a membership function (also known as the *degree* of membership).

The degree is a value between 0 and 1 (inclusive). A fuzzy set can be an attribute value [Zemankova & Kandel 1985].

The meaning of a fuzzy set value in a multiset notation is similar to the meaning of a probabilistic value. But a possibility is a subset of the attribute domain rather than just an element of that domain. If we ignore the degree of membership, a fuzzy set value has the same meaning as an open value over the domain composed of the possibilities. That is, if F is a fact with an fuzzy set value with N members, then the meaning of F is given by a multiset with 2^N members; each member is a unique subset of the set of possibilities. For example, assume $F = emp(\{b,\ c\})$ (for the moment, we leave off the degree for b and c), then the meaning of F is $\{\{emp(b),\ emp(c)\},\ \{emp(b)\},\ \{emp(c)\},\ \{emp()\}\}$. The degree of membership will be represented as a coefficient in the multiset.

The meaning of F is given by a multiset with 2^N elements (at least). A subset of possibilities may appear many times in the multiset. The coefficient of each subset is the minimal degree of membership out of all the elements in that subset. That is, if $f_1, \ldots, f_n$ are possibilities of fact F with degrees of membership, $\sigma_1, \ldots, \sigma_n$, respectively, then

$$min(\sigma_1, \ldots, \sigma_n) \cdot \{f_1, \ldots, f_n\} \in [\![F]\!].$$

For example, if $F = emp((b, 0.25)(c, 0.75))$ (indicating that b is an employee with degree of membership 0.25 and that c is an employee with degree of membership 0.75), then the meaning of F is

$$\{0.25 \cdot \{emp(b),\ emp(c)\},\ 0.75 \cdot \{emp(c)\},\ 0.25 \cdot \{emp(b)\}\}.$$

Note that $poss_F$ and def_F are the same even if the elements are weighted differently (unless they are weighted with zero).

The distinction we make between probabilistic and fuzzy attribute values is controversial. Often researchers in the field stress the similarity between fuzzy and probabilistic by advancing proofs that fuzzy values can model probabilistic values. But in the multiset model, the goal is to highlight the differences rather than the similarities between these kinds of values. Therefore, we have labeled the meanings as probabilistic and fuzzy although those kinds of values are capable of modeling each other (with some restrictions).

Note that the meaning of a probabilistic value is a subset of the meaning of a fuzzy value; in the probabilistic meaning, only the singleton sets are retained as elements. For example, the “probabilistic” meaning of the fuzzy value $emp((b, 0.25)(c, 0.75))$ is

$$\{0.75 \cdot \{emp(c)\},\ 0.25 \cdot \{emp(b)\}\},$$

while the “fuzzy” meaning of the probabilistic value value $emp((b, 0.25)(c, 0.75))$ is

$$\{0.25 \cdot \{emp(b),\ emp(c)\},\ 0.75 \cdot \{emp(c)\},\ 0.25 \cdot \{emp(b)\}\}.$$

2.2.2 Truncation Incompleteness

Attribute values in database relations are sometimes shortened to accommodate limited space resources. A common example is limiting the size of surnames, by truncating any name that exceeds a fixed, but small, number of characters. But, truncation sometimes discards information. In those case where relevant information is lost, a truncated fact is incomplete with respect to its untruncated partner. We will refer to this kind of incompleteness as *truncation incompleteness.* This kind of incompleteness has received scant attention although database storage strategies and techniques have been extensively researched.

The multiset meaning of a fact accommodates truncation incompleteness by treating the truncated value as an imprecise value. The imprecise value is a value from a known subset of the attribute domain. Very informally, the known subset is all those members of the entire attribute domain that include the non-truncated portion of the value. For example, assume that the name attribute of an employee relation is limited to names of three characters or fewer. The employee with the name “Jill” is truncated by the database to the fact $emp(jil)$. The multiset meaning of $emp(jil)$ (to four characters) is $\{\{emp(jil),\ emp(jila),\ emp(jilb),\ \ldots,\ emp(jilz)\}\}$.

2.2.3 Complete Information Null Values

Unknown, partially known, open, no information, and maybe null values are different interpretations of a null value. There are other null value interpretations, but none of these is a kind of incomplete information.

An *inapplicable* or *does not exist* null is a very common null value. An inapplicable null, appearing as an attribute value, means that an attribute does not have a value [Grant 1977]. The prototypical example is an employee relation with a Maiden name attribute. The employee Jill may not be married, hence she does not have a Maiden name, and the value for this attribute would be an inapplicable null. An inapplicable value neither contains nor represents any incompleteness; it is known that the attribute value does not exist. Inapplicable values indicate that the schema (usually for reasons of efficiency or clarity) does not adequately model the data. The relation containing the inapplicable value can always be decomposed into an equivalent set of relations that do not contain it [Atzeni & Parker 1982, Lerat & Lipski 1986, Linn 1987]. Hence the presence of inapplicable values indicates inadequacies in the schema, but does not imply that information is being incompletely recorded.

A single null value is often semantically overloaded to mean either an unknown value or an inapplicable value [Codd 1986, Codd 1979, Date 1986] in which case it is an no information null. Problems that result from such an overloading have been identified [Zaniolo 1982, Zaniolo 1984]. Recently, use of four-valued logics has been advocated to untangle the semantic confusion [Gessert 1990].

Another non-incomplete information null is a *universal* null [Biskup 1981]. The universal null, appearing as an attribute value, means that every value from the attribute domain is an attribute value. For example, assume that a Parts relation has a color attribute and that the "crayon" part comes in every color. To save space, we could store a single "crayon" tuple with the relation and use a universal null to indicate the color. The universal null serves as an abbreviation for a set of facts.

2.3 A Taxonomy of Incomplete Information Data Models

The formal model developed in previous sections serves as a foundation for classifying various schools of thought in incomplete information databases. In this section, we propose a tree structured taxonomy based on issues that are illustrated with the formal model. Below we describe these issues in some detail. Each interior node in the taxonomy tree is an issue choice point. A branch indicates a choice made on an issue. Some of

the choices are orthogonal. That is, there are data models that can support both choices. Others are important special cases. At each leaf in the taxonomy tree are listed research proposals that fit the choices made on the path from the root to the leaf.

2.3.1 Value vs. Temporal Incompleteness

The first issue in the taxonomy concerns whether the proposed model supports *value* or *temporal* incompleteness. The difference between value and temporal incompleteness can be characterized as that between "don't know what" and "don't know when" information. The choice of which kind of information to model is orthogonal, that is, there are (or could be) data models that support both. For instance, although this dissertation focuses exclusively on temporal incompleteness, temporal indeterminacy can be combined with support for null values in nontemporal attributes. The taxonomy subtree for both value and temporal incompleteness has exactly the same structure, which is described further below.

Research in value incompleteness has been extensive. The field is certainly deserving of a taxonomy. The relative newness of valid, transaction, and bitemporal databases has limited research on temporal incompleteness. However, we believe that there is enough work to make classification meaningful.

Another kind of incomplete information is spatial incompleteness ("don't know where"). Yet another is abductive incompleteness ("don't know why"). In the interest of space, we shall leave investigation of these (and other) areas to future researchers with the warning that our taxonomy is as yet "incomplete."

2.3.2 Alternative vs. Possible

The second choice to be made concerns whether the incomplete information models an attribute value that can be a set of values or just a single value. An incomplete data value can either be a set of *alternatives* or a set of *possibilities*. Exactly one *member* in a set of alternatives is the attribute value; which member is unknown. Any *subset* of a set of possibilities could be the attribute value; which subset is unknown (and could be the empty set). Alternative proposals are more common in databases, since it is atypical to

have set attribute values (e.g., nested relations). In the formal model, the meaning of a data value with N alternatives (and without weights) is typically a multiset with $O(N)$ members, whereas the meaning of a data value with N possibilities is a multiset with $O(2^N)$ members.

2.3.3 Unweighted or Weighted

Weights play an important role in some data models. Weights are typically normalized values in the range [0,1]. Weights are assigned to individual alternatives or individual possibilities in an incomplete value. For an alternative, a weight gives the chance or probability that the alternative is *the* actual value. For a possibility, a weight indicates the likelyhood that the possibility is *an* actual value.

The unweighted school is an important special case of the weighted school. An appropriate weighting scheme can usually be used to encode unweighted information (e.g. using uniform weights). However, the query evaluation semantics with weighted incomplete information differs substantially from that with unweighted information since queries need to be able to utilize the weighted information. Unweighted means not only that weights are not present, but that query evaluation semantics make no use of weights.

Unweighted incomplete information may be either *unrestricted* or *restricted.* By restricted we mean that the value of each possibility or alternative is restricted to a subset of the attribute domain. The subset to which it is restricted must be encoded as part of the incomplete information in the value (if it were kept in the schema, the restriction would simply be to the domain of the attribute). Queries must take account of the restriction during query evaluation. An unrestricted value has no such constraints.

In the weighted case, this distinction is made by the weighting scheme. While weighting schemes use restricted values, an appropriate weighting scheme (using zero weights) can mimic the unrestricted case.

2.3.4 The Taxonomy Tree

The taxonomy tree is split over several figures. Figures 2.1 and 2.2 are the value incomplete proposals, while Figures 2.3 and 2.4 are the temporal incomplete propos-

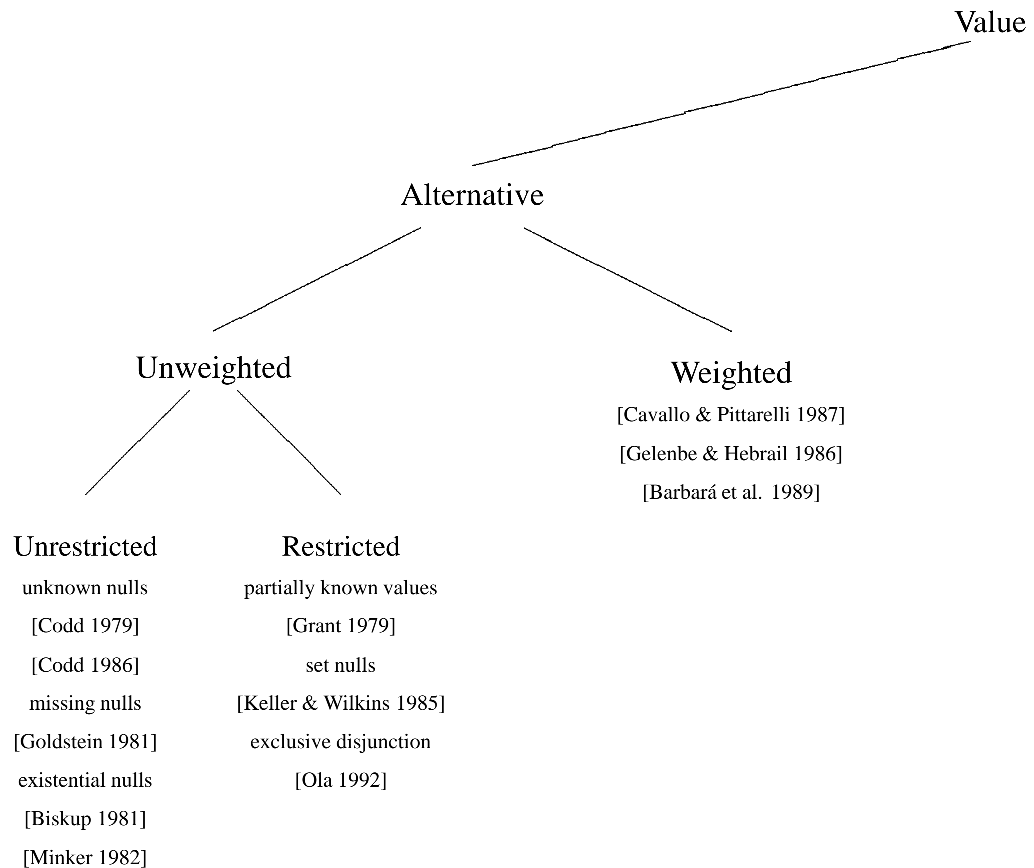


Figure 2.1: The taxonomy subtree for value/alternative incompleteness

als. The value incomplete alternative schools are shown in Figure 2.1. Most of the work in incomplete information are represented by these schools, in particular, at the Value/Alternative/Unweighted/Unrestricted leaf. This is where unknown null values reside. Null values are incorporated in the SQL-92 standard [Melton & Simon 1993] and are supported in several commercial products. In fact, no major database vendor supports any other kind of incomplete information. The other figures showing the taxonomy cover Value/Possible (Figure 2.2), Temporal/Alternative (Figure 2.3), and Temporal/Possible proposals. (Figure 2.4).

In the taxonomy the classification of a school is given by the path from the root to the leaf. The primary difference between the schools is that each models a different kind of incomplete information. In Figure 2.5 we give a table showing situations with different kinds of value incompleteness, an example of a tuple with an incomplete data value, and

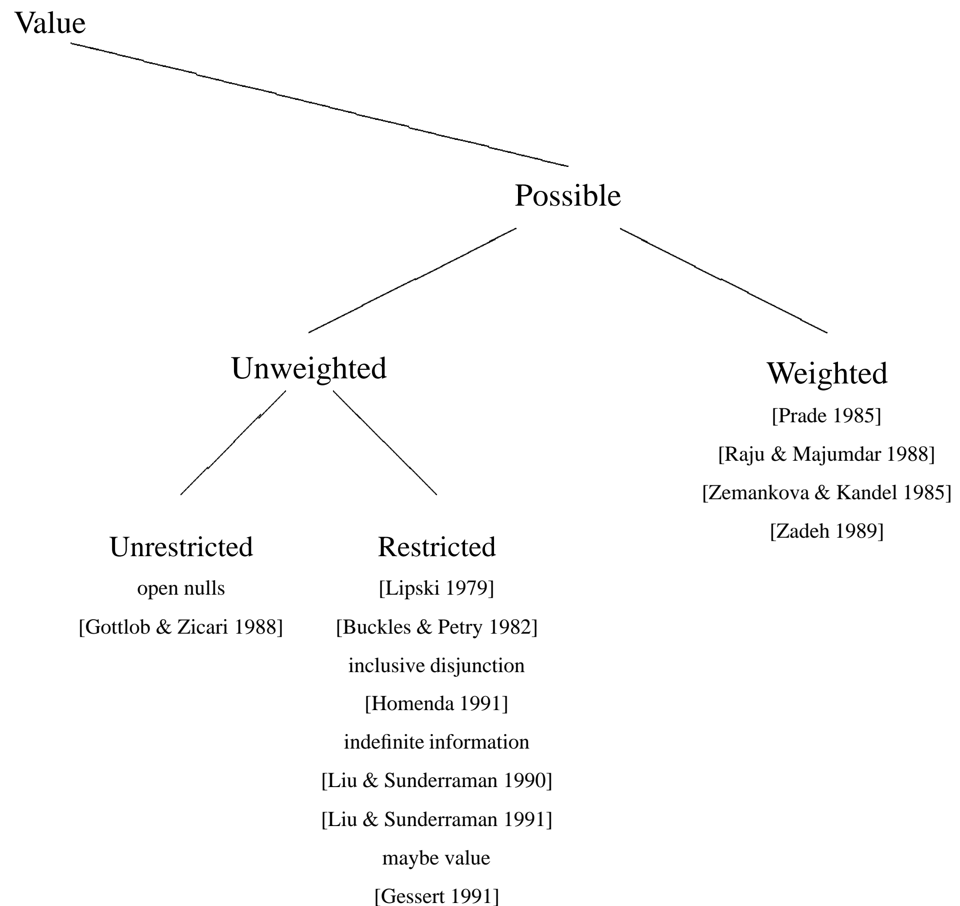


Figure 2.2: The taxonomy subtree for value/possible incompleteness

to which leaf in the taxonomy the tuple belongs. Figure 2.6 shows a similar tripartite table for temporal incompleteness.

The relations depicted in the figures relate an individual to zero, one, or more contracted diseases. Each tuple in the two relations represents a different school (as indicated in the **SCHOOL** column). The difference lies in the interpretation of the information recorded for the **DISEASE** attribute in Figure 2.5 and the **VALID** attribute in Figure 2.6. The intended interpretation is given by the **SITUATION** column.

2.3.5 Valid-time Indeterminacy

The taxonomy shows that valid-time indeterminacy is in the temporal/alternative/weighted school. Indeterminacy models the situation where exactly one member of a set of times is the actual time, but we do not know which member. In addition, each member of the

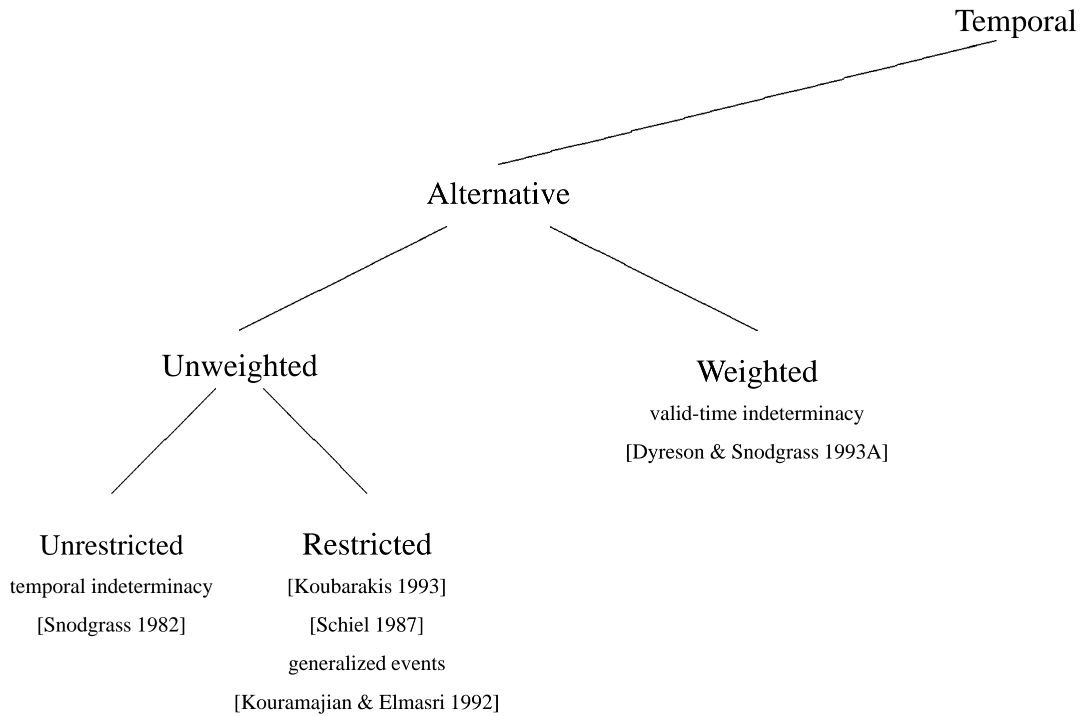


Figure 2.3: The taxonomy subtree for temporal/alternative incompleteness

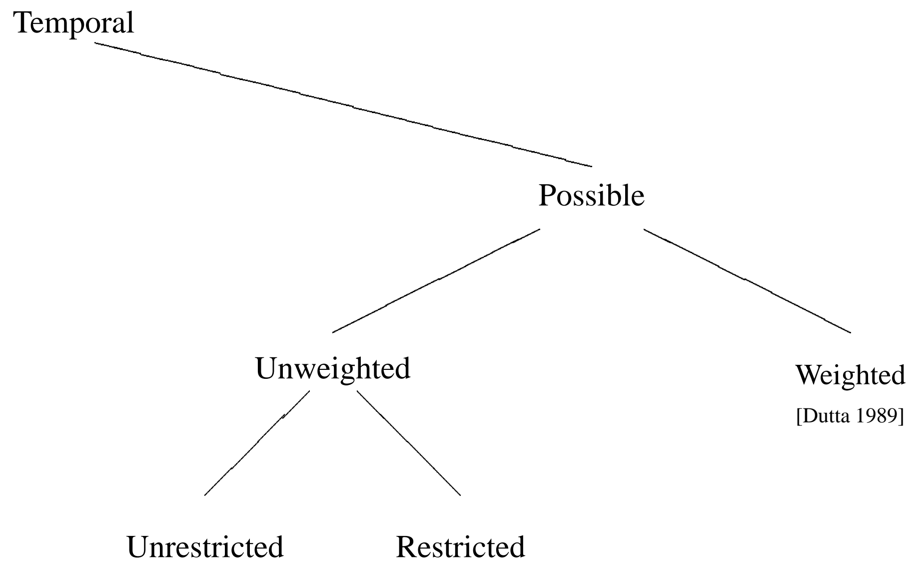


Figure 2.4: The taxonomy subtree for temporal/possible incompleteness

SITUATION	PERSON	DISEASE	SCHOOL
Jethro has Malaria	Jethro	Malaria	No incompleteness
Juan has one disease, but we aren't sure which	Juan	$Unknown_1$	alternative/unweighted/unrestricted
Jerome has either Malaria or Typhoid	Jerome	Malaria or Typhoid	alternative/unweighted/restricted
There is a 70% chance Jules has Malaria but only a 30% chance he has Typhoid.	Jules	Malaria .7 Typhoid .3 Rest 0	alternative/weighted
Juana has zero or more diseases.	Juana	$Unknown_2$	possible/unweighted/unrestricted
Julie has Malaria or Typhoid, perhaps both.	Julie	Malaria, Typhoid	possible/unweighted/restricted
There is a "good" possibility Jill has both Malaria and Typhoid	Jill	Malaria .8 Typhoid .9 Rest 0	possible/weighted

Figure 2.5: The situation modeled by schools in value incompleteness

set of times has an a weight. In valid-time indeterminacy, the weight is interpreted as a probability, that is, the weight gives the probability that the time is the actual time. The weights in a set must sum to 1. Indeterminacy has one further condition that is not shown in the taxonomy. The set of probability distributions is limited by the implementation, as described further in Chapter 5.

2.4 Indeterminacy and Related Work

In this section, we more closely examine research related to indeterminacy. Despite the wealth of research on adding incomplete information to databases [Dyreson 1993], there are few efforts that address incomplete temporal information. Much of the previous research in incomplete information databases has concentrated on issues related to null values (e.g., [Codd 1990, Date 1986, Vassiliou 1979, Zaniolo 1984]). Another primary research thrust has studied the applicability of fuzzy set theory to relational databases (e.g., [Dubois et al. 1988, Prade 1993, Zemankova & Kandel 1985]). By and large, most of

SITUATION	PERSON/ MALARIA	VALID	SCHOOL
Jethro had Malaria at time 2	Jethro	2	no incompleteness
Juan had Malaria, but when?	Juan	$Unknown_1$	alternative/unweighted/unrestricted
Jerome had Malaria at either time 2 or time 8	Jerome	2 or 8	alternative/unweighted/restricted
Jules had Malaria either at time 2 or time 8 with the indicated probabilities	Jules	2 .7 8 .3 Rest 0	alternative/weighted
Juana might have had Malaria, perhaps repeatedly, but when?	Juana	$Unknown_2$	possible/unweighted/unrestricted
Julie had Malaria at time 2, or time 8, or times 2 and 8, or never.	Julie	2 or 8	possible/unweighted/restricted
There is a “good” possibility Jill had Malaria at both time 2 and time 8	Jill	2 .8 8 .9 Rest 0	possible/weighted

Figure 2.6: The situation modeled by schools in temporal incompleteness

these proposals do not address the issue of *cost*, which is a primary focus of our research. We first place our work in the context of *value* incompleteness and then examine in detail several papers that concern *temporal* incompleteness.

Information that is valid-time indeterminate is similar to disjunctive information, especially in the context of deductive databases [Liu & Sunderraman 1990]. Disjunctive information is in the value/alternative/unweighted/restricted school. Disjunctive information is a collection of facts, one (or more) of which is true. Indeterminacy is of the exclusive-or variety of disjunctive information (only one disjunct is true) [Ola 1992], but differs from the above investigations because the alternatives are “weighted” and the weights are integrated into the query semantics.

The field of probabilistic databases covers a wide spectrum of different uses of probabilistic information. Probabilistic weights have been attached to attribute values modeling situations where an attribute value could be one of several more or less likely values [Barbará et al. 1990, Barbará et al. 1992, Gelenbe & Hebrail 1986]. These research efforts are in the value/alternative/weighted school. Probabilistic weights have also been appended to tuples, where the weight is the probability that the tuple belongs to the relation [Cavallo & Pittarelli 1987, Kornatzky & Shimony 1993A, Kornatzky & Shimony 1993B, Zimányi 1992]. Decision support systems, vague queries and data mining have also utilized probabilistic information [Fuhr 1990, Henrion & Suermondt 1993, Wong 1982]. Our work concerns only probabilities in attribute values and can be seen as an extension of the Probabilistic Data Model (PDM) [Barbará et al. 1992]. In PDM, attribute values are sets with weights attached to each element. The weight is the probability that the corresponding element is *the* value of the attribute. Queries use the probabilistic representation in conjunction with a single user-given "confidence" to compute a result within the framework of the possible world semantics. The novelty in our work can be seen in the methods used to retrieve the incomplete information and in how that information is represented. In PDM each element in a set of possible values is stored and processed separately. The cost of the probabilistic operators in PDM are proportional to the number of alternatives in the set (some of the operations have a cost that is proportional to the square of the number of alternatives). We could not adopt the PDM approach since there might be several million elements in a set of possible chronons. The encoding of the probabilistic information that we developed, using a period of indeterminacy and a mass function, is space efficient; and the operations on that encoding are time efficient. In addition, our use of both credibility and plausibility values permits greater flexibility and finer control in query evaluation. These techniques are a product of the unique nature of the type of incomplete information, valid-time indeterminacy, that we investigated. The remaining approaches that we discuss do not use probabilities.

In the earliest work on incomplete temporal information, an indeterminate instant was modeled with a set of possible chronons [Snodgrass 1982]. *Before* was extended to return the value *unknown*, necessitating an extension to a three-valued logic. Also, a

four-valued logic was proposed to model times and values that are unknown, imprecise, or *negative* (under the open world assumption) [Schiel 1987]. Our current approach allows a probability mass function to be associated with each indeterminate instant, and does not require a multi-valued logic.

Dutta uses a fuzzy set approach to handle *generalized temporal events* [Dutta 1989]. A generalized temporal event is a single event that has multiple occurrences. For example the event “Margaret’s salary is high” may occur at various times as Margaret’s salary fluctuates to reflect promotions and demotions. The meaning of “high” is incomplete. “High” is not a crisp predicate. In Dutta’s model all the possibilities for “high” are represented in a *generalized* event and the user selects some subset according to his or her interpretation of “high.” This contrasts with the task of encoding the type of information we have characterized as valid-time indeterminate. We view events as having a single occurrence. An indeterminate instant is a set of alternatives, one and only one of which is the actual time. Every member in a fuzzy set is always possible, to a greater or lesser extent, depending on the degree of membership, but always possible (although some fuzzy databases stipulate by fiat that only one member is possible [Dubois et al. 1988]). Our approach and that of Dutta’s model different kinds of temporal incompleteness. We feel that a probabilistic approach is better suited to modeling valid-time indeterminacy as formulated in this paper, and that fuzzy set approaches like Dutta’s (e.g., [Dubois & Prade 1989, Vitek 1983]), are better suited to modeling generalized events. The two approaches are orthogonal, and the user may pick the one(s) most appropriate to her application.

Generalized bitemporal elements are defined somewhat differently in a more recent paper [Kouramajian & Elmasri 1992]. Bitemporal elements combine transaction time and valid time in the same temporal element. Since TQuel also supports transaction time, valid-time indeterminacy and generalized bitemporal elements differ mainly in their handling of valid time. In Kouramajian and Elmasri’s model, both the upper and lower support on a valid time interval could be a set of noncontiguous possible chronons. Unlike valid-time indeterminacy no probabilities are used. Since there are no probabilities, the user in general is limited to querying for answers which are either “definite” or those which are “possible” (or combinations thereof). Historically, these alternatives have a

well-defined meaning in incomplete information databases [Lipski 1979]. Generalized valid times are composed of valid times by the operators of alternation (only one valid time applies) and/or union (both valid times could apply). We provide no capability for "generalizing" valid times to handle alternation or union.

Other approaches consider the subtle interaction of determinate timestamps and incomplete information [Gadia et al. 1992, Kurutach & Franklin 1993]. In contrast, we study timestamps that are incompletely specified.

Recently, global and local inequality constraints on the occurrence time of an event have been added to a temporal data model [Koubarakis 1993]. The resulting model supports *indefinite* instants. An indefinite instant is a very general kind of instant that includes indeterminate instants, instants with disjoint sets of possible chronons, and instants with incompletely specified upper and lower supports. For instance, we may know that α occurred before β but after 2 PM (2 PM $< \alpha < \beta$), and we may know that β happened before 4 PM. In Koubarakis's data model we can then conclude that α happened between 2 and 4 PM. Koubarakis explores the complexity of query processing and achieves polynomial time complexity for retrieving information by restricting the kinds of constraints that are allowed (to disjunctions of inequalities). The primary difference between his model and ours (other than a difference in weights) is that in TQuel (and TSQL2) no inter-tuple constraints are supported. Consequently, Koubarakis supports a much richer set of global constraints and is able to model temporal information that we cannot (and vice-versa for probabilistic information). We believe that adding global constraints to TQuel (and TSQL2) would adversely impact performance, and to reach our goal of an efficient implementation, we restricted the kinds of incomplete information that we could represent.

We note that there is little discussion in any of the aforementioned papers of implementation aspects. We feel that both efficient representations and efficient query processing algorithms are essential, especially when the incomplete information is weighted.

Finally, the approach to valid-time indeterminacy espoused by Kahn and Gorry [Kahn & Gorry 1977] is reminiscent of those employed by the artificial intelligence community [Maiocchi & Pernici 1991]. In their model, instants and intervals are specified relative to

each other; only a subset are actually tied to the valid time line. An instant may only be known to have occurred, say, between two other instants. Their model is more general than our data model without probabilities, but also exhibits significant query processing overhead.

2.5 Summary

In this chapter we gave a simple definition of an incomplete fact, and built that definition into a formal model, which in turn served as a foundation for a taxonomy of incomplete information database proposals. We observed that a fact is incomplete only in relation to another fact, rather than somehow being "intrinsically" incomplete. This observation led to the development of a formal model. In the model, the meaning of a fact is given in terms of the definite and possible information associated with the fact. A fact is incomplete with respect to another fact if it has less definite or more possible information. We then sketched the meaning of several common kinds of incomplete data values, such as unknown, imprecise, and probabilistic values. These values model different kinds of incomplete information. Previous research in this area is extensive, and it is not always clear which kind of incomplete information is being investigated. We proposed a simple taxonomy to classify incomplete information data models. A researcher interested in a particular kind of incompleteness can refer to the appropriate branch in the taxonomy to spot the data models that can serve her or him best. This is by no means an exhaustive classification.

CHAPTER 3
TIME MODEL

Temporal relational databases add comprehensive support for time to relational databases. This support rests on three temporal dimensions: valid time, and transaction time, and user-defined time. As discussed previously, valid time is the real world time of a fact. Transaction time is the database time during which that information was stored. User-defined time is an uninterpreted temporal domain. A single model of time, however, is the foundation for each of these separate dimensions. In this chapter, we present our model of time. We focus on the concepts of instant, interval, and span. We discuss how each is modeled, and we give an overview of the semantics of operations on the modeled entities.

3.1 The Ontology of Time

Time has a standard geometric metaphor. In this metaphor, time itself is a line; a point on the time-line is called an *instant*; the time between two instants is known as a *time interval* (interval for short); and a length, or unanchored segment, of the time-line is a *span* [Jensen et al. 1994].

In the temporal database community, three basic time models have been proposed: the *continuous model*, in which time is viewed as being isomorphic to the real numbers, with each real number corresponding to a "point" in time; the *dense model*, in which time is viewed as being isomorphic to the rationals; and the *discrete model*, in which time is viewed as being isomorphic to the integers [Clifford & Tansel 1985]. Science and metaphysics have yet to determine which model best fits reality; good arguments can be made for each of the three models (some quantum theories view time as ultimately quantized or discrete [Anderson 1982]). In contrast, we abstain from choosing among these three models, that is, the choice is unimportant. What is important is how we

represent the *times* in our model. We assume that the representation is both discrete and finite, that is, we assume that we are limited to representing a finite number of different times or "collections" of times. This assumption is strongly influenced by practical considerations since the times stored in our database are encoded in fixed-sized records called *timestamps*. Since timestamps are a fixed number of bits, they can represent only a finite number of different values.

In our ontology, time is modeled as a closed interval of the natural numbers (or, alternatively, the real numbers). Conceptually, while time may extend into the infinite future or the infinite past, in our model, time is bounded at both ends. This is not a crucial feature of our model, which could just as easily be unbounded; rather it is a pragmatic choice. Times are stored in fixed-sized data structures. An unbounded range of times would make such a storage scheme impractical since timestamps would also need to be unbounded in size. The choice is also consistent with a popular cosmology which asserts that time begin at the "Big Bang," 14 ± 4 billion years ago, and will end at the "Big Crunch," sometime in the future [Dyreson & Snodgrass 1994A].

3.2 A Discrete View of Time

The closed interval of time is partitioned into a finite number of discrete, smaller segments known as *chronons* [Ariav 1986, Clifford & Rao 1987, Jensen et al. 1994]. Our model of time does not mandate a specific chronon size or *(minimum) granularity*; a chronon may be of any duration (e.g., nanoseconds, years, Chinese imperial dynasties). We believe that specifying the minimum granularity should be left to the implementation rather than fixed in the data model. Our time model has only a single granularity; multiple granularities can be handled by representing the indeterminacy explicitly or implicitly (as described further in Chapter 7). The chronons are consecutively labeled with the integers in the set $\{0, \ldots, N\}$, where N is the number of different values that our timestamp can represent. The set of chronons is linearly ordered. There are two instants, just outside the closed interval of time, called *beginning* and *forever*, that are the least and the greatest values, respectively, in the linear ordering.

3.3 Modeling Instants

To model instants, we introduce an instant timestamp. An instant timestamp stores the granule number of the time that it represents. A chronon is the smallest amount of time that an instant timestamp can represent. But a chronon is a segment of a time-line whereas an instant is a point on a time-line. What then is the relationship between chronons and instants?

We could either assume that chronons are the same size as instants or we could assume that the duration of chronons far exceeds that of instants, that is, that every chronon contains a large (possibly infinite) number of instants. If we assume that chronons and instants are the same, then we must also use the discrete model of time. If the model of time were continuous or dense then there would have to be an infinite number of chronons because, in either of these two models, there are an infinite number of instants in any non-zero length segment of the time-line. Since the theme of our model is that time is (possibly) continuous, we must assume that chronons are much bigger than instants.

An instant timestamp records that an instant is located sometime *during* a particular chronon (e.g., a day). Without loss of generality, we assume that an instant timestamp represents any instant during a chronon. Hence, at a very abstract level, the *exact* instant modeled by an instant timestamp is never precisely known. At best, only the chronon during which it is located is known. Two instants that are represented by the same chronon may still model different instants. For example, assume that two different instants are positioned during the same hour-long chronon. The two instants are represented by identical instant timestamps, each with a granularity of an hour.

Alternatively, we might assume that when we represent an instant, we are actually representing the very first instant in each chronon. However, the first assumption, that an instant is sometime within a chronon better describes actual clock measurements. When we glance at a wristwatch and report that the time is currently "3:45," we typically do not mean the very first instant in the 45th minute after three o'clock, rather, we mean that we simply do not know more than that it is sometime within the minute described by "3:45."

All (current) operations on timestamps support both of these (conflicting) assumptions, and a user will be unable to detect any difference between these two assumptions by testing various operations. Stated differently, if we assume that an instant may occur anywhere within a chronon, we may just as easily assume that it is always the first instant within that chronon because no timestamp operation uses the location of an instant within a chronon. So while the distinction between the two assumptions has no practical import, the first assumption better models the process by which we derive temporal information.

Yet another alternative is to assume that the instant is really the entire chronon. However, this is not what is meant by an instant. An instant is instantaneous (of no duration) rather than being a chronon, an hour, a day, or even a year in duration. Nor would such an assumption model the reality of clock measurements. We when glance at our wristwatch and report the time is currently "3:45," we typically do not mean that it is the entire chronon (assume nanoseconds) starting with "3:45."

In summary, an instant is modeled by an instant timestamp that stores a chronon label. The instant modeled by the timestamp is some instant within the indicated chronon. We will use the delimiters "`| |`" to denote an instant timestamp. For example, the constant `|June 1, 1993|` (assuming a chronon size of a day) models some instant during the 1st day of June in the year 1993. It is important to note that we cannot ask which instant, that is, there is no timestamp operation that permits us to ask which instant. Furthermore, there is no operation which will require us to know more about the instant than that it is sometime during that day. For example, the operation that adds the above instant timestamp to a span of "3 days" is akin to asking, "In terms of days, what is some instant during June 1, 1993 displaced by 3 days?" The answer is some instant in `|June 4, 1993|`.

3.4 Modeling Intervals

To model intervals, we introduce a interval timestamp. The timestamp is the composition of two instant timestamps and a constraint. The constraint is that the instant timestamp that starts the interval equals or precedes (at the level of chronons) the instant timestamp that terminates the interval. We will use the delimiters "`[ ]`" to denote an interval timestamp.

For example, the interval timestamp [June 1, 1993 - June 1, 1993] means that the interval is all within the same day, but we are not sure when it begins or ends during that day (note that |June 1, 1993| equals |June 1, 1993| at a chronon size of days).

3.5 Modeling Spans

A span in an unanchored duration. To model spans, we introduce a span timestamp. The timestamp is a count of chronons. We will use the delimiters "%%" to denote a span timestamp. For example, the span timestamp %3 days% is a count of three day-sized chronons.

3.6 Impact of the Model on the Semantics of Timestamp Operations

The partitioning into chronons creates a discrete image of a (possibly) continuous underlying time-line. Operations on timestamps are defined with respect to this discrete view of time. For example, if the chronon size is days, then a comparison operation at the granularity of days compares days. It is very important to note that operations on *instants* are not supported; the granularity of chronons is the smallest possible granularity. Consequently, timestamp operations that are performed at the level of instants do not exist (i.e., we cannot ask if one instant precedes, in an image of instants, another instant). In fact, such a question is not only unaskable, but unanswerable in our model of time since an instant timestamp does not record the exact location of an instant.

3.7 Summary

The theme for our model of time is that users manipulate a discrete image of a time-line that is possibly continuous, dense, or discrete. The discrete image is created by partitioning the time-line into chronons. Instant timestamps model durationless temporal values, instants, that are located sometime during a particular chronon. Interval timestamps model temporal values with duration, intervals, that are represented as a sequence of chronons. Span timestamps model unanchored temporal values with duration, spans, that

are represented as a count of chronons. Timestamp operations are defined with respect to the discrete image and are performed to the granularity of chronons.

CHAPTER 4

EXTENSIONS TO TQUEL

In this chapter the TQuel query language [Snodgrass 1987] is extended with support for indeterminacy. TQuel is a strict superset of Quel, the query language for Ingres [Stonebraker et al. 1976]. In the next section indeterminate instants, spans, and intervals are added to TQuel's data model. To make use of the increased power of the data model, the query language is then extended to support queries on indeterminate values. We focus on changes to the syntax and semantics of the `retrieve` statement, in particular, on the addition of controls for the range credibility and ordering plausibility.

4.1 Extending the Data Model with Indeterminacy

In this section, we discuss how to represent indeterminate instants, intervals, and spans in TQuel's data model. In Section 5.1 we discuss how these representations are implemented.

4.1.1 Indeterminate Instants

An instant is a point on the time-line [Jensen et al. 1994]. An instant is *determinate* if it is known when (i.e., during which chronon) it is located. Often, however, we do not know the exact chronon during which an instant is located; instead, we only know that the instant is located sometime during a set or range of chronons. We call such an instant an *indeterminate* instant. The indeterminacy refers to the location of the instant, not whether the instant exists. Indeterminate instants do not model the situation where it is unknown if an instant exists. For example, we may know that a plane left sometime on June 12, 1994. With minute-sized chronons, we do not know the exact minute during which that plane departed, but only that it departed sometime during an interval represented by 1440 consecutive chronons.

An indeterminate instant is described by a *lower support*, an *upper support*, and a *probability mass function (p.m.f.)* [Dyreson & Snodgrass 1993A]. The supports are chronons that delimit when the instant is located; the instant is no earlier than during the lower support and no later than during the upper support. Between the supports lies a *period of indeterminacy*. The period of indeterminacy is a contiguous set of *possible chronons*. The instant is located during some chronon in this set, but which chronon is unknown. We denote a set of possible chronons that extends from the lower support, α_*, to the upper support, α^*, using the notation $|\,\alpha_* \sim \alpha^*\,|$, e.g., `|May 10 ~ May 29|`.

4.1.1.1 The Probability Mass Function

Although the instant is located during some possible chronon, not all the possible chronons are equally likely. For example, it could be that the instant is most likely located during the earliest chronon in the period of indeterminacy. The probability mass function gives the probability of each chronon. The probability mass function, P_α, for the indeterminate instant, α, is

$$P_\alpha(i) \;=\; \mathbf{Pr}[\alpha = i] \quad i \,\in\, \{0, 1, \ldots, N\}$$

where $\mathbf{Pr}[\alpha = i]$ is the probability that the instant is located during chronon i. Since the instant is not any time outside the period of indeterminacy, $\mathbf{Pr}[i < \alpha_*] = 0$ and $\mathbf{Pr}[i > \alpha^*] = 0$. All indeterminate instants are considered to be independent, that is

$$\mathbf{Pr}[\alpha = i \;\wedge\; \beta = j] \;=\; \mathbf{Pr}[\alpha = i] \times \mathbf{Pr}[\beta = j].$$

Like other probabilistic databases, we make no provisions for joint or dependent probabilities [Barbará et al. 1990, Barbará et al. 1992, Cavallo & Pittarelli 1987, Gelenbe & Hebrail 1986, Kornatzky & Shimony 1993A, Kornatzky & Shimony 1993B, Zimányi 1992]. We sometimes denote an indeterminate instant, α, using the notation, $(\,|\,\alpha_* \sim \alpha^*\,|\,, P_\alpha)$.

4.1.1.2 Mass Function Sources

The probability mass function for an indeterminate instant is supplied by the user. In Section 5.2 we show how users can provide mass functions that are relevant to their

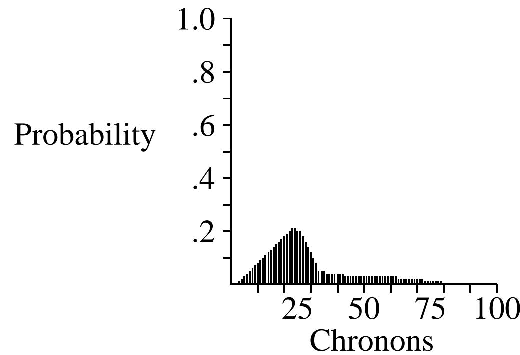


Figure 4.1: A "probably early" distribution

applications. For example, the mass function depicted in Figure 4.1 might illustrate the probability that a performer is "gonged" on *The Gong Show* (the performance is likely to end early).

In many common cases, the probability mass function for an indeterminate instant stems from the source of the indeterminacy.

- *Granularity mismatch* — The uniform or equi-probable mass function is a useful assumption. For example, an instant known to within one day and recorded on a system with timestamps in the granularity of a microsecond happened sometime *during* that day, but during which microsecond is unknown.

- *Dating techniques* — A property of radioactive dating techniques is that the estimate is described by a normal, "bell-shaped curve" distribution.

- *Uncertainty in planning* — Analysis of past data (the past data may be readily available in a temporal database) can sometimes provide a good indicator of future performance (when used carefully). For instance, we may not know exactly when an airline will depart. However, an analysis of past departure times for that route, type of airline, and day of the week (the analysis could be much more elaborate) may show that this flight tends to leave later than scheduled. Based on this analysis, a "probably late" distribution could be used for the departure time of that flight.

- *Unknown or imprecise instants* — Typically, if the location of an instant is unknown,

the distribution is also unknown. In these situations, a user can specify that the distribution is *missing* (see below).

- *Clock measurements* — Clock-specific distributions model the imprecision of specific clock measurements [Petley 1991].

In general, a user just may not know the underlying mass function because that information is unavailable or the mass function could exceed the implementation capacities of the system (Section 5.2 describes the implementation and the constraints it imposes on mass functions). In such cases, the distribution can be specified as *missing*. A distribution that is missing represents a complete lack of knowledge about the distribution. It is a kind of second-order incompleteness, that is, the distribution that is missing is incomplete information about indeterminate information. Unlike some other probabilistic data models [Barbará et al. 1990, Barbará et al. 1992], we do not allow partially known distributions.

While the terminology introduced so far suggests a difference between indeterminate and determinate instants, it is instructive to note that an indeterminate instant can be used to model a determinate instant. A determinate instant is modeled by an indeterminate instant with a singleton set of possible chronons. A determinate instant records that an instant is located sometime *during* a particular chronon. Without loss of generality, we assume that a determinate instant represents any real-world instant during a chronon. Hence, at a very abstract level, the *exact* real-world instant modeled by a determinate instant is never precisely known. At best, only the chronon during which it is located is known.

4.1.2 Indeterminate Intervals

A *determinate* interval is the time between two instants. In our model of time, it is represented by a sequence of chronons, denoted by the starting and terminating chronons in the sequence.

An interval bounded by indeterminate instants (called the *starting* and *terminating instants*) is termed an *indeterminate interval*. An indeterminate interval could start during any member of the set of possible chronons of the starting instant. Likewise, the inde-

terminate interval could end during any member of the set of possible chronons of the terminating instant. Since the location of the starting and terminating instants are known only imprecisely, it follows that it is unknown precisely when an indeterminate interval begins or ends.

An indeterminate interval represents a set of *possible (determinate) intervals*, one of which is the "real" interval, but which is unknown. A single possible interval is obtained by choosing one possible chronon from each bounding indeterminate instant's set of possible chronons. Every combination of chronons in the starting and terminating instants' set of possible chronons is a possible interval.

Thus far we have only considered indeterminate intervals bounded by indeterminate instants. Since indeterminate instants can be used to model determinate instants, no special provisions are needed to handle determinate instants that serve as one or both bounding instants.

4.1.3 Indeterminate Spans

A *span* is an unanchored duration of time [Jensen et al. 1994]. It has a known length but no specific starting or ending chronons. For example, the span "6 days" is known to have a duration of six days, but can refer to any block of six consecutive days. A span can be either positive, denoting forward motion in time, or negative, denoting backwards motion in time.

A determinate span is a precisely known duration of time and is is represented as a count of chronons. An *indeterminate span*, on the other hand, is an imprecise duration that describes a set of possible durations. An indeterminate span is represented by an imprecise number of chronons, e.g., "from two to three chronons." Like the representation of an indeterminate instant, the representation of an indeterminate span has an associated probability mass function which gives the likelihood of each possible duration.

4.1.4 Indeterminate Tuples

The data model that we propose is an extension of the TQuel data model [Snodgrass 1987]. The TQuel data model supports *bitemporal* relations. A bitemporal relation may

be thought of as a trajectory of transaction-time states, each of which is a complete valid-time relation. The *transaction time-slice* operation on a bitemporal relation selects a particular transaction-time state, on which a valid-time query may be performed [Jensen et al. 1994]. For our purposes, we can assume that a bitemporal relation is embedded in a snapshot relation by adding two implicit attributes. The value of the first attribute specifies a valid time while the value of the second attribute specifies a transaction time. Thus, tuples in relations in the database are timestamped with both transaction and valid times. The valid-time attribute may be an instant or an interval while the transaction-time attribute is always an interval.

The granularity of a transaction-time timestamp is the smallest inter-transaction time. Transaction times are system-supplied and are always determinate since the time during which a transaction takes place is known. We will largely ignore transaction time in this paper. The example database (Figure 1.1) shows several valid-time relations with implicit valid-time attributes only.

The TQuel data model is an ungrouped data model [Clifford et al. 1994A] with tuple timestamping. Tuples in TQuel relations are "row-independent," that is, no information is shared between tuples. Since the indeterminate data model is based on TQuel, it makes no overt provisions for sharing indeterminate information between tuples. Such provisions would significantly increase the complexity of query processing. Instead, each tuple is independent of the other tuples in the relation, and no inter-tuple temporal requirements, such as the probabilities of the sets of possible chronons associated with a key must sum to one, are imposed. For example, assume that we have information that a plane engine is shipped from the Trump warehouse to the factory sometime between June 1 and June 30. Assume that this information is recorded in two different relations: *Shipped* and *Received*. The *Shipped* relation stores when parts are sent by a warehouse to the factory while the *Received* relation keeps track of when shipments are received at the factory. An inter-tuple constraint could be used to stipulate that parts are shipped prior to when they are received, however, such constraints can be expensive to enforce. In this case, a query could impose the constraint that parts be shipped before they are received.

	Warehouse	Lot_Num	Part	Valid time (at)
s_5	Trump	40	*unknown*	May 31
s_6	Griffin	70	*some electrical part*	May 31
s_7	Trump	41	{yoke, throttle}	May 31

Figure 4.2: Examples of value incompleteness

4.1.5 Other Kinds of Indeterminacy

In TQuel with indeterminacy, valid-time indeterminacy is orthogonal to other sources of incompleteness (as discussed in Chapter 2). In particular, it can peacefully coexist with *value incompleteness*, where the value of a nontemporal attribute is not fully known, and *tuple incompleteness*, where the membership of a tuple in a relation is not fully determined. For example, in the *Received* relation, a part may exist which we have yet to identify (s_5 in Figure 4.2), has been partially identified (s_6 restricts the kind of part to belong to the specified class of parts), or has been narrowed down to a set of possibilities (s_7). We advocate separating the various kinds of indeterminacy, so that users can choose the combination that is most appropriate for their application. We turn now from the data model to the query semantics.

4.2 Review of TQuel

In this dissertation we focus on extending TQuel's *retrieve* statement to support indeterminacy. Below we quickly review the syntax and semantics of TQuel's retrieve statement. The interested reader will find many examples as well as a complete description of the language elsewhere [Snodgrass 1987, Snodgrass 1993].

An example query that determines which `wing strut` shipments arrived during production of a `Centurion` airplane is shown in Figure 4.3. As discussed in Chapter 1, this query might be issued by a `Centurion` owner looking for the source of a faulty part. The retrieve has several components: the *target list*, specifying how the attributes of the relation being derived are computed from the attributes of the underlying relations;

```
range of r is Received
range of p is In_Production
retrieve (WH=r.Warehouse, L_Num=r.Lot_Num, S_Num=p.Serial_Num)
  valid at r overlap p
  where p.Model="Centurion" and r.Part="wing strut"
  when r overlap p
```

Figure 4.3: An example *retrieve* statement

a *valid clause*, specifying the valid time of tuples in the target relation; a *where clause*, specifying a relationship that must be satisfied among the explicit attributes (those visible to the user) of the participating tuples; a *when clause*, specifying a relationship among the valid-time attributes of the participating tuples; and an *as of* clause (not shown in Figure 4.3) that performs a transaction time-slice on the bitemporal database.

In the valid clause, a temporal expression consisting solely of *temporal constructors* specifies the valid time of tuples in the target relation. A temporal constructor chooses an instant or interval that satisfies some constructor-specific constraints. For example, the *First* temporal constructor chooses the earliest instant from a pair of instants. The temporal expression associated with the when clause is composed of temporal constructors, boolean connectives, and *temporal predicates*. A temporal predicate determines whether a pair of instants or intervals satisfies some specific constraint. For example, the *precede* predicate determines whether one instant (or interval) is earlier than another. If so, the predicate evaluates to "true;" if not, it evaluates to "false."

In the example query shown in Figure 4.3, pairs of `wing strut` shipment tuples and `Centurion` airplane tuples that overlap are candidates for the target relation. The valid clause constructs the valid time for the tuples in the target relation. In this case, the valid time of each target tuple is the valid time of the received `wing strut` shipment. The other attributes in the target relation are specified by the target list: the `Warehouse`,

the `Lot_Num`, and the `Serial_Num`. (An as of clause is used only when the query is evaluated on bitemporal or transaction-time relations.)

The semantics for TQuel associates a tuple calculus statement with each TQuel retrieve statement, ensuring that each construct has a clear and unambiguous meaning [Snodgrass 1987]. Tuple relational calculus statements are of the form $\{t^i \mid \Psi(t)\}$, where the variable t denotes a tuple of arity i and $\Psi(t)$ is a first-order predicate calculus expression containing only one free tuple variable, t. The kth attribute of tuple t is denoted $t.D_k$. The tuple calculus statement for the skeletal TQuel retrieve statement

range of t_1 *is* R_1

$\ldots$

range of t_k *is* R_k

retrieve $(t_{x_1}.D_{y_1}, \ldots, t_{x_r}.D_{y_r})$

valid at v

where ψ

when τ

is

1) $Reduce(\{u^{r+1} \mid (\exists t_1) \cdots (\exists t_k)\,(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$

2) $\wedge\ u[1] = t_{x_1}[y_1] \wedge \cdots \wedge u[r] = t_{x_r}[y_r]$

3) $\wedge\ u[r+1] = \Phi_v$

4) $\wedge\ \Psi'$

5) $\wedge\ \Gamma_\tau)\ \})$

Line 1 comes from the range statements. Line 2 is constructed from the target list. The symbol Φ_v, appearing in line 3, is a function over the valid-time attributes of a subset of the tuple variables. The function constructs a valid-time interval using the temporal constructors given in the valid clause. Line 4 is constructed from the where clause. Γ_τ,

appearing in line 5, is a predicate over the valid-time attributes of a subset of the tuple variables; it uses the temporal predicates and constructors given in the when clause. The *Reduce* function ensures that tuples with identical values for all explicit attributes, called *value-equivalent* tuples [Jensen et al. 1994], that overlap in valid time or are contiguous are *coalesced* into a single resulting tuple.

4.3 Syntactic Extensions to Support Valid-time Indeterminacy

This section describes the syntax for retrieving information from a database with valid-time indeterminacy; the next section provides a formal semantics for these constructs. A primary design goal in extending TQuel to support valid-time indeterminacy is to make a minimal extension. It will be shown in Section 4.4.5 that the new syntax and semantics preserves the meaning of all extant TQuel retrieve statements.

We make four syntactic extensions to TQuel: one to indicate that a relation is indeterminate, one to specify the range credibility, one to specify the ordering plausibility, and one to specify defaults.

Our first syntactic extension involves the schema specification statements. To the create statement we add the option of specifying that the valid-time timestamps may be indeterminate. Before the keywords `event` and `interval` a user may add the modifier `indeterminate` or `indeterminate general`. These options toggle between alternative storage strategies for indeterminate timestamps, discussed at length in Section 5.1 (the "general" version is a more expressive, less compact timestamp). We also add an optional "with" phrase to the end of the create statement that allows the user to specify *standard* or *nonstandard* mass functions. These two categories of mass functions are discussed in Section 5.1. The default is `with standard distribution`.

To the modify statement we provide clauses that allow instants or intervals to be specified as determinate or indeterminate, and to specify a kind of distribution that applies

to all the tuples in the relation. Below are some examples of the create and modify statements.

```
create indeterminate event Received(Warehouse: string;
                                     Lot_Num: integer;
                                     Part: string)
create indeterminate interval In_Production(Model: string;
                                     Serial_Num: string)
modify Received to nonstandard distribution
```

With the create or modify statement, it is also possible to specify the duration or probability mass function of an indeterminate instant intensionally. In that case, the upper support or mass function need not be stored; the upper support can be computed from the lower support and this duration while the mass function is the same for all the valid-time instants in the relation. The semantics of these extensions are straightforward. The intensional information concerning the indeterminacy of the timestamps is recorded in the system catalogue. The intensional specification of distributions is particularly helpful in optimizing both the representation and query processing, and is also of intuitive value to the user.

Range credibility appears in the range statement. The range credibility is the credibility in each valid time in the specified interval relation. The credibility applies independently to the starting and terminating instants in an interval. It can be any integer value between 0 and 100 (inclusive). The credibility phrase is optional and has an initial default value of 100. This default value can be changed using a set statement as follows.

```
set default range credibility to 50
```

The set statement is very useful when a group of queries is to be made at a particular credibility level, or when the credibility is to be specified for a novice. Range credibility is not applicable to event relations (an *event* relation is timestamped with *instants* [Jensen et al. 1994]) because removing indeterminacy from an indeterminate instant might require partitioning the instant's period of indeterminacy.

```
range of r is Received
range of p is In_Production with credibility 0
retrieve (WH=r.Warehouse, L_Num=r.Lot_Num, S_Num=p.Serial_Num)
  valid at r overlap p with plausibility 60
  where p.Model="Centurion" and r.Part="wing strut"
  when r overlap p with plausibility 60
```

Figure 4.4: An example query

Ordering plausibility is the plausibility in the temporal ordering of the instants that participate in the retrieval. The ordering plausibility may be specified either for the entire when clause or valid clause (or both). The ordering plausibility is specified with an integer between 1 and 100 (inclusive). The plausibility phrase is optional and has an initial default value of 100, which can be changed using a set statement.

An example query with optional credibility and plausibility phrases is shown in Figure 4.4. Intuitively, the query will determine, within the specified plausibility and credibility levels, which `wing strut` shipments were received during production of each `Centurion`. A credibility of 0 is specified in the query (this is also the default credibility). The selected credibility keeps all the indeterminacy in the underlying data. The retrieve statement in the figure has a plausibility value of 60 for the overlap temporal predicate in the when clause. When this query is applied to the database shown in Figure 1.1, the relation shown in Figure 4.5 is computed. The where clause selects the tuples from *Received* that are shipments of `wing struts` and the tuples from *In_Production* that are `Centurions`. The when clause selects pairs of `Centurion` and `wing strut` tuples that overlap with a plausibility of 60. Finally, the valid clause determines when the shipment of possibly defective parts was received. We will discuss in detail in Section 4.4.4 how this query is evaluated to obtain the indicated result.

Defective_Shipment_Candidates(WH, L_Num, S_Num)

WH	L_Num	S_Num	Valid time (at)		
Trump	23	AB33	May 10	∼	May 29
Griffin	30	AB33	May 30	∼	June 18
Griffin	31	AB34	June 13	∼	July 2

Figure 4.5: Result of the example query

4.4 Semantic Extensions to TQuel

The semantic extensions to support valid-time indeterminacy involve the redefinition of several existing functions and relations and the introduction of new functions. Specifically, we redefine the temporal ordering relation to support ordering plausibility, we introduce two "shrink" functions to effect range credibility, and we redefine the coalescing operator, *Reduce*. In subsequent sections we consider each of these modifications in some detail. It is important, however, to note that each function or relation that we redefine or add incorporates the determinate semantics. Support for valid-time indeterminacy is an extension of the determinate semantics rather than a replacement. Hence, the semantics of existing queries is left unchanged (this point is reiterated in Section 4.4.5).

4.4.1 Supporting Ordering Plausibility

To support ordering plausibility we redefine the ordering relation *Before*. The semantics of retrieve without indeterminacy is based on a well-defined ordering of the valid time instants in the underlying relations [Snodgrass 1987]. Every temporal predicate and temporal constructor refers to the ordering given by *Before* to determine the truth value of the predicate or the instant or interval returned by the constructor. A set of *determinate* instants has a single temporal ordering. Given a temporal expression consisting of temporal

predicates and temporal constructors, this ordering either satisfies the expression or fails to satisfy it.

A set of indeterminate instants, however, typically has many possible temporal orderings. Some of these temporal orderings are plausible while others are implausible. The user specifies which orderings are plausible by setting an appropriate ordering plausibility value. We stipulate that a temporal expression is satisfied if there exists a plausible ordering that satisfies it. This semantics reduces to that of the determinate case where there is only one ordering.

In the determinate semantics, *Before* is the "$<$" relation on instants. In the indeterminate semantics, the temporal ordering is based on the probability that one instant is before another. For any two indeterminate instants, α and β, the probability that α is before β is

$$\mathbf{Pr}[\alpha < \beta] = \sum_{i<j} \mathbf{Pr}[\alpha = i] \times \mathbf{Pr}[\beta = j] \quad i, j \in \{0, \ldots, N\}.$$

Figure 4.6 shows the value of $\mathbf{Pr}[\alpha < \beta]$ (to two decimal places) for each pair of instants in the relation *Received*. Those instants are placed on a time-line in Figure 4.7. For instance, $\mathbf{Pr}[e_2 < e_3] = .83$.

To handle indeterminate instants, we modify *Before* to include an additional initial parameter, the ordering plausibility, γ. The value of γ can be any integer between 1 and 100 (inclusive). In general, higher (closer to 100) ordering plausibilities stipulate that only highly probable orderings be considered plausible. The indeterminate *Before* is defined as follows.

$$\mathit{Before}(\gamma,\ \alpha,\ \beta) \iff \neg(\alpha\ is\ \beta) \wedge ((\mathbf{Pr}[\alpha < \beta] \times 100) \geq \gamma)$$

An instant is never *Before* itself, regardless of the value of γ. Two instants are said to be *representationally-equivalent* if they have the same support and the same probability mass functions. Two representationally-equivalent instants may or may not be *Before* one another, depending on γ. To distinguish representationally-equivalent instants, each instant appearing as an argument to *Before* is tagged with the tuple from which it originates. The tags are compared by the *Before* function. If the tags do not match, the binary infix operator $\mathbf{Pr}[\alpha < \beta]$ determines the discrete probability of one instant occurring "before" another.

$\mathbf{Pr}[\alpha < \beta]$	e_1	e_2	e_3	e_4
e_1		1.00	1.00	1.00
e_2	0		.83	.94
e_3	0	.13		.70
e_4	0	.03	.26	

Figure 4.6: Table of $\mathbf{Pr}[\alpha < \beta]$ for the indeterminate instants in *Received*

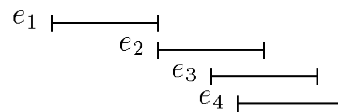


Figure 4.7: A pictorial representation of the instants in *Received*

This formulation of *Before* treats ordering probabilities that are between 0 and $.01$ as 0. That is, it treats two instants that have a small chance of occurring before each other as well-ordered in time. To distinguish the well-ordered case from this other case, we define the ordering probability to be $.01$ whenever its value is between 0 and $.01$. Hence, to evaluate every possible ordering, however improbable, an ordering plausibility of 1 suffices.

A mass function that is *missing* is treated specially. If one (or more) of the instants being ordered has a mass function that is missing, then the mass is assumed to be distributed in such a way that there is a small, but non-zero, probability of ordering the two instants. For instance, if we introduce the instant e_5 with a mass function that is missing, then $\mathbf{Pr}[e_2 < e_5] = \epsilon$ and $\mathbf{Pr}[e_5 < e_2] = \epsilon$. Consequently, in the semantics, an instant with a mass function that is missing behaves exactly like a *null* value, in that the participation of such an instant in a *Before* operation makes the *Before* false (for all plausibilities greater than 1). However, since there is a small probability that an instant with a missing mass is before another instant (when their periods of indeterminacy overlap), *Before* will be satisfied for an ordering plausibility of 1.

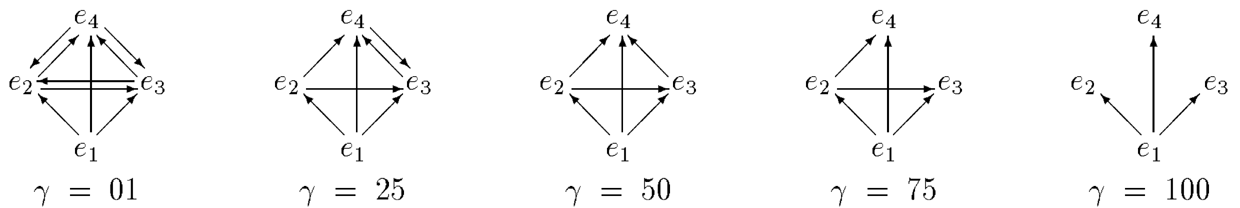


Figure 4.8: Ordering the events in *Received* depends on γ

Before assumes that there are no dependencies between the probabilities associated with indeterminate instants. Hence, it cannot be used to accurately compute the probability of orderings such as **Pr**$[(\alpha < \beta < \eta)]$. In the expression "($\alpha$ `precede` β) `and` (β `precede` η)" the two `precedes` are separately evaluated, returning boolean values that are subsequently anded. While this evaluation strategy is consistent with the determinate semantics, it is not equivalent to computing an ordering of $(\alpha < \beta < \eta)$.

The ordering relation among the instants in the relation *Received* depends on the ordering plausibility, γ. The orderings given by differing values of γ are graphically depicted in Figure 4.8. Each directed edge in a graph indicates that the originating instant is *Before* the terminating instant. Some pairs of instants are "indistinguishable," that is each occurs *Before* the other. If no edge connects two instants, the instants are "incomparable," neither occurs *Before* the other. From the definition, it can be shown that *Before*, for $\gamma \neq 100$, is not a typical ordering relation in that it is not transitive nor anti-symmetric, although it is always irreflexive (*Before* for $\gamma = 100$ is transitive, anti-symmetric, and irreflexive). The following example demonstrates that the ordering relation is not transitive. Consider the indeterminate instant $\alpha = (|1 \sim 7|, \text{uniform})$ and the determinate instants $\beta = |2|$ and $\omega = |4|$. There is a nonzero probability that ω is before α, so *Before*$(1, \omega, \alpha)$. There is also a nonzero probability that α is before β, so *Before*$(1, \alpha, \beta)$. However, ω is not before β.

A generalization of *Before* is *Set_Before*. *Set_Before* is used below in the redefinition of various temporal constructors and predicates. *Set_Before* is similar to *Before*, but operates

on sets of instants.

$$\textit{Set_Before}(\gamma,\ \alpha,\ \beta) \iff \forall x \in \alpha\, \forall y \in \beta\ \textit{Before}(\gamma,\ x,\ y)$$

Set_Before stipulates that the set of instants, α, is before the set of instants, β, if every instant in α is before every instant in β, to the specified ordering plausibility.

The new ordering relations are used to redefine the temporal constructors and predicates. Below, we consider the *First* constructor in some detail since *First* is used in other constructors. Recall that *First* chooses the earliest instant among a pair of instants. With indeterminate instants, choosing the earliest instant among a pair of instants is not always straightforward. In particular, for a given ordering plausibility, it could be that neither instant in a pair or instants is earlier, or it could be that both are earlier. In the indeterminate semantics,

$$\textit{First}(\gamma,\ \alpha,\ \beta) \ = \begin{cases} \alpha & \text{if } \textit{Set_Before}(\gamma,\ \alpha,\ \beta) \\ \beta & \text{if } \textit{Set_Before}(\gamma,\ \beta,\ \alpha) \\ \eta - \delta & \text{otherwise, where} \\ & \quad \eta = \alpha \bigcup \beta \text{ and} \\ & \quad \delta = \{x |\ x \in \eta \wedge \neg \exists y \in \eta\ (\textit{Before}(\gamma,\ y,\ x))\} \end{cases}$$

To simplify discussion of *First*, consider the case where α and β each contain a single indeterminate instant. Determining which instant occurs first has several possible outcomes:

- only α is first,
- only β is first,
- both α and β are first (each is before the other; the instants are indistinguishable), or
- neither α nor β is first (neither is before the other; the instants are incomparable).

The first two outcomes are straightforward. The third outcome, that for indistinguishable instants, is handled by the fact that *First* is nondeterministic; each instant is generated separately and may result in a separate output tuple. Alternatively, a set of possible outcomes can be maintained, generating with each element in the set generating a separate output tuple. Only plausibilities below 50 can generate indistinguishable instants. For the final possible outcome, since neither instant is before the other, *First* constructs the set containing both instants. Other temporal constructors and temporal predicates will treat the set as a set of instants with no *Before* relationships between the members. In general, all members in such sets are pairwise incomparable. Below we show several temporal expressions composed of the *First* constructor and the result of each expression using the instants from the relation *Received*.

$\textit{First}(50,\ \{e_2\},\ \{e_3\}) = \{e_2\}$ (α is first)

$\textit{First}(100,\ \{e_2\},\ \{e_1\}) = \{e_1\}$ (β is first)

$\textit{First}(1,\ \{e_2\},\ \{e_3\}) = \{e_2\}$ then $\{e_3\}$ (both α and β are first,
(the operation generates both instants)

$\textit{First}(100,\ \{e_2\},\ \{e_3\}) = \{e_2,\ e_3\}$ (α and β are incomparable)

The *First* constructor can deduce the first instant among a group of instants, even when some of those instants are incomparable (e_2, e_3, and e_4 are incomparable for a plausibility of 100 as shown in Figure 4.8), for example:

$$\textit{First}(100,\ \{e_1\},\ \textit{First}(100,\ \{e_2\},\{e_3\})) = \{e_1\}$$
$$\textit{First}(100,\ \{e_2\},\ \textit{First}(100,\ \{e_3\},\{e_1\})) = \{e_1\}.$$
$$\textit{First}(100,\ \textit{First}(100,\ \{e_2\},\ \{e_1\}),\ \textit{First}(1,\ \{e_3\},\ \{e_4\})) = \{e_1\}$$

The *First* constructor also works when some of the instants are indistinguishable (e_2, e_3, and e_4 are indistinguishable for a plausibility of 1), for example:

$$\textit{First}(1,\ \{e_1\},\ \textit{First}(1,\ \{e_2\},\{e_3\})) = \{e_1\}$$
$$\textit{First}(1,\ \{e_2\},\ \textit{First}(1,\ \{e_3\},\{e_1\})) = \{e_1\}$$
$$\textit{First}(1,\ \textit{First}(1,\ \{e_2\},\ \{e_1\}),\ \textit{First}(1,\ \{e_3\},\ \{e_4\})) = \{e_1\}.$$

The redefinition of the *Last* temporal constructor is similar to that of *First* and is omitted to save space. The definitions of the other temporal constructors change little; a

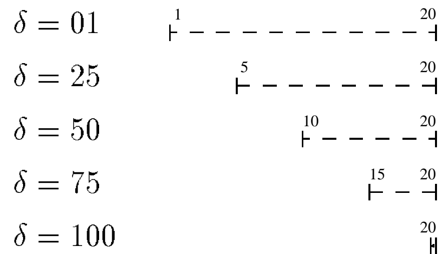


Figure 4.9: *Shrink_s*$(\delta, (|1 \sim 20|, Uniform))$ for several values of δ

parameter for the plausibility is added to each, e.g.,

$$overlap(\gamma, \langle \alpha, \beta \rangle, \langle \eta, \delta \rangle) \quad = \quad \langle Last(\gamma, \alpha, \eta), First(\gamma, \beta, \delta) \rangle.$$

Contrast this with the determinate semantics for the overlap constructor:

$$overlap(\langle \alpha, \beta \rangle, \langle \eta, \delta \rangle) \quad = \quad \langle Last(\alpha, \eta), First(\beta, \delta) \rangle.$$

We are now in a position to redefine the temporal predicates. These definitions differ only slightly from the determinate semantics. A plausibility parameter is added to each predicate and *Set_Before* replaces *Before* since the temporal constructors now build sets of instants rather than instants. For example, in the determinate semantics,

$$precede(\langle \alpha, \beta \rangle, \langle \eta, \delta \rangle) = Before(Last(\alpha, \beta), First(\eta, \delta)),$$

while in the indeterminate semantics it is

$$precede(\gamma, \langle \alpha, \beta \rangle, \langle \eta, \delta \rangle) = Set_Before(\gamma, Last(\gamma, \alpha, \beta), First(\gamma, \eta, \delta)).$$

4.4.2 Supporting Range Credibility

Range credibility changes the data that is available for query evaluation. In general, range credibility is used to eliminate some possible intervals from an indeterminate interval. It does so by eliminating some possible chronons from both the starting and terminating instants' set of possible chronons. To support range credibility we introduce two "shrink" functions: *Shrink_s* (shrink the *s*tarting instant) and *Shrink_t* (shrink the

*t*erminating instant). The shrink functions compute a "shortened" version of an indeterminate instant by shrinking its period of indeterminacy and modifying its probability mass function.

Shrink_s computes a "later" period of indeterminacy by removing some of the "earlier" chronons from the set of possible chronons. How many chronons to remove is governed by the first argument, δ, the range credibility. The value of δ is between 0 and 100 (inclusive). Every possible chronon that has a cumulative probability less than the level of credibility is removed. Higher values (closer to 100) of δ will remove more chronons from the set. *Shrink_s*$(100, \alpha)$ will remove every chronon except the latest possible chronon in α. *Shrink_s*$(0, \alpha)$ will leave α unchanged. Figure 4.9 shows the result of *Shrink_s* for several credibility values on the indeterminate instant $\alpha = (|\texttt{1} \sim \texttt{20}|, Uniform)$. *Shrink_s* is defined as follows.

$$\textit{Shrink_s}(\delta,\ (|\,\alpha_* \sim \alpha^*\,|,\ P_\alpha)) = (|\,x \sim \alpha^*\,|,\ P'_\alpha)$$

where x is constrained by

$$\begin{aligned}&(\alpha_* \leq x \leq \alpha^* \wedge F_\alpha(x) \geq \delta)\\ &\qquad \wedge \neg(\exists i)(x < i \leq \alpha^* \wedge F_\alpha(i) = F_\alpha(x)\\ &\qquad \wedge \neg(\exists j)(\alpha_* < j < x \wedge F_\alpha(x) > F_\alpha(j) \geq \delta)\end{aligned}$$

and P'_α is the new mass function, $P'_\alpha(i) = \frac{P_\alpha(i)}{1-F_\alpha(x)}$. $F_\alpha(x)$ is the *cumulative distribution function* for the probability mass function.[1] Intuitively, the conditions on *Shrink_s* stipulate that the desired chronon is in a group of chronons with the same cumulative probability (the cumulative probability is the chance that the chronon is before or during the chronon in question). This group is the latest group such that the cumulative probability of all the chronons earlier than the group falls below δ while the cumulative probability of each chronon within the group matches or exceeds δ. The desired chronon is the latest chronon within this group. It is the latest rather than an arbitrary chronon so that repeated shrinks will make progress.

The function must also compute a new probability mass function since the old mass function might have assigned nonzero probability to times that are no longer in the period

[1] The cumulative distribution function is $F_\alpha(i) = \mathbf{Pr}[\alpha \leq i] = \sum_{k \leq i} P_\alpha(k)$.

of indeterminacy. To construct the new mass function, the probability of each of the remaining times is scaled by the cumulative probability of the chopped times. The new mass function is a *conditional* probability function. That is, the probabilities are conditioned by the fact that the period of indeterminacy is shrunk.

Shrink_t is similar to *Shrink_s*, but it removes the "late" instants from an instant's period of indeterminacy. The definition of this function is similar to that of *Shrink_s*.

With these two functions, it is possible to define the temporal constructor consisting entirely of a tuple variable associated with an interval relation.

$$
\mathit{interval}(\gamma, \delta,\ t) \;=\; \begin{cases} \langle\{\mathit{Shrink_s}(\delta,\ t_{from})\},\ \{\mathit{Shrink_t}(\delta,\ t_{to})\}\rangle & \text{if } \mathit{Before}(\gamma, \mathit{Shrink_s}(\delta, t_{from}), \\ & \qquad \mathit{Shrink_t}(\delta, t_{to})) \\ \text{no interval} & \text{otherwise} \end{cases}
$$

This function extracts the *from* timestamp from the tuple, shrinks it by δ to create a "later" set of possible chronons, extracts the *to* timestamp from the tuple, and shrinks it by δ to create an "earlier" set of possible chronons, thereby effecting the range credibility. If $\delta = 100$, then all valid-time indeterminacy will be eliminated. The function then constructs an interval consisting of the pair of the starting instant and the terminating instant, each perhaps indeterminate, but only if the starting instant precedes the terminating instant at the chosen plausibility.

The semantics of the temporal constructor consisting solely of a tuple variable associated with an instant relation changes little. We represent the instant α (determinate or indeterminate) as the pair $\langle\{\alpha\},\ \{\alpha\}\rangle$, to simplify the semantics of the temporal constructors and predicates.

4.4.3 Coalescing

Tuples in TQuel relations are assumed to be coalesced, in that value-equivalent tuples (i.e., tuples with identical values for the explicit attributes) neither overlap nor are adjacent in determinate valid time. However, the tuples could overlap in indeterminate valid time. The valid times associated with the value equivalent tuples are coalesced into a *temporal*

element by the *Reduce* function. Essentially, a temporal element is a set of non-overlapping intervals [Jensen et al. 1994]. A new function, *Reduce*$'$, computes the minimal set of value-equivalent indeterminate tuples, i.e., the set for which there are no such tuples.

The behavior of *Reduce*$'$ can be made clear by examining the process of adding a new interval to a temporal element. We add an indeterminate interval, i, to a temporal element, t' as follows. Sequentially consider each interval, j, in t'. If the determinate portions of i and j overlap then make one interval by gluing i and j together; this will eliminate one delimiting indeterminate instant from each interval. The resulting interval becomes i. Continue until all the intervals in the temporal element have been considered.

4.4.4 Semantics of the Example Query

As a review, the following is the tuple calculus semantics of the query in Figure 4.4 on page 62, using a range credibility of 0 and an ordering plausibility of 60.

1) $\textit{Reduce}'(\{u^{3+2} \mid (\exists r)\ (\exists p)\ (Received(r)\ \wedge\ In_Production(p)$

2) $\wedge\ u[1] = r[1]\ \wedge\ u[2] = r[2]\ \wedge\ u[3] = p[2]$

3) $\wedge\ u[4] = \textit{Last}(60,\ r[4],\ \textit{Shrink_s}(0,\ p[3]))$

$\wedge\ u[5] = \textit{First}(60,\ r[4],\ \textit{Shrink_t}(0,\ p[4]))$

4) $\wedge\ r[2] =$ “wing strut" $\wedge\ p[1] =$ “Centurion"

5) $\wedge\ \textit{Set_Before}(60,\ \textit{Last}(60,\ r[4],\ \textit{Shrink_s}(0,\ p[3])),$

$\textit{First}(60,\ r[4],\ \textit{Shrink_t}(0,\ p[4])))$

6) $\wedge\ \textit{Before}(60,\ \textit{Shrink_s}(0,\ p[3]),\ \textit{Shrink_t}(0,\ p[4]))\})$

We have chosen this example because it illustrates both a temporal constructor and a temporal predicate. The `overlap` appearing in the where clause (a predicate) generates the *First* and *Last* constructors and the *Set_Before* on line five. Line six is generated by the interval constructor constraint. The valid clause generates line three. The default range credibility generates the *Shrink* function invocations.

At this point, the semantics of the retrieve statement have been specified. As an example, we trace the computation of the query given in Figure 4.4 on the database given in Figure 1.1. The query will result in three tuples, shown in Figure 4.5. First, the extent of the intervals in *In_Production* is unchanged by the shrink functions because the query uses a range credibility of 0. The where clause eliminates every tuple from *In_Production* except the Centurions. Likewise, the where clause also eliminates every tuple from *Received* except the wing strut tuples.

The shipment of lot number 23 was definitely received during production of the Centurion serial number AB33; it satisfies the overlap with every plausibility. The other shipments might have been received. Lot number 30 satisfies the overlap for plausibilities lower than 60 because |`May 30` ~ `June 18`| is before |`June 1` ~ `June 30`| for every ordering plausibility below 65. The other shipment, however, arrived too late in June to be considered plausible. It is plausible that lot number 31 arrived before the end of production only for ordering plausibilities of 28 or less. For production of the Centurion serial number AB34, all the shipments arrived too early, except for lot number 31 from the Griffin warehouse.

4.4.5 Query Reducibility

An important feature of the extended syntax and semantics is that evaluation of a retrieve statement using the default plausibility and credibility (both 100) on a valid-time database with indeterminate or determinate interval relations and determinate event relations is equivalent to evaluation of the retrieve statement with the previous semantics (which has no support for valid-time indeterminacy) on the corresponding "interval reduced" database without valid-time indeterminacy. We will call this property *query reducibility*. By an *interval reduced* database, we mean a database in which the interval indeterminacy has been removed by replacing each indeterminate interval with its determinate portion via shrinking by a range credibility of 100. Query reducibility shows that the meaning of all extant TQuel queries and relations is preserved under the new semantics. It also shows that even if there is some indeterminacy in the database (i.e., if there are indeterminate interval relations), the user can choose to ignore it (this is the default choice).

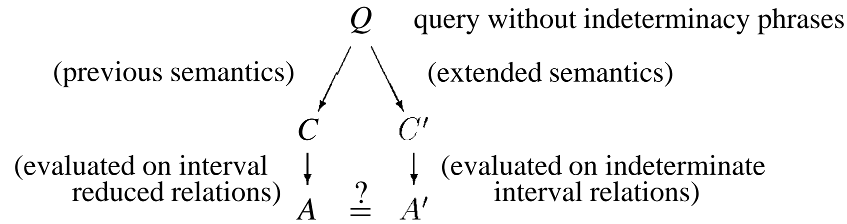


Figure 4.10: Demonstrating Query Reducibility

More specifically, under the extended semantics, a retrieve statement without credibility or plausibility phrases, *Q*, will be translated to a tuple calculus statement, *C*, of the form described earlier in this paper, assuming the credibility and plausibility defaults. Under the previous semantics, *Q* will be translated to a tuple calculus statement, C', of the form discussed previously. We claim that if every event relation participating in *Q* is determinate (but every interval relation need not be), then *C* is query reducible to C', that is, evaluation of *C* is equivalent to evaluation of C'. The claim is diagramed in Figure 4.10.

Theorem 1 *The extended semantics is query reducible to the previous, valid-time determinate, semantics.*

PROOF. The outline of what to prove is illustrated in Figure 4.10. The only differences between *C* and C' are the new and redefined functions *Before*, *Shrink_s*, *Shrink_t*, and *Reduce*′. We will show that when working with determinate tuples the redefined functions in the extended semantics reduce to their definitions in the previous semantics, and the new functions have no effect.

We observe that a determinate tuple is timestamps with a determinate instant (or interval). A determinate instant has a period of indeterminacy that is a single chronon. All determinate instants have the same probability distribution; the probability of that single chronon is 1. Finally, the default credibility and plausibility are both 100.

The shrink functions have no effect on determinate instants. Since a chronon is the smallest granule of time, a determinate instant cannot be shrunk any further. Thus,

evaluation of any of the shrink functions on a determinate instant will return the same instant regardless of the range credibility.

The default ordering plausibility of 100 will select the determinate ordering of the instants in the underlying relations to evaluate temporal expressions. For any pair of determinate instants, either one instant is before the other with probability 1 or the instants are represented by the same chronon, and neither is before the other. An ordering plausibility of 100 serves to establish the well-ordering of instants represented by different chronons. Hence the determinate *Before* operation on determinate instants reduces to that of the indeterminate *Before* operation on determinate instants at a plausibility of 100

Finally, the new definition of $Reduce'$ coalesces value-equivalent tuples that are adjacent or overlap in determinate time exactly as the old definition. ∎

CHAPTER 5

IMPLEMENTATION OF INDETERMINACY

Changes to the semantics to support valid-time indeterminacy trigger changes in the implementation. These changes are isolated to the representation of instants, spans, and intervals, and to the new or modified temporal operators: Reduce', *Before*, *Set_Before*, *Shrink_s* and *Shrink_t*. In the next two sections we describe the data structures and algorithms to implement these new or modified operators. Our goal is to provide support for valid-time indeterminacy without adversely impacting storage requirements or query evaluation efficiency.

At first glance, support for valid-time indeterminacy appears to be expensive. For example, *Shrink_s* must compute the new probability for each chronon in the shrunken set of possible chronons in a terminating indeterminate instant. Unfortunately, there could be quite a large number of possible chronons: a typical indeterminate instant with a one-day period of indeterminacy stored to the granularity of a microsecond has over 86 *million* possible microseconds. In addition, some of the modified operators, e.g., *Before*, are executed in the "inner loop" of query processing, potentially performed many times for each combination of tuples in the queried relations. Significant slowdown of these operators would have a dramatic effect on the overall speed of query evaluation.

Although *Shrink_s* and the other operators appear costly, we show below how the new operators can be implemented efficiently. We begin with the representation of indeterminate information. In particular, we describe timestamp formats that compactly store valid-time indeterminate instants, intervals, and spans. We then show how the new operators can be implemented efficiently, focusing on the probabilistic ordering used in *Before*.

5.1 Indeterminate Timestamp Formats

Valid-time indeterminate instants, intervals, and spans model new kinds of temporal information. To represent indeterminate temporal values, new temporal data types, or *timestamps* [Jensen et al. 1994], are needed. In this section we describe the indeterminate timestamps in detail. First, we present the indeterminate instant timestamps. Next, we show that the span and interval timestamps are natural extensions of the instant timestamps. Finally, we motivate the decisions we made in designing the timestamps.

5.1.1 Instant Timestamp Formats

The instant formats described here build upon the determinate instant format. We briefly review this format here; a full description is given elsewhere [Dyreson & Snodgrass 1993B, Dyreson & Snodgrass 1994B].

5.1.1.1 Determinate Instants

An instant timestamp must meet several requirements. First, the timestamp must support a multitude of *ranges*. Range is a measure of how much of a time-line can be represented. A timestamp should be capable of storing times that range over just a few seconds to those that range over the age of the universe. Second, it must support a variety of *granularities* [Jensen et al. 1994], from those as large as a year to those as small as a femtosecond. In this paper, we have considered only a single granularity, that of chronons; in general, multiple granularities should be supported [Dyreson & Snodgrass 1994C]. Third, the timestamp must be capable of storing times both before and after a granularity anchor (an *anchor* is a point on the time-line). Finally, since it is difficult to anticipate the demands of future language designers, the format must allow for growth, primarily the addition of new timestamp types.

The determinate instant timestamp is shown in Figure 5.1. The dashed lines mark word boundaries. The number above a subfield is the size of that subfield, in bits. The size of the timestamp, in particular, the size of the *data field* varies depending on the range and granularity of the timestamp; the data field is either 28, 60, or 92 bits in size. We assume

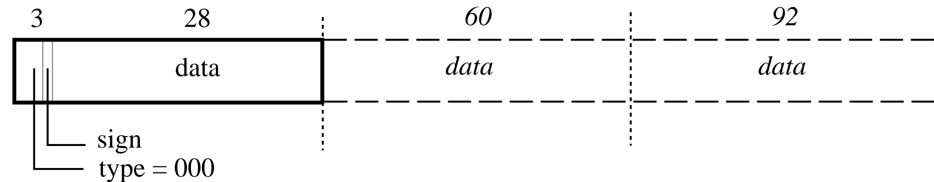


Figure 5.1: The standard instant format

that the range and granularity are specified for the instant as part of a create or modify statement and stored with the schema. Larger ranges and finer granularities require more bits to represent. The one, two, and three word formats are depicted in the figure. Two words should be sufficient for most applications. The two word timestamps can store a range of historical times to the granularity of a microsecond, or times within a range of 36 billion years (all of time, back to the big bang) to the granularity of a second. Three words, which can represent times over a range of 36 billion years to a nanosecond, should take care of the few remaining applications.

An instant is located on the time-line at some distance from the anchor. The instant format stores the distance from the anchor as a signed number. The number is stored in the data field and is a count of units of a given granularity, that is, a count of chronons. It is important to note that the range and granularity of an instant timestamp are not stored in the timestamp. Hence, the interpretation of the data field requires information from the schema, provided by the query processor. The sign bit field indicates whether the instant is before or after the anchor.

To differentiate amongst the timestamp formats, each format has a type field. The type field is stored in the high order portion rather than the low order portion because not every format is the same size. The type field distinguishes special instants, such as “beginning” and “forever,” from other instants. Beginning is the youngest possible time while forever is the oldest.

FORMAT	*Starting Time*	*Terminating Time*	*Probability Distribution*
Determinate	explicit	implicit	implicit
Chunked, Standard	explicit	implicit	implicit
Chunked, Nonstandard	explicit	implicit	explicit
General, Standard	explicit	explicit	implicit
General, Nonstandard	explicit	explicit	explicit

Table 5.1: Encodings in the indeterminate formats

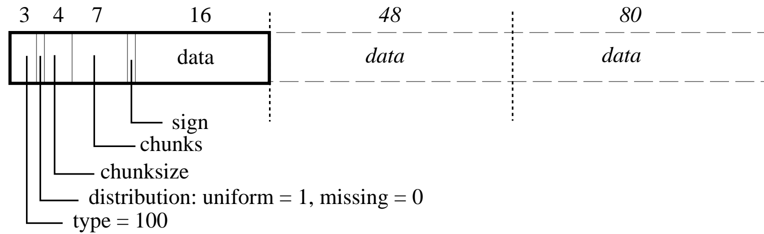
5.1.1.2 Indeterminate Instants

Support for indeterminate instants greatly compounds the complexity of the representation because an instant is no longer just a single time, rather it is two times and a probability mass function. In the worst case, the mass function information alone adds an extra word to the representation.

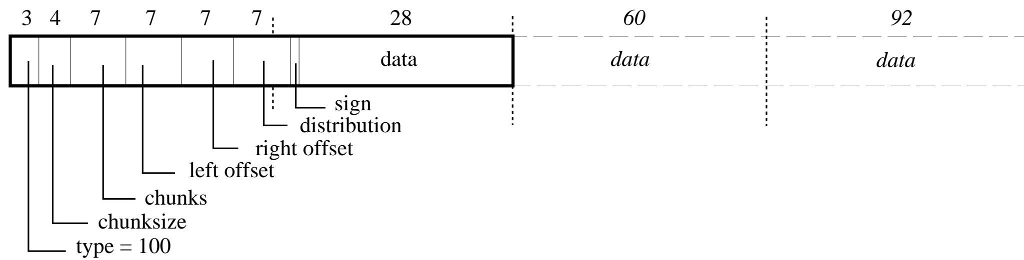
To represent indeterminate instants, we add four new formats, shown in Figure 5.2. The type field of every format is the same, indicating that this timestamp is for an indeterminate rather than a determinate or other kind of instant. The format that is used is specified by the user as part of creating a relation. The first format from the top depicted in Figure 5.2 is the default format. The second format is used if the user wishes a "nonstandard" distribution; the standard distributions are the uniform mass functions and that the mass function is missing. Nonstandard distributions add an extra word of information to the format. If the user wants a "general" indeterminate instant, to be described shortly, one of the last two formats is used.

The formats appear to be very different, but they are all fundamentally alike. Each timestamp format has the three basic parts needed to describe an indeterminate instant, as described in Section 4.1.1: a *lower support*, an *upper support*, and a *probability mass function*. These three parts are encoded in the timestamp either implicitly or explicitly. Table 5.1 indicates for each format whether the representation is explicit or implicit. For example, the determinate format (shown in Figure 5.1) has an explicit lower support, but an

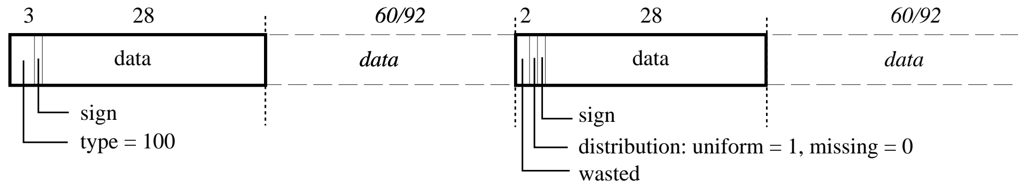
Indeterminate, Chunked, Standard Distribution (32/64/96/128 bits)



Indeterminate, Chunked, Nonstandard Distribution (64/96/128 bits)



Indeterminate, General, Standard Distribution (64/128/192 bits)



Indeterminate, General, Nonstandard Distribution (96/160/224 bits)

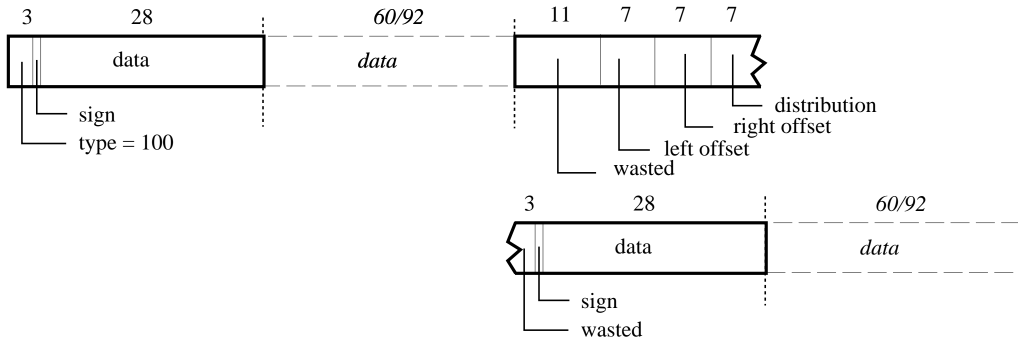


Figure 5.2: The indeterminate formats

implicit upper support (identical to the lower support). The lower and upper supports are represented to the range and granularity of the timestamp. As with determinate instants, the range and granularity of indeterminate instants are stored with the schema rather than with each timestamp, and the size of the data field varies depending on the range and granularity. If the determinate format has a granularity of microseconds then the lower support is the represented microsecond and the upper support is the same microsecond. The period of indeterminacy, one microsecond, is implicit, as is the probability mass function, which is assumed to be the uniform distribution. It is in this sense that every timestamp stored in the database is indeterminate.

Sometimes the upper support is also encoded implicitly. If it is encoded implicitly then it it composed of a *chunk size* and a number of *chunks*. Intuitively, chunks are used to support coarse-grain units efficiently. A chunk is a span, the length of which is specified by the chunk size field. The upper support is computed by adding the number of chunks, each of size *chunk size*, to the lower support. For example, to represent a period of indeterminacy of seven hours using chunks, the timestamp would record that there are seven hour-sized chunks. The chunk sizes that can be used depend on the granularity of the timestamp. Chunk sizes smaller than the granularity cannot be used since the timestamp cannot store these times. One of the duties of the database implementor is to specify chunk size tables, one for each granularity.

By making the number of units in a chunk exponential in the *chunk size* (e.g., by having the chunk size jump by a factor of between 10 and 100), it is generally possible to cover a large portion of the range with 16 chunk sizes.

Chunks and chunk sizes are either recorded explicitly or implicitly in the timestamp format. Every timestamp that does not explicitly encode a particular upper support uses a chunking scheme to compute the upper support. For example, the determinate format has an implicit upper support. Furthermore it has an implicit chunking scheme. It is assumed to have a single chunk with a chunk size of one unit in the granularity of the timestamp.

In general, the (timestamp) representation of a probability mass function has three parts, the name of a mass function, a *left offset*, and a *right offset*, occupying somewhat less than 32 bits in toto. The name of the mass function is stored with the timestamp

but the actual mass function is stored separately as described further in Section 5.2.1.2. Indeterminate instants that have the same probability mass function and period of indeterminacy may still differ since some of the mass may have been removed through the machinations of the *Shrink* operators as described further in Section 5.2.2. The left and right offsets are the percentage that has been removed from the "early" and the "late" portion of the period of indeterminacy. Instants with a uniform distribution or distribution that is missing, termed the *standard* distributions, do not need to keep track of the left and right offset since these distributions are independent of the offset. Consequently, we optimized representation of the standard distributions, which are one word shorter than the nonstandard distributions. The user specifies the kind of distribution, standard or nonstandard, when creating or modifying a valid-time relation as described in Section 4.3. When the relation is created, the standard distributions are assumed.

The chunking scheme and the use of standard distributions yield a compact timestamp. SQL-92's limited `TIMESTAMP` format without fractional seconds and without *indeterminacy* (assuming that the *positions* in the SQL-92 timestamp are nibbles) is 56 bits. Our indeterminate chunked timestamp with the same range and granularity as the SQL-92 datetime format requires only 64 bits.

5.1.2 Indeterminate Intervals and Spans

Like determinate intervals, indeterminate intervals are represented using two instant timestamps; one for the starting instant and one for the terminating instant. Since indeterminate instant timestamps are sometimes bigger than determinate instant timestamps, some of the indeterminate interval formats are larger than their determinate brethren; the largest indeterminate interval timestamp is fourteen words. However, we anticipate that the four word indeterminate interval formats will be the most common.

Many common spans are of indeterminate duration. For example the typical response as to when the garbage will be carried out is "in about five minutes," which actually represents a span of between five minutes and several days. Just as determinate spans use the same formats as determinate instants [Dyreson & Snodgrass 1994B], indeterminate

spans use the same representational formats as indeterminate instants. The format is interpreted as a span rather than an instant.

5.1.3 Design Decisions

The design of the indeterminate timestamp formats optimizes representation of the common mass functions (the standard mass functions cost only a single bit). The design also optimizes encoding of the upper support via chunking. To store the upper and lower support (two separate times) in a single timestamp, it would appear that we would require at least two data fields for an indeterminate instant. But we expect that arbitrary periods of indeterminacy will be rare. What will be common are periods such as N hours, N days, or N years. We also expect that precise knowledge of starting and terminating times for large periods of indeterminacy will be rare. For example, it would be very uncommon for a user to know that an instant occurred sometime between 6:23:43.003 A.M. July 23, 1985 and 3:00:57.23409 August 15, 1990. It is more likely that the user knows it occurred sometime between July 1985 and August 1990. The user specifies whether the indeterminate instants in a relation are chunked or unchunked when a relation is created. The chunked formats are the default. The unchunked formats result from a user adding the modifier "general" when creating an indeterminate instant relation as described in Section 4.3.

The chunking scheme was developed to meet the expectation that regular periods of indeterminacy will the norm. Chunking is a very efficient method of encoding a terminating time; the encoding only occupies eleven bits. But the space efficiency comes at the expense of some run-time computation since the terminating time must be computed on the fly. The computation costs one addition to add the chunks to the lower support. Another cost is that many periods of indeterminacy cannot be represented using the chunking scheme. For example, we cannot represent a period of 3 hours and 46 minutes using 1 minute chunks (the maximum period in this case is 2 hours and 8 minutes). However, a chunk size of 10 minutes would permit an approximation of 3 hours and 50 minutes. While users specify which timestamps are chunked, when a timestamp is translated from a string constant and stored on disk, it is the responsibility of the

database system to determine the chunk size and number of chunks that best fits the user's timestamp constant. We anticipate than an exact match can be found in most cases since the database implementor will select chunk sizes that are natural periods of indeterminacy (e.g., an hour, a week, a year, etc.). A good system will also give feedback to the user in those cases where the stored data only approximates, because of chunking, the specified temporal constant.

5.2 Implementation of Operators

In this section we discuss the implementation of *Before*, *Set_Before*, *Shrink_s*, *Shrink_t*, and *Reduce*$'$, beginning with the most common new operation: the temporal comparison predicate *Before*.

5.2.1 The *Before* Operation

We observed in Section 4.4 that the semantics of temporal constructors and predicates such as `overlap` and `first` are ultimately based on *Before*. If the instants being compared by *Before* are determinate, that is, if we know precisely when or during which chronon they are located, then *Before* is the "$<$" relation on the integers and its implementation is a single integer comparison.

Indeterminate instants complicate the implementation of *Before*. In the indeterminate semantics, deducing that one instant is before another may require computing the *probability* that one instant is before the other. This is a potentially costly computation. We show below how this computation can be made efficient.

5.2.1.1 The Common Interface

The interface to the *Before* routine is given in Figure 5.3. The interface determines if the relatively costly computation of the ordering probability can be avoided. If α and β are the same instant, then *Before* is trivially false since an instant cannot be earlier than itself. *Before* is also trivial if α's and β's periods of indeterminacy are disjoint. Disjointness implies that one instant is before the other in all possible cases. We anticipate that disjoint

function *Before*(**in** α, β : instant; **in** γ : integer) : boolean;

begin

if α *is* β **then return** FALSE

else if $\alpha^* < \beta_*$ **then return** TRUE

else if $\beta^* \leq \alpha_*$ **then return** FALSE

else if $\gamma = 100$ **then return** FALSE

else if $\gamma = 1$ **then return** TRUE

else return *PROB_*α*_BEFORE_*β(α, β, γ)

end; { *Before* }

Figure 5.3: Interface to *Before*

periods of indeterminacy will be common. Even if the periods of indeterminacy overlap, some ordering plausibilities can be trivially satisfied. An ordering plausibility of 100 can only be met if the periods of indeterminacy are disjoint while an ordering plausibility of 1 is attained if the periods overlap at all. Again, we anticipate that these will be common cases, representing the definite and possible orderings, respectively. If no special case applies, then the ordering probability, **Pr**$[\alpha < \beta]$, must be calculated.

5.2.1.2 Probability Mass Function Representation

The algorithm for **Pr**$[\alpha < \beta]$ presented below uses the probability mass function. In this section we describe a data structure to store that function. We present the data structure first since it impacts the algorithm design.

In general, a function can either be computed on the fly or precomputed and its values cached, say, in an array. The latter strategy is best for a probability mass function. *Before* is executed in the "inner loop" of query processing, performed many times during a query. We anticipate that many useful probability mass functions are not easily computable functions, making computing values on the fly expensive in terms of execution time, whereas table-lookup is quite cheap, although potentially expensive in terms of space.

Storing a probability mass function in an array, however, is not straightforward. If the function is precomputed and cached, the domain of the function gives the number of elements in the array while the range stipulates the size of each element. A probability mass function cannot be stored directly in an array since its domain is the integers (or reals) and its range is the real number interval [0,1].

To make storage costs reasonable, the distribution given by a probability mass function must be *approximated*. We approximate a mass function as follows. First, the mass is *quantized*; that is, it is parceled into indivisible, discrete chunks of probability. The quanta can be thought of as *rods* of equal mass but (possibly) differing lengths. If a probability mass function has P rods in total, then the mass of each rod is $\frac{1}{P}$. The number of rods is called the *precision* of the approximation. Next, the domain of the mass function is sampled at C evenly-spaced *points*. C is called the *coarseness* of the approximation. The sample points are $\{\frac{1}{2C}, \frac{3}{2C}, \ldots, \frac{2C-1}{2C}\}$ (assuming that the domain of the mass function has been normalized to [0,1]). Typically, the coarseness is much larger than the precision. For instance, in our experiments, we use a coarseness of 2^{16} but a precision of 2^{8}. The approximation of the uniform mass function with a coarseness of 8 and a precision of 3 is shown in Figure 5.4.

A useful mental model is to think of the rods as covering the sample points. One condition that we impose on the approximation is that each point is covered by at most one rod, although a single rod can span several points. A pigeon-hole argument shows that a probability mass function's precision can never exceed its coarseness. That is, if there are 256 points, then there could be at most 256 rods (otherwise some point must covered by more than one rod).

The rod and point method of approximating a distribution has some limitations. The coarseness and precision restrict the variety of probability mass functions that can be approximately represented; some mass functions cannot be represented at all. If the coarseness equals the precision, then only the uniform probability mass function (every point is equally likely) can be represented (a different rod on every point). As the coarseness and precision diverge, more mass functions can be represented. In general, with a precision of P and a coarseness of C, at most $\binom{C}{P}$ different probability mass

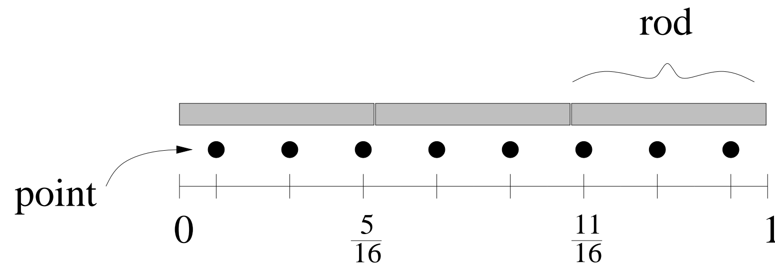


Figure 5.4: The approximated uniform mass function with $P = 3$ and $C = 8$

functions are possible. Also, mass functions that have a mass of more than $\frac{1}{P}$ spread over less than $\frac{1}{C}$ of their domain cannot be approximated. To model such "spiky" mass functions, two or more rods might have to span the same point. It is the database implementor's task to choose the appropriate *C* and *P* values, supporting the kinds of mass functions that are of interest to the users of the system.

Using the rod and point method, a probability mass function is approximated with an absolute error of less than $\frac{1}{P}$. That is, the probability of a possible instant in the approximated distribution is within $\frac{1}{P}$ of the actual probability. If the difference between the probabilities is more than $\frac{1}{P}$ then the approximation has been done incorrectly, as a new rod should have been introduced.

The approximated mass function is stored in a binary tree rather than an array. There is one leaf for each sample point. For example, the first leaf in a preorder traversal corresponds to the sample point $\frac{1}{C}$. At each leaf in the tree, the number of rods to the left and right of the sample point are stored. For example, in the approximation of the uniform mass function shown in Figure 5.4 there are no rods to the left and two rods to the right of the first point. The example shows that it is not always the case that the number of rods left and right of a sample point will sum to *P*; often, the number of rods will sum to $P - 1$ since the rod covering the node is uncounted. The tree for the approximated uniform mass function is shown in Figure 5.5.

In the tree shown in Figure 5.5, *C* and *P* are small values, consequently the entire tree can be easily stored in just a few bytes. But when *C* and *P* are large, it is infeasible to store the full tree, nor do we need to store the full tree. We are primarily interested in

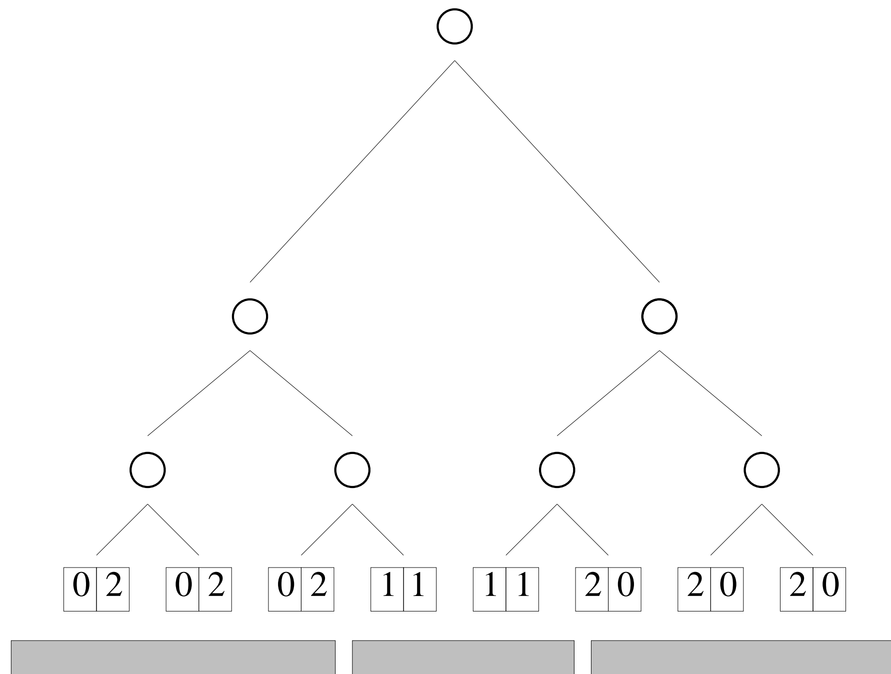


Figure 5.5: The tree storing the approximated uniform mass function with $P = 3$ and $C = 8$

the number of rods to the left and right of sample points. Observe that for many subtrees these numbers are the same for every node in the subtree. All such subtrees can be pruned, keeping only the root of the subtree, which is specially marked. When traversing a pruned subtree, the tree traversal algorithm treats a specially marked node as the root of a "virtual" subtree and traverses the subtree as though it were stored. The tree pruning technique saves quite a bit of space. The pruned tree has at most $2P$ leaves (one leaf might be needed per rod end) and could have as few as *P* leaves. In contrast, the unpruned tree has *C* leaves (in general $C >> P$). The number of interior nodes also varies, with as few as $P - 1$ interior nodes and as many as $2P - 1$ interior nodes in a tree (a binary tree with N leaves has $N - 1$ interior nodes). Each interior node is two $\mathbf{log}_2(C)$-bit pointers while each leaf node is two $\mathbf{log}_2(P)$-bit fields to store the number of rods. For $C = 2^{16}$ and $P = 2^8$, the storage cost of a pruned search tree is between 1.5K and 3K bytes. The pruned tree for the example distribution is shown in Figure 5.6. The specially marked

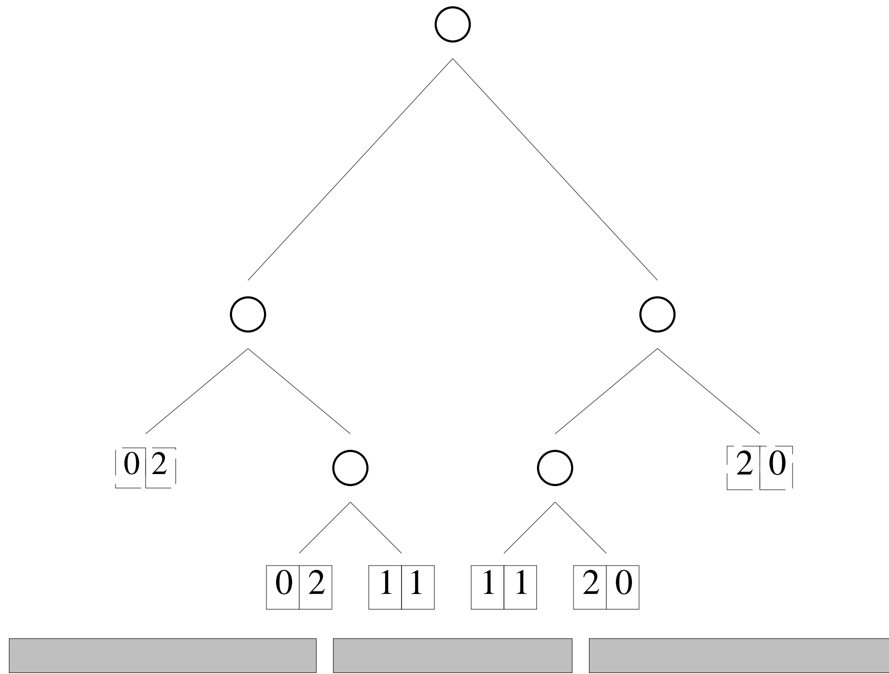


Figure 5.6: The pruned tree storing the approximated uniform mass function with $P = 3$ and $C = 8$

nodes have dashed line borders.

The distribution tree efficiently stores the approximated probability mass function, but the approximation impacts the computation of **Pr**$[\alpha < \beta]$, changing the problem to a rod counting problem.

5.2.1.3 An Overview of Computing Pr$[\alpha < \beta]$

Calculating the probability that one instant is before another using the approximated mass function can be reformulated as a rod counting problem. Assume that there are two rows of *P* rods. The rods could be of varying lengths, or they could all be the same length. The two rows of rods, which we shall call the α-row and the β-row, are parallel to each other as shown in Figure 5.7. The rod counting problem is to count the pairs of rods, one rod from each row, such that the rod from the α-row is before the rod from the β-row. Each such pair represents a contribution of $\frac{1}{P^2}$ to **Pr**$[\alpha < \beta]$.

The rod counting problem is complicated by the fact that the rods in each row might be further grouped into chronons. For the purpose of computing **Pr**$[\alpha < \beta]$, each chronon has an indivisible mass, that is, all the rods entirely within the same chronon should be treated as a single rod with a mass equivalent to the total mass of the constituent rods. For example, consider an indeterminate instant with a uniform mass function and a set of possible chronons consisting of only two chronons. There are $\lfloor \frac{P}{2} \rfloor$ rods within each chronon, consequently each chronon in this indeterminate instant has an indivisible mass of 0.5.

The rod counting problem differs from the original problem of computing the probability that one instant is before another instant in a subtle, but significant way: the sum of the mass in pairs of rods where α's rod is before β's rod is not quite the same as **Pr**$[\alpha < \beta]$. Consider a pair of rods, neither of which is before the other (the rods are at the same place in the overall ordering of rods). Each rod represents the probability that the instant is located during a certain range of chronons (the range could be just a single chronon); but how the probability is distributed among the chronons within that range is unknown. Although neither rod is before the other, it is probably the case that some chronon within the range represented by the rod is before a chronon in the range represented by the other rod. The rod counting problem does not count the (small, $\leq \frac{1}{P^2}$) probability of this case and thus undercounts **Pr**$[\alpha < \beta]$. In Section 5.2.1.6 we quantify the error on the rod counting technique.

The algorithm for counting pairs of rods is based on a divide-and-conquer technique. Each step in the algorithm is illustrated in Figure 5.7. The first step is to chose a *pivot*. A pivot is a rod in α's row or rods. The pivot splits the rods in α-row into three groups: those before the pivot, α_{before}, those after the pivot, α_{after}, and the pivot itself.

The second step is to identify where the right-end of the pivot belongs in the ordering of β's rods. The right-end of the pivot divides β's row of rods into three parts: those before the right-end of the pivot, β_{before}, those after the right-end, β_{after}, and, perhaps, a rod that overlaps the right-end, $\beta_{overlap}$.

The third step is the conquer step. Observe that all the rods in $\alpha_{before} \bigcup pivot$ are before all the rods in β_{after}. Each pair of rods, one chosen from each of these two groups,

adds $\frac{1}{P^2}$ to a running sum of **Pr**$[\alpha < \beta]$. If the running sum (scaled by 100) exceeds the plausibility, γ, then the algorithm terminates since the plausibility has been met and *Before* is true. This is called an "early exit" condition.

Similarly, all the rods in β_{before} are before the rods in α_{after}. Each pair of rods, one chosen from each of these two groups, adds $\frac{1}{P^2}$ to a running sum of **Pr**$[\beta \leq \alpha]$. If the running sum (scaled by 100) exceeds $100 - \gamma$, then the algorithm terminates since *Before* is false. This is the only other "early exit" condition.

If an early exit is not taken, then two subproblems remain to be solved. The algorithm has yet to determine the relationships between the rods in $\alpha_{before} \bigcup pivot$ and those in β_{before}, as well as the relationships between the the rods in α_{after} and those in $\beta_{after} \bigcup \beta_{overlap}$. Each of these subproblems is solved recursively in the next "round" of the algorithm.

5.2.1.4 Choosing the *Pivot*

The choice of pivot is an important factor in controlling the algorithm. The algorithm chooses as the pivot the rod corresponding to half of the remaining rods in α (those rods that have yet to be counted). This choice enables the algorithm to reach an "early" exit condition quickly. Overall, the total work performed by the algorithm is to count all P^2 pairs of rods. But the counting can stop when enough pairs are counted to determine if either **Pr**$[\alpha < \beta]$ or **Pr**$[\beta \leq \alpha]$ is satisfied (the early exit conditions). It is better if, in the first few pivot choices, an algorithm maximizes the pairs of rods it counts since it will then hit an exit condition in fewer pivots.

Theorem 2 *The kth pivot will count $P^2/2^{\lfloor \log_2(k) \rfloor + 1}$ pairs.*

Proof. By choosing the rod corresponding to half of the remaining rods, the algorithm counts half the pairs on the first pivot, that is, it counts $P^2/2$ pairs. On the second and third pivots, it counts half of half of the remaining pairs, or $P^2/8$ pairs per pivot, assuming "breadth-first" recursion. On the fourth through seventh pivots, it counts half of half of half of the remaining pairs, or $P^2/32$ pairs. So in general, the kth pivot will count $P^2/2^{\lfloor \log_2(k) \rfloor + 1}$ pairs. In the worst case, $2P$ pivots are required. ∎

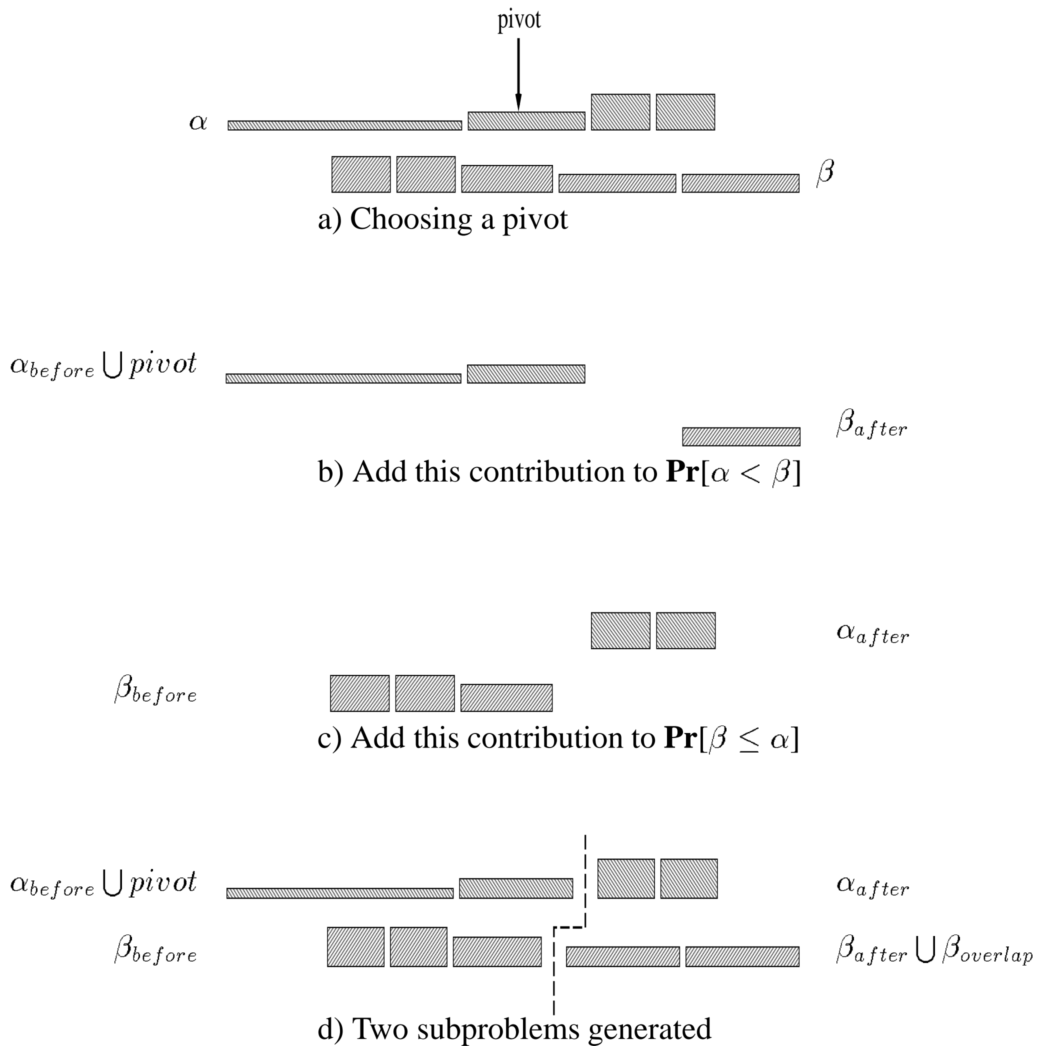


Figure 5.7: A rod counting operation

Observe that, for a precision of 2^8, after the fourteenth pivot, the algorithm will have counted 93% of the total number of pairs. In other words, to approximate the ordering probability to within 10%, at most fourteen pivots must be performed.

5.2.1.5 Implementation Details for Computing Pr$[\alpha < \beta]$

The code for the "pivoting" algorithm is shown in Figure 5.8. The counting stops when the count of pairs exceeds the needed number of true pairs or false pairs (it simultaneously solves for both **Pr**$[\alpha < \beta]$ and **Pr**$[\beta \leq \alpha]$), or when all the possible pivots have been tried (an undercount has occurred and we assume that **Pr**$[\alpha < \beta]$ is false). The most important

feature of this code is that the majority of instructions are "cheap" integer operations: shifts, assignments, and additions. There are only two multiplications, no divisions, and no floating point operations. Although, for pedagogical reasons, we have presented the pivoting code as a recursive procedure, the procedure is implemented using a queue and iteration, thus avoiding the expense of recursive procedure calls and supporting breadth-first recursion. One final observation, calculating the number of true and false pairs has been reduced to a table-lookup since the ordering plausibility, γ, can take on only 100 different values.

5.2.1.6 Error in Rod Counting

As we pointed out earlier, the approximation by rods and points leads to an undercounting of **Pr**$[\alpha < \beta]$. In this section we quantify the magnitude of the undercounting.

Theorem 3 *The undercount is less than* $\frac{2}{P}$.

Proof. First consider the error once a pivot has been chosen. The error is the rods in the other row of rods that remain uncounted. The uncounted rods are those that overlap the pivot. These rods are uncounted because it is unknown how the probability mass is distributed within each rod, consequently it is impossible to determine whether the mass is before or after the mass in the pivot. Figure 5.9 shows the rods that are uncounted for an example pivot; the rods that are either partially or wholly within the dotted lines are not counted. But how many pairs of rods possibly overlap? We claim that there can be at most $2P - 1$ pairs that overlap.

We demonstrate this claim by modeling the overlapping rods with an undirected graph. Let each rod be a node in a graph. Add an edge between each pair of rods that overlaps. Observe that the edges cannot "cross" each other, that is, the graph is *planar*. Now count the total number of edges in the graph. Choose the first, or "leftmost" edge in the graph. Since edges cannot cross, at least one node on this edge is a sink, unconnected to any other nodes by a different edge. Eliminate both the node and the edge. Repeat this process, always choosing the "leftmost" remaining edge, until there are no more edges. Initially there are $2P$ nodes. One node is eliminated at every step along with one edge. At least

one node remains after the final edge is removed. Consequently, initially, there were at most $2P - 1$ edges.

Each edge represents a pair of rods that overlap, corresponding to a mass of $\frac{1}{P^2}$ that remains uncounted. Since there are at most $2P - 1$ edges, the total missing mass is less than $\frac{2}{P}$. For a precision of 2^8, the total error is less than 1%. ∎

The rod counting algorithm allows *Before* to be efficiently implemented (Section 5.2 will examine its efficiency in detail). We now turn to the implementation of the other operations that change with support for indeterminacy: *Shrink_s*, *Shrink_t*, *Set_Before*, and *Reduce′*.

5.2.2 The *Shrink* functions

A *Shrink* function changes an indeterminate instant in two ways. First, it shortens the instant's period of indeterminacy, and second, it modifies the instant's mass function. In this section, we describe the implementation of these two changes. Surprisingly, the first change is more challenging than the second to implement.

A period of indeterminacy is shortened by computing a new upper or lower support. In a *Shrink_s* operation, the new support is the chronon where the cumulative probability exceeds the user-specified credibility. The implementation of the *Shrink_s* operation treats the credibility as a percentage of rods that are to be removed from the row of rods. The leftmost chronon in the earliest remaining rod becomes the new lower support, thereby removing the "early" portion of the period of indeterminacy. For example, in a *Shrink_s* operation with a credibility of 50, the chronon corresponding to half of the remaining probability becomes the new lower support. The location of that chronon depends on the mass function. For a uniform mass function, the chronon is in the middle of the period of indeterminacy, whereas for a probably early mass function it is towards the early portion of the period of indeterminacy. Wherever it is in a period of indeterminacy, the chronon corresponding to half of the remaining probability is in the "middle" rod in the row of rods. We use a binary search technique to find the leftmost chronon in that rod.

To compute a new mass function, we record in the left or right offset of the indeterminate timestamp how much of the mass function has been removed by the *Shrink*

```
function PROB_α_BEFORE_β(in α, β : event; in γ : integer) : boolean;
    const
        P : integer = 256; C : integer = 65536;
        α_tree, β_tree : probability_mass_function_tree(C,P);
        plausibility_map : array[1..100] of 1..⌊P²/100⌋;
    var
        false_pairs, true_pairs, pivot, α_mid : integer;
        leaf : tree_node_pointer;
    procedure ROD_COUNTING(in α_from, α_to, β_from, β_to : integer);
        begin
        { Check the exit conditions }
            if (true_pairs ≤ 0) ∨ (false_pairs < 0) then return;
            if ((α_to − α_from) = 0) ∨ ((β_to − β_from) = 0) then return;
        { Calculate pivot }
            pivot ← α_from + ((α_to − α_from) div 2);
        { Figure out α's contribution }
            α_before ← pivot − α_from;
            α_after ← α_to − pivot;
        { Figure out the chronon in which the pivot ends using binary search }
            α_mid ← binary_search(α_tree, pivot);
        { Find the rod in β just after the pivot's chronon using binary search }
            leaf ← binary_search(β_tree, α_mid);
        { Figure out β's contribution }
            β_before ← leaf.left_rods − β_from;
            β_after ← β_to − leaf.right_rods;
        { How much is the total contribution? }
            true_pairs ← true_pairs − ((α_before + 1) × β_after);
            false_pairs ← false_pairs − (α_after × β_before);
        { Continue counting }
            ROD_COUNTING(α_from, α_from + α_before, β_from, β_from + β_before);
            ROD_COUNTING(α_to − α_after, α_to, β_to − β_after, β_to);
        end; { ROD_COUNTING }
    begin
        true_pairs ← plausibility_map[γ];
        false_pairs ← P² − true_pairs;
        ROD_COUNTING(1, P, 1, P);
        return true_pairs ≤ 0
    end; { PROB_α_BEFORE_β }
```

Figure 5.8: The pivoting algorithm

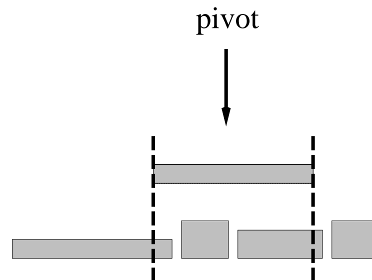


Figure 5.9: The rods within the dotted lines are the undercount for the pivot

functions. Specifically, the left and right offsets keep track of which rods remain in the approximated mass function after a shrink operation. For example, assume that the user executes a *Shrink_s* with a range credibility of 50 on an instant with *P* rods. This operation will remove half of the rods, or $\frac{P}{2}$ rods from the left side of the probability mass function associated with the instant. We mark these rods as removed by updating the left offset to $\frac{P}{2}$.

When a rod is removed, the remaining rods become more massive to compensate for the lost mass. If ten rods are removed, the mass represented by each remaining rod is effectively $\frac{1}{P-10}$ rather than $\frac{1}{P}$. Hence, all that is needed to compute a new mass function is to keep track of how many rods have been removed.

But how does the shrunken mass function affect the pivoting algorithm used in *Before*? Recall that *Before* counts pairs of rods that precede each other in two rows of rods, until the count satisfies a certain ordering plausibility. Assume that the mass of each rod in one row is $\frac{1}{x}$ (initially there were *P* rods, but $P - x$ rods have been removed as recorded in the left and right offsets) and in the other row $\frac{1}{y}$. Further assume that by counting n pairs of rods, the ordering plausibility, γ, is reached. Mathematically, we can express this as $\frac{\gamma}{100} \leq \frac{n}{xy}$. Since all the variables in the equation are non-negative, we can rewrite the relationship as $\frac{\gamma xy}{100} \leq n$. The rewritten relationship shows that counting with rods of increased mass is equivalent to first scaling γ by an appropriate factor, by $\frac{xy}{100}$, and then simply counting the rods. Hence we do not have to change the pivoting algorithm or data structures. It is straightforward to add the code to scale γ to the header code for *Before*. The code for

```
function Shrink_s(in α : event; in γ : integer) : event;
    const
        P : integer = 256; C : integer = 65536;
        α_tree : probability_mass_function_tree(C, P);
    var
        j : integer;
    begin
    { The function returns β }
        β ← α;

    { Determine the new left offset }
        β.left_offset ← α.left_offset
                    + (((α.right_offset − α.left_offset) × γ) div 100);

    { Find the chronon of the rod for the new left offset }
        β_* ← binary_search(α_tree, (α.left_offset × P) div 100);
        return β
    end; { Shrink_s }
```

Figure 5.10: The *Shrink_s* algorithm

Before must also be sensitive to which rods have been removed from a probability mass function. Only the initial call to *ROD_COUNTING* in the body of *PROB_α_BEFORE_β* needs to be changed; instead of asserting that both mass functions start at rod 1 and continue to rod *P*, the initial call to *ROD_COUNTING* must pass the correct beginning and ending rods (computed from the left and right offests for both instants).

It is important to note that since the shrink functions change the mass that is approximated by a rod, the error on the approximation and the error in the *Before* algorithm both increase due to cumulative shrinking.

The straightforward pseudo-code for *Shrink_s* is given in Figure 5.10. Similar code for *Shrink_t* is omitted for brevity.

5.2.3 *Set_Before*

Set_Before also appears costly since it must compute over sets of instants. But we stipulate that there can be at most 32 instants in a set (the maximum number of instants in an expression, corresponding to between eight and sixteen tuple variables, a very high

maximum in practice); hence, a single word of storage suffices to represent a set, each bit indicating membership of an instant in the set. *Set_Before* efficiently determines if all the instants in one set are *Before* those in another by performing logical *and*s between rows in the boolean table of previously computed *Before* results.

5.2.4 *Reduce′*

The implementation of *Reduce′* is little changed from that of the original *Reduce*. The indeterminate instants with the earliest and latest extent must be computed, but this adds only two timestamp comparisons to each step. Of greater consequence is that *Reduce′* must deal with more tuples than *Reduce* because of the nondeterminism in *First* and *Last*. The valid clause specifies the valid times of tuples that are input to *Reduce′*. The valid clause is an expression consisting solely of temporal constructors such as *First* and *Last*. For plausibility values below 50, *First* and *Last* are nondeterministic and might produce both argument instants (if the instants are indistinguishable), each of which could be the valid-time timestamp of a tuple. For plausibility values above 50, at most one instant can be produced (as is the case for the determinate semantics). Consequently, in the indeterminate semantics (for plausibilities below 50) more tuples may result than with the determinate semantics. For an expression involving the *First* and *Last* constructors, at most $2^{|First|+|Last|}$ different instants could be produced, but no more than the total number of instants in the expression (a maximum of 32). In the tuple calculus semantics shown in Section 4.4.4 for the query in Figure 4.4, only one tuple will be input to *Reduce′* since the plausibility on the valid clause is above 50. If the plausibility were lowered to 25, then at most four tuples could be input to *Reduce′* since there is one *First* and one *Last* constructor in the valid clause.

5.2.5 Impact of Indeterminacy on the Determinate Implementation

In parallel with the theorem of query reducibility given in Section 4.4.5, conventional TQuel queries on determinate relations will incur no additional execution overhead under

the new semantics, and executing such queries on indeterminate relations will only add *Shrink_s*$(100, ,)$ and *Shrink_t*$(100, ,)$ invocations, which effectively discard the indeterminate information.

5.3 Empirical Analysis of the Implementation

We implemented the indeterminate operations in the C programming language using the GNU C compiler, version 2.0, with compiler optimization fully enabled. We used a precision of 2^8 and a coarseness of 2^{16} in the code for the indeterminate *Before*. These values limit the maximum error in the pivoting algorithm to less than 1%. We also implemented *Before* with a maximum possible error of 10%. This version of *Before* performs (at most) 14 pivots as discussed in Section 5.2.1.6. Each operation was coded as a separate C function.

We then tested the performance of each operation. All tests were performed in a controlled environment on a dedicated SPARC station $1+$ (a 12-MIP machine). A single test consisted of ten separate runs, where each run executed the operation between one thousand and one million times to avoid any internal clock sampling errors. Table 5.2 shows the results for each operation. The execution times shown in the figure are the average over all runs and include the cost of the function call. 'NA' denotes "not applicable." The cost of *Reduce′* is given in terms of the cost of *Before*. For most of the operations, the best and worst cases differ very slightly, but the behavior of *Before* is complex and depends on the instants being compared.

To further examine *Before*'s behavior, we devised several additional tests for the indeterminate *Before*. These tests were designed to capture both the worst case and the expected case performance of the pivoting algorithm. The worst case for *Before* happens when the two indeterminate instants span the same chronons and have nearly uniform distributions. We tested this worst case performance of *Before* on a pair of events, each of which has a period of indeterminacy of one million chronons. The results are given in Figure 5.11. The graph plots the execution time (in microseconds) of *Before* for the plausibility values 1 to 100. As the ordering plausibility approaches 50, the execution times increase because more pivots are needed to determine the outcome of *Before*. The average

Operation	Determinate Cost (in microseconds)	Indeterminate Best Case Cost (in microseconds)	Indeterminate Worst Case Cost (in microseconds)
Determinate *Before*	0.4	NA	NA
Indeter. *Before* - 1% error	NA	0.6	5000
Indeter. *Before* - 10% error	NA	0.6	320
Shrink_s	NA	0.4	2
Shrink_t	NA	0.4	2
Set_Before	NA	0.4	0.4
Reduce' (per pair of tuples)	1-2 *Before*'s	1 *Before*	2 *Before*'s

Table 5.2: Timings on indeterminate operations

worst case *Before* operation with 1% error, across all plausibilities, is approximately 157 microseconds, with a high of 5000 microseconds at a plausibility of 50, and a low of 0.6 microseconds. With a maximum error of 10%, the average worst case for *Before* is somewhat less, 72 microseconds, with a high of 320 microseconds at a plausibility of 50.

The worst case performance does not always occur at a plausibility of 50, but depends on the relative positions and mass functions of a pair of indeterminate instants. Two instants with nearly uniform distributions which have partially (but not fully) overlapping periods of indeterminacy will exhibit worst case performance at plausibilities other than 50. Using the same two instants from the first test, we tested a range of relationships between their periods of indeterminacy, from no overlap to complete overlap. We fixed the position of one instant in chronon space and slid the other instant relative to the fixed instant. Figure 5.12 shows the results. The z-axis is the cost (in microseconds) of a single call to *Before* (we used the 10% error version). The x-axis is the plausibility. The y-axis is the relative position of the two instants, “far apart” indicates no overlap in the periods of indeterminacy whereas “even” means complete overlap. The figure shows that if the instants do not overlap (a common case), *Before* is very cheap. If the instants overlap, *Before* only exhibits poor performance along a central ridge. Note that the graph

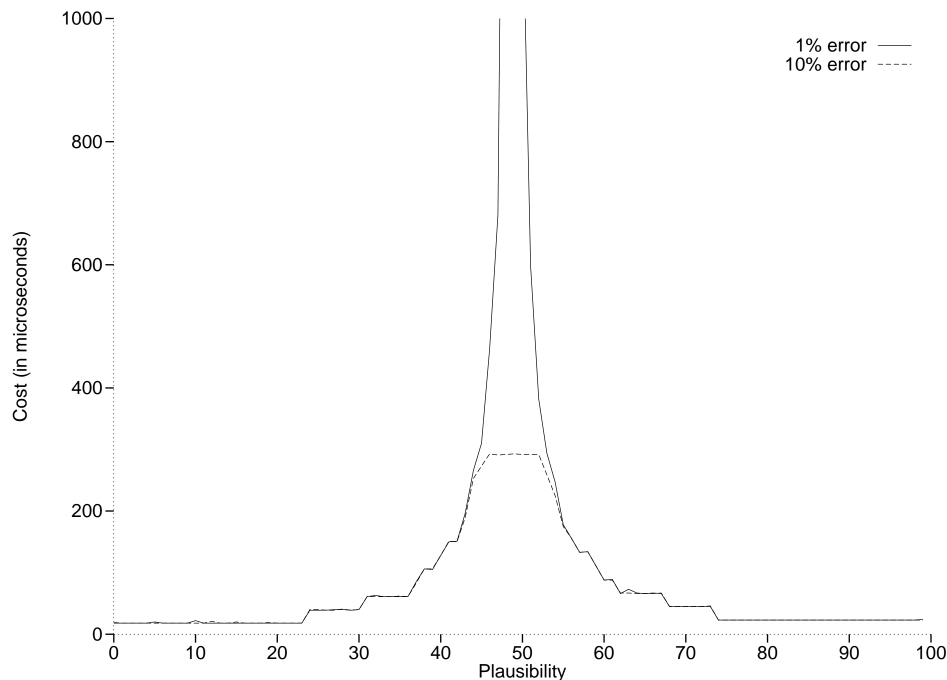


Figure 5.11: Worst-case performance of *Before*

in Figure 5.11 is a slice of the graph in Figure 5.12 at the "even" point along the "relative position" axis.

While examining worst-case behavior is sometimes illuminating, we anticipate that it will be uncommon for two instants to have overlapping periods of indeterminacy, and that worst-case behavior will be rarer still. For example, consider an event relation of employee hires timestamped with the day of hiring. The day an employee was hired is an indeterminate instant, assuming a common timestamp granularity of a second. Suppose we query this relation to determine which employees were hired before the third fiscal quarter began. The quarter began at 8 AM on October 1st. It is unlikely that most of the hiring instants overlap 8 AM on October 1st. Hence, the precedes comparison for most of the instants in the relation will be very efficient, and the impact of the other comparisons on the total work done in the query will be slight.

To explore this aspect further, we devised a further test. We randomly placed ten instants, each of which had a one-day period of indeterminacy (86,400 chronons) and a

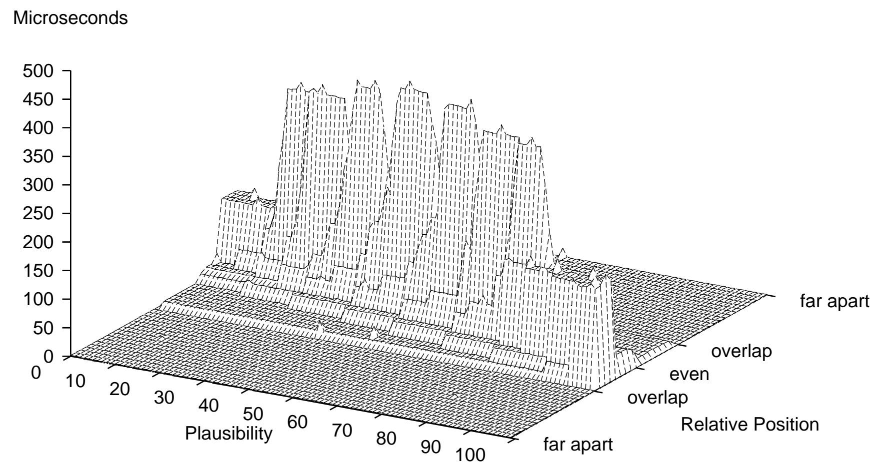


Figure 5.12: Sliding one instant relative to another performance of *Before*

nearly uniform distribution, in a chronon space that varied between 1 and 50 days (between 86,400 and 4,320,000 chronons) in size. For every plausibility between 1 and 100, we tested *Before* on every possible combination of instants (10^2 possible combinations) at plausibilities ranging from 1 to 100. Instants were not compared to themselves. We performed one hundred such tests for each plausibility and point in chronon space. We rerandomized the location of the instants between each test (i.e., per one hundred comparisons). The results are depicted in Figure 5.13. The graph in Figure 5.11 is a slice of this graph at a value of one unit of random space. The graph shows that in a normal mix of instants, rare worst-case situations have little impact on overall performance. Only in when the size of the random space is small is the cost of *Before* significant.

To this point we have not determined how much more expensive we expect the indeterminate *Before* to be than the determinate *Before*. To measure the relative cost of the indeterminate *Before* we reran the “random placement of instants” test described above on both the indeterminate and determinate *Before*. But this time we let the size of the random chronon space vary between 1 and 1800 days rather than 1 and 50. For each day,

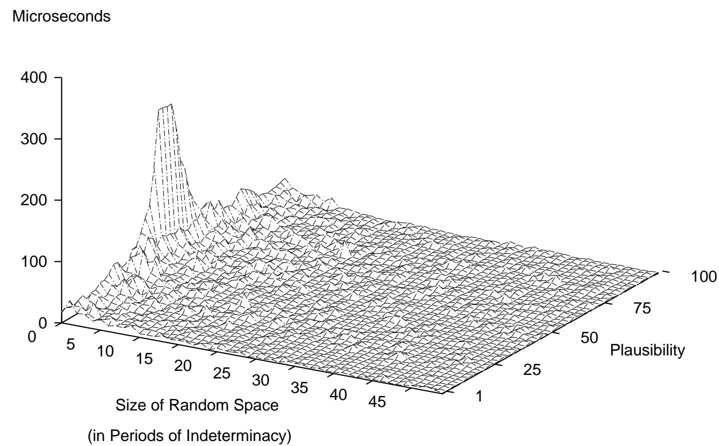


Figure 5.13: The cost of comparing ten instants randomly placed in a chronon space of varying size

we averaged the cost of the *Before* operation across all the plausibilities, 1 to 100. The results are plotted in Figure 5.14. When all ten instants are randomly placed in a chronon space three months in size (i.e., there are ten employee hires in three months and only these hires are used in the query), the indeterminate *Before* is approximately six times more expensive than the determinate *Before*. By one year it is only twice as expensive, and thereafter, it is approximately half again as expensive as the determinate operation. The asymptotic time for the indeterminate *Before* is $.6$ microseconds (1% meets 10% in the limit) and $.35$ for the determinate *Before*.

Although the run-time cost of each operation considered in isolation is informative, it does not address the "actual" cost of a query with indeterminate information, since the frequency of operations and the interactions between operations are absent from the analysis. In addition, these operations are only one portion of query evaluation; many other operations are included. To measure *Before* and the other operations in context, we designed a test of a complete query. We hand-compiled the example TQuel query given in Figure 4.4 on page 62, into a series of calls on a prototype system for temporal

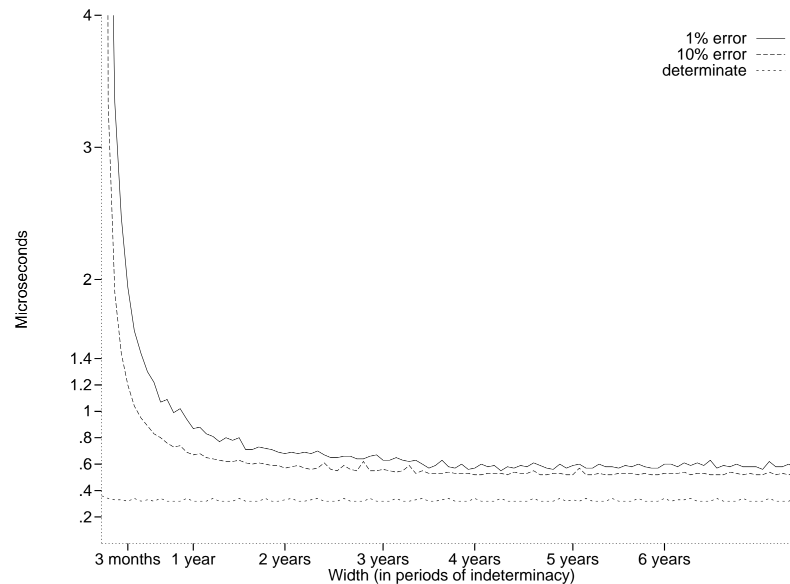


Figure 5.14: The average cost of comparing ten instants randomly placed in a chronon space of varying size

support called MULTICAL [Soo et al. 1992]. We chose MULTICAL primarily because it is the only such system in the public domain (of which we are aware). The sequence of MULTICAL calls are shown in Figure 5.16. The determinate and indeterminate sequences of calls differ only slightly; the differences are highlighted in italics in the indeterminate sequence. For both sequences we suppressed all input, output, and disk I/O as these expensive operations tend to dominate the cost of other operations. We also used the chunked indeterminate timestamp formats, which are more expensive to unpack, but have the same space cost as the determinate timestamps used in the experiment (consequently disk I/O costs should be the same).

To test the query we once again used a variation of "the random placement of instants." We used the tuples shown in Figure 1.1, but randomly placed the indeterminate (determinate) instants in an chronon space of increasing size. We used a period of indeterminacy of twenty-four hours for the instants in both relations. The results are shown in Figure 5.15. Except for the rare situations where most of the instants are packed into a relatively tight

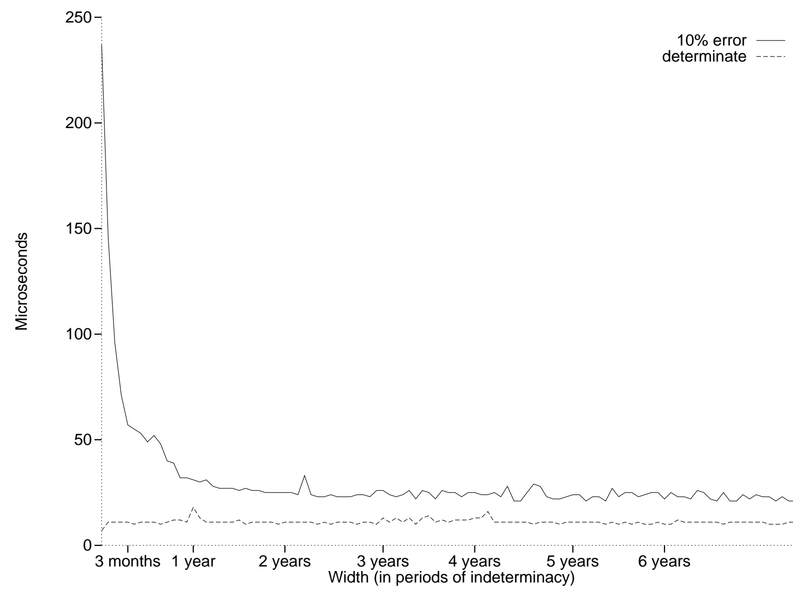


Figure 5.15: The average cost of the example query per combination of tuples

random space, indeterminacy roughly doubles the cost of the query, from an asymptotic time of 17 microseconds to that of 31 microseconds with indeterminacy.

{ The determinate sequence }	{ The indeterminate sequence }
for every combination of r and p	**for** every combination of r and p
	Shrink_t(0, p.to);
	Shrink_s(0, p.from);
unpack_instant(r);	unpack_instant(r);
unpack_interval(p);	unpack_interval(p);
c1 ← Before(r, p.to);	*c1 ← Before(60, r, p.to);*
c2 ← Before(p.from, r);	*c2 ← Before(60, p.from, r);*
if (c1 and c2) **then**	**if** (c1 and c2) **then**
e1 ← First(r, p.to);	*e1 ← First(60, r, p.to);*
e2 ← Last(p.from, r);	*e2 ← Last(60, p.from, r);*
temp ← Before(e1, e2);	*temp ← Before(60, e1, e2);*
add temp to result	add temp to result
end	**end**
end	**end**

Figure 5.16: MULTICAL calls for example query

CHAPTER 6

INDETERMINACY IN TSQL2

This chapter is a proposal for adding temporal indeterminacy to the TSQL2 language, data model, and algebra. We outline syntactic and semantic extensions to TSQL2 to support retrieval of temporally indeterminate information. These extensions were adopted by the TSQL2 committee and are now part of the proposed TSQL2 language [Snodgrass et al. 1994]. Temporal indeterminacy is orthogonal to SQL-92's support for null values.

The TSQL2 language has slightly different terminology for the temporal concepts of instant, span, and interval; in TSQL2, they are known as datetime, interval, and period, respectively. In order to present a coherent thesis we will continue to use instant, span, and interval in this chapter to refer to these familiar temporal entities. However, we will use TSQL2 syntax in the language examples and in the appendices (Appendix A gives the BNF for the language extensions to support indeterminacy). Also, in the appendices, we will use TSQL2 terminology for temporal concepts.

6.1 Syntactic Extensions to Support Temporal Indeterminacy

This section describes the syntax for retrieving information from databases with temporal indeterminacy; the next section outlines an informal semantics for these constructs. The proposed syntax and semantics are extensions of the syntax and semantics of the TSQL2 select and create table statements. A primary design goal in extending TSQL2 to support indeterminacy was to make a minimal extension. The new syntax and semantics preserves the meaning of all extant non-indeterminate TSQL2 select statements, just as it did for extant TQuel statements.

We make four syntactic extensions to TSQL2, one to indicate that a table is indeterminate, one to specify the range credibility, one to specify the ordering plausibility, and one to specify defaults.

Our first syntactic extension involves the schema specification statements. To the create table statement we add the option of specifying that a timestamp may be indeterminate. Before the reserved words DATE (specifying an instant), TIME (specifying an instant), DATETIME (specifying an instant), TIMESTAMP (specifying an instant), INTERVAL (specifying a span), or PERIOD (specifying an interval) a user may add either the modifier INDETERMINATE or GENERAL INDETERMINATE. For example,

```
CREATE TABLE Emp( Bdate GENERAL INDETERMINATE DATE );
```

These options toggle between alternative storage strategies for indeterminate timestamps, discussed at length in Chapter 5. The "general" version is a less compact timestamp. We also add an optional reserved word that allows the user to specify either the *standard* (the uniform or distribution that is missing) or *nonstandard* mass functions; for example

```
CREATE TABLE Emp( Bdate NONSTANDARD GENERAL
                        INDETERMINATE DATE );
```

The default is standard; the optional reserved word NONSTANDARD chooses the slightly larger, nonstandard distribution formats.

We also extend the CREATE statement to allow dynamic creation or modification of distributions. A user specifies creation of a distribution rather than a table by using the reserved word DISTRIBUTION. For example, to create a local distribution named probably_early, a user might code the following.

```
CREATE LOCAL DISTRIBUTION probably_early
                        USING probably_early_table;
```

The new distribution is described by a two-column table named in the USING clause. The first column of this table is of type integer and is the domain of the distribution. The second column is of type float and is the probability associated with each point

in the domain. The probabilities in the second column must sum to one. If elements of the domain are missing, they are assumed to have probability 0. The distribution table is processed by an implementation-dependent tool that generates the distribution data structures. Some distributions cannot be supported in the implementation; the tool will detect unsupportable distributions and report errors. Distributions created in this way can be subsequently altered (using `ALTER DISTRIBUTION`) and dropped (using `DROP DISTRIBUTION`).

Range credibility appears in the from clause of a select statement. The range credibility is the credibility in each interval column. The credibility applies independently to the starting and terminating instants in the interval. It is an integer value between 0 and 100 (inclusive). The credibility phrase is optional and has an initial default value of 100. This default value can be changed using a set statement as follows.

```
SET CREDIBILITY 50
```

The set statement is very useful when a group of queries is to be made at a particular credibility level, or when the credibility is to be specified for a novice. Range credibility is not applicable to instant tables because removing indeterminacy from an indeterminate instant might require partitioning the instant.

Ordering plausibility is the plausibility in the temporal ordering of the instants that participate in the select. The ordering plausibility is an integer between 1 and 100 (inclusive) and may be specified for the entire where or valid clause. The plausibility phrases are optional and have an initial default value of 100, which can be changed using a set statement.

An example query with optional credibility and plausibility phrases is shown in Figure 6.1. Intuitively, the query will determine, within the specified plausibility and credibility levels, which `wing strut` shipments were received during production of each `Centurion`. The select statement in the figure has a plausibility value of 60 for the where clause.

```
SELECT Warehouse, Lot#, Part#
   VALID R
   FROM Received R, In_Production IP WITH CREDIBILITY 0
   WHERE Model = 'Centurion' AND
      Part = 'wing strut' AND
      R OVERLAPS IP WITH PLAUSIBILITY 65
```

Figure 6.1: An example query

6.2 Semantic Extensions to TSQL2

The semantic extensions to support TSQL2 are essentially the same as those for TQuel given in Chapter 4. The semantic extensions to support temporal indeterminacy involve the redefinition of several existing functions and relations and the introduction of new functions. Specifically, we redefine the temporal ordering relation to support ordering plausibility, we introduce two "shrink" functions to effect range credibility, we redefine the coalescing operator, *Reduce*, and we define the semantics of arithmetic and aggregate operations on indeterminate values. Only the semantics of arithmetic on indeterminate values is new in TSQL2; we discuss their semantics below. Support for temporal indeterminacy is an extension of the determinate semantics rather than a replacement. Hence, the semantics of existing queries is left unchanged.

6.2.1 Semantics of Arithmetic Operations

Arithmetic using indeterminate temporal values is similar to the arithmetic of interval mathematics [Kulisch & Miranker 1981], introduced to analyze the error bounds on floating-point operations (among other uses). Below, we reproduce the interval mathematics formulæ for the addition of two intervals and the multiplication of an interval and a scalar.

Operand	Operand	Mass function of Result
determinate	determinate	not applicable
determinate	indeterminate	mass function of indeterminate operand
determinate	scalar	not applicable
scalar	indeterminate	mass function of indeterminate operand
scalar	scalar	not applicable
indeterminate	indeterminate	*missing*

Table 6.1: The mass function that results from an arithmetic operation

$[a, b] + [c, d] = [a + c, b + d]$ assuming $a \leq b \wedge c \leq d$
$[a, b] \times c = [a \times c, b \times c]$ assuming $a \leq b$

In the arithmetic of indeterminate values, the period of indeterminacy is treated as an interval, and the interval of the result corresponds to the period of indeterminacy of the result. For instance,

$$\texttt{DATE'June 1} \sim \texttt{June 2'} + \texttt{INTERVAL'2 days} \sim \texttt{3 days'}$$
$$= \texttt{DATE'June 3} \sim \texttt{June 5'}$$
$$\texttt{INTERVAL'2 days} \sim \texttt{3 days'} \times 2 = \texttt{INTERVAL'4 days} \sim \texttt{6 days'}$$

If the result of an arithmetic operation is an indeterminate value, both a new period of indeterminacy and a new probability mass function must be computed. Which mass function is computed depends on the kind of arithmetic operation. In the addition of two indeterminate values, the mass function of the result is the convolution of the mass functions belonging to the operands. Typically, the convolution is not among the pretabulated mass functions; consequently, the result of the operation is a distribution that is *missing*. Table 6.1 lists the possible operands in arithmetic operations and the resulting mass function (assuming that the result is not a scalar, as is the case with some span/span operations). The table shows that arithmetic between two indeterminate values always results in a distribution is missing.

6.2.2 Semantics of Aggregate Operations

For aggregates, temporal indeterminacy impacts both the grouping of tuples and the semantics of individual aggregate operations [Kline et al. 1994].

6.2.2.1 Aggregation via Selection

Selection based aggregates over instants, intervals, spans, or temporal elements are all ultimately based on the *Before* operation. To support temporal indeterminacy, the semantics of aggregates on temporally indeterminate values remains essentially intact, only the determinate *Before* is replaced with the indeterminate *Before*. For example, consider the aggregate `MIN(T)` where `T` is a column of indeterminate instants. This aggregate computes the tuple that has the earliest time for that column. In the determinate semantics, *Before* is used to determine which instant in a column of instants is the earliest. For a column of indeterminate instants, the indeterminate *Before* is used with the plausibility given by the user for that query.

6.2.2.2 Aggregation via Computation

Computation based aggregates over instants, intervals, spans, or temporal elements are all ultimately based on arithmetic operations. To support temporal indeterminacy, the semantics of computation based aggregates on temporally indeterminate values remains essentially intact, only the determinate arithmetic operations are replaced with the indeterminate operations. For example, consider the aggregate `SUM(S)` where `S` is a column of indeterminate spans.. `SUM` computes the cumulative sum of a column of spans. For a column of indeterminate spans, the indeterminate addition operation is used. Note that the plausibility given by a plausibility phrase for the aggregate is inert in aggregation via computation, as it is for all arithmetic operations.

6.2.2.3 Weighted Aggregates

Weighted selection or computation based aggregates (e.g., computing the average salary over time, where the average salary at an instant is computed from timestamps that span

several months) over temporally indeterminate values are not supported and will result in a compile-time error. For weighted aggregates, values are weighted by their length in time. If that length is indeterminate, the resulting weighted value will also be indeterminate (e.g., computing the average salary over time would result in an indeterminate salary). Since indeterminate (nontemporal) values are currently unsupported, weighted aggregates can be defined only over temporally determinate values.

6.2.3 Input and Output of Indeterminate Temporal Constants

We suggest a simple form for indeterminate temporal constants that minimizes the impact of indeterminacy on input and output. Since calendars can already parse determinate constants, the suggested form of an indeterminate constant consists of two determinate constants separated by the character $\sim$[1]. Indeterminate spans have a similar standard form, that is, two determinate spans separated by a $\sim$, e.g., `INTERVAL'2 days ~ 3 days'`.

A probability mass function can be named inside a constant by using an optional phrase "with distribution X", where X is the name of some known distribution, e.g., `INTERVAL'2 days ~ 3 days with distribution uniform'`.

6.3 Summary

This chapter describes an extension of TSQL2 to support temporal indeterminacy. This support mimics TQuel's support insofar as it provides the user with two controls on querying a database, range credibility and ordering plausibility. We have augmented the create statement to specify which tables incorporate indeterminacy, extended the from clause with an optional with clause to specify range credibility, extended the select statement to specify ordering plausibilities, and added variants to the set statement to specify default plausibilities and credibilities. We also considered the impact of indeterminacy on arithmetic and aggregate operations. The full list of modifications to the syntax appears in the appendix.

[1] The particular character(s) are specified by the *indeterminate_datetime_format* and the *indeterminate_interval_format*.

CHAPTER 7

INDETERMINACY AND GRANULARITY

There is one feature common to all temporal data: *granularity*. Granularity is the unit of measure for a temporal datum. For instance, birthdates are typically measured in or known to the granularity of days, business appointments to the granularity of hours, and train schedules to that of minutes. The mixing of temporal data at different granularities in a single database is common, but creates various problems.

- What are the semantics of operations with operands at differing granularities? For example, what is the meaning of comparing a time known to the granularity of a day with a time known to the granularity of an hour? Can the two times ever be ordered?
- Can times be converted from one granularity to another, for instance, from days to years? How about from years to days?
- How expensive is maintaining and querying times at different granularities? Can times at differing granularities be stored as efficiently as times at a single granularity? How much more expensive are temporal operations at mixed granularities?

It is surprising that the issue of mixed granularities, of basic importance to modeling "real-world" temporal data, has, to date, lacked a comprehensive solution. In this chapter, we address each of the problems just posed. The practical solution that we outline is based on a realistic model of time, can be easily integrated with SQL-92 syntax, has a clear semantics for temporal operations on operands at differing granularities, and most importantly, has an efficient implementation.

This chapter is organized as follows. We first give a realistic example that illustrates the need to support mixed granularities. We then define a granularity as a calendar dependent

partitioning of the time-line. Granularities naturally form a hierarchy, or more precisely, a lattice in that some granularities are finer or coarser with respect to others. Next, we extend the syntax and semantics of SQL-92 to permit the definition of timestamps at various granularities. The BNF for the extended syntax is given in Appendix A. We also extend the semantics of temporal operations to handle operands at differing granularities. This query language support rests on two operations, *scale* and *cast*, that convert times from one granularity to another. Indeterminacy is critical to correctly implementing the lattice and operations that move times within the lattice. We also describe the implementation, focusing on maximizing the efficiency of the scale operation. Finally, we summarize our work and related work (pertaining to granularity issues).

7.1 Motivation

Consider the airline flight database depicted in Figure 7.1. The database consists of two relations: *Flight_Departures* and *Vacations*. The *Flight_Departures* relation stores information about airplane flight departures. The flight departure time is recorded in the granularity of minutes. The *Vacations* relation stores information about vacations, specifically, the days that make up a vacation. The temporal information in *Vacations* is ostensibly stored to the granularity of days, with each tuple recording an "interval" of days rather than just a single day. The vacations listed in *Vacations* include traditional holidays such as Labor Day, Christmas, and Thanksgiving. The Thanksgiving vacation is a four day weekend beginning on the fourth Thursday in November.

A user, interested in flying home for Thanksgiving, queries this database to determine which flights leave during the Thanksgiving vacation. In SQL-92, this query might be formulated as follows [Melton & Simon 1993].

```
SELECT *
FROM Vacations, Flight_Departures
WHERE Vacation = 'Thanksgiving' AND
      Flight_Departures.Time OVERLAPS Vacations.Time
```

In this query, the user specifies the `OVERLAPS` operation to determine which flights leave

Flight_Departures(Flight#,Time)

Flight#	Time
53	Nov. 20, 2:38 PM 1994
200	Nov. 25, 2:34 PM 1994
653	Nov. 27, 12:38 PM 1994
658	Nov. 30, 10:03 AM 1994

Vacations(Name,Time)

Vacation	Time
Labor Day	Sep. 1 — Sep. 3 1994
Thanksgiving	Nov. 24 — Nov. 28 1994
Christmas	Dec. 24 — Dec. 26 1994

Figure 7.1: A flight database

during the Thanksgiving vacation. `OVERLAPS` is a temporal intersection operator. However, the times participating in the `OVERLAPS` are at different granularities. To determine the flights that leave during Thanksgiving, the query processor needs information about the relationship between minutes and days. In particular, it needs to know that each minute belongs to a unique day. Consequently, a flight that departs on a given minute during a day also departs on that day. With this extra information the query processor can "scale" or convert the granularity of flight departure times from minutes to days allowing the `OVERLAPS` to determine which flights leave during the Thanksgiving vacation.

Several aspects of this query remain unexplained. Primarily, how does the database know the relationship between days and minutes, why is the `OVERLAPS` performed at the granularity of days rather than minutes, and how are the minutes of flight departures "scaled" to days? In the remainder of this chapter we outline answers to these questions.

7.2 A Discrete Image of Time

The time-line segment can be partitioned into a finite set of smaller segments known as *granules* [Wiederhold et al. 1991]. The partitioning scheme is a *granularity* and is informally described by two pieces of information:

1. a length, or unanchored segment of the time-line, which gives the size of each granule, and

2. an anchor point, which establishes where the partitioning begins.

Both the anchor point and partitioning length are given in time-line clock chronons. A partitioning scheme partitions the time-line into granules, each the size of the partitioning length, beginning from the anchor point, and extending both forwards and backwards along the time-line. The resulting granules are consecutively labeled with their distance from the anchor point. A granularity creates a discrete image, in terms of granules, of a (possibly continuous) time-line. By choosing an appropriate granularity, a user can view the time-line as a discrete set of seconds, or days, or years. The granularity of *chronons*, used in previous chapters, is merely the smallest supported granularity. Figure 7.2 shows a portion of the time-line partitioned into day size granules.

Granularities are calendar-dependent entities. *Calendars* relate times on the time-line clock to more familiar temporal descriptions [Snodgrass et al. 1994]. For example, the Gregorian calendar associates the temporal description "December 9, 1921" with a specific set of time-line clock chronons (a segment of the time-line). Calendars incorporate the cultural, legal, and even business orientation of the user to define the time values that are of interest. For example, an employee time card can be regarded as a calendar which measures time in eight hour increments and is only defined for five days of each week. Although the calendar most familiar to the typical database user is probably the Gregorian calendar, many different calendars exist and no calendar is inherently "better" than another; the value of a particular calendar is wholly determined by the population that uses it.

A granularity anchor point is typically a *calendar* origin. For example, the anchor point for most Gregorian calendar granularities is midnight, January 1, A.D. 1. Different calendars have different anchor points and sometimes there are multiple anchor points within the same calendar. For example, assume that we have a length the size of a Gregorian calendar "day." If we use the anchor-point corresponding to *midnight*, January 1, A.D. 1, then we would partition the time-line into days. But if we choose the anchor to be *noon*, January 1, A.D. 1, then we obtain a different granularity, sometimes called the *civil day*.

In reality, partitioning by using a single, fixed length is impractical since most common granularities carve the time-line into partitions of differing lengths. For instance, a year

varies in length since it could have 365 or 366 days. A month varies in length since it might have 28, 29, 30, or 31 days. But even a day is of varying length since leap seconds can be inserted or deleted [Fraser 1987].

While we will continue to use the informal description of a granularity as an anchor point and a length, formally, a granularity is a partitioning function which maps time-line clock chronons into granules. Not every partitioning function is a granularity. We assume that calendars know which partitioning functions are granularities. For instance, a Gregorian calendar might support granularities of seconds, minutes, hours, days, weeks, fortnights, months, years, and decades. A granularity is, in fact, only a partial function since calendars may be undefined over certain portions of the time-line, e.g., the Gregorian calendar is undefined 10 billion years ago since it is based on the the revolution of the Earth around the Sun, and the Earth did not exist at that time. Another example is the Mayan Long Count calendar. It begins in 3114 B.C. and ends in A.D. 2012 (it is limited by the syntax of dates).

The smallest, possible granularity is that of time-line clock chronons. The largest is "all of time," that is, the entire time-line considered as a single granule. Calendars do not define either the largest or smallest granularities; those granularities are built-in.

Within a given granularity, the set of granules is well-ordered. Two special values, *beginning* and *forever*, are the least and greatest values, respectively, in the ordering. Beginning and forever are not granules; rather, they are instants just outside the closed interval of time and have special operational semantics (operationally, they are treated as $-\infty$ and ∞, respectively) [Dyreson et al. 1993].

Within a calendar, granularities are related in the sense that one granularity may be a further partitioning of another. For example, days are a further partitioning, a *finer* partitioning, of months or weeks. Note that months are not a further partitioning of weeks, or vice-versa. With respect to further partitionings, the set of granularities forms a lattice [Wang et al. 1993]. The lattice for a Gregorian calendar is depicted in Figure 7.3. The top element, $\top$, in the lattice is the granularity of "all of time." The bottom element, $\perp$, in the lattice is the granularity of time-line clock chronons.

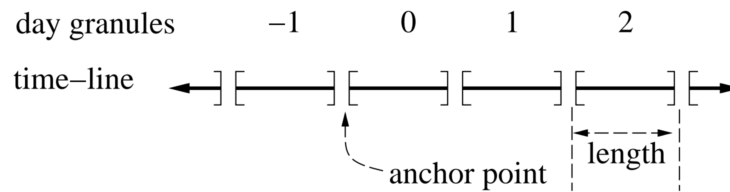


Figure 7.2: The time-line at a granularity of days

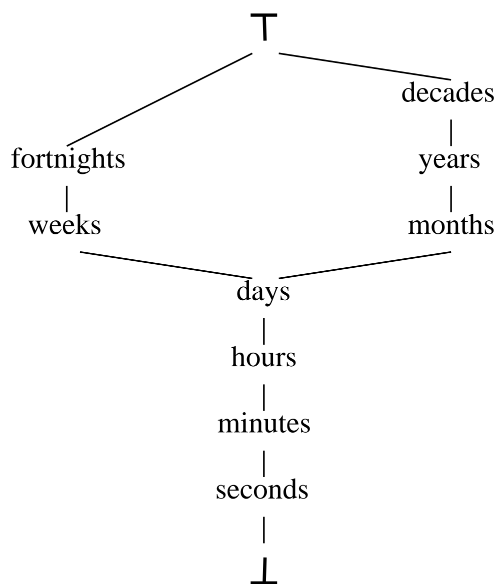


Figure 7.3: A Gregorian calendar granularity lattice

The lattice in Figure 7.3 shows some granularities in a Gregorian calendar. In a multi-calendar system granularities in different calendars are woven together into a single lattice. A granularity is just a partitioning of the time-line, and often, different calendars have the same granularity. For instance, days in the Julian and Gregorian calendars are the same granularity, although the anchor points differ. Granularity equivalences are checked when the system-wide lattice is built, as described further in Section 7.3.1.

7.2.1 Modeling Instants at Various Granularities

Recall that an instant is a point on the time-line. To model instants at various granularities, we use an instant timestamp. An instant timestamp records that an instant is located sometime *during* a particular granule (e.g., a day, a year, a month). The timestamp stores the granule number (an integer) (the actual format is given in Section 7.4.1). Without loss of generality, we assume that an instant timestamp represents any instant during a granule. Hence, at a very abstract level, the *exact* instant modeled by an instant timestamp is never precisely known. Interval and span timestamps are natural extensions of the instant timestamp. In this chapter, we will follow the syntax developed in previous chapters (not TSQL2 syntax, except where noted) and delimit instant *constants* with vertical bars, for example, `|June 1, 1994|`, interval constants with brackets (`[June 1994 - July 1994]`) and span constants with percent signs (`%6 days%`).

7.2.2 Indeterminacy and Granularity

An instant timestamp records that an instant is located sometime during a particular granule. Often, however, we do not know the exact granule during which an instant is located; instead, we know that the instant is located sometime during a set or range of granules. We call such an instant an *indeterminate* instant. For example, we may know that a plane left sometime on June 12, 1994, but, at the granularity of a minute, we do not know the exact minute during which that plane departed.

Granularity and indeterminacy are two sides of the same coin. A general maxim is that a determinate instant is indeterminate with respect to *all* finer granularities. For instance, suppose that at the granularity of days we record that a plane took off on June 12, 1994. At the granularity of hours, the departure time is indeterminate since we did not record the exact hour that the plane departed; we only know that it left sometime during a 24 hour period. Conversely, an indeterminate instant is determinate with respect to *some* coarser granularity. For example, suppose we record, at the granularity of hours, that a flight departs sometime between 2 P.M. and 4 P.M. on June 12, 1994. At the

granularity of days, months, and years, the flight departs wholly within a single granule: `|June 12, 1994|`, `|June 1994|`, and `|1994|`, respectively.

7.3 A Proposal for Accommodating Mixed Granularities

In this section, we propose a syntax and semantics for supporting mixed granularities. Implementation details are discussed in Section 7.4. The proposed support for mixed granularities is based on the SQL-92 language standard [Melton & Simon 1993], and has been accepted into the consensus temporal query language TSQL2 [Dyreson & Snodgrass 1994D].

We first describe how the granularity lattice is constructed during database implementation. We then propose a syntax for specifying the granularity of timestamps. The fact that there are timestamps at different granularities impacts the semantics of temporal operations. As the default semantics, we adopt *coarse granularity semantics* which converts operands to the coarsest granularity prior to the operation. We propose two operations that make granularity conversions and show how these operations can be used to implement the desired semantics.

7.3.1 Building the Lattice

There are many methods that could be used to build the granularity lattice. Perhaps the easiest is to assume that the lattice is built-in. We feel that such an assumption is unrealistic. Instead, we outline a method for building the lattice using a specification provided by the database administrator (DBA). We note in passing that the MULTICAL system [Soo et al. 1992] is a concrete realization of this approach to supporting multiple calendars in a conventional DBMS.

In this approach, each calendar has a *specification file* which contains calendar specific information, such as the calendar origin. The calendar specification files are parsed when the DBMS is configured by the DBA. We propose to add granularity descriptions to each specification file. In the specification file, a granularity is described as a further partitioning of some other granularity using either an "irregular" mapping (i.e., a C function) or a "regular" mapping into groups of size n. An anchor point must also be

given for each granularity. Below is an example from a Gregorian calendar specification file that uses both kinds of descriptions.

```
GRANULARITY seconds PARTITIONS chronons IRREGULAR
       WITH ANCHOR gregorian_origin;
GRANULARITY minutes PARTITIONS seconds IRREGULAR
       WITH ANCHOR gregorian_origin;
GRANULARITY hours PARTITIONS minutes REGULAR USING 60
       WITH ANCHOR gregorian_origin;
GRANULARITY days PARTITIONS hours REGULAR USING 24
       WITH ANCHOR gregorian_origin;
       . . .
GRANULARITY decades PARTITIONS years REGULAR USING 12
       WITH ANCHOR gregorian_year_boundary_2000;
```

The lattice built by this specification file is shown in Figure 7.4. The lattice is decorated with "upward" and "downward" edges between the granularities. Each edge represent a mapping. A dashed edge is an irregular mapping while a solid edge is a regular mapping. Each edge is labeled with the mapping function between those granularities. The granularities of `chronons` ($\perp$) and `all_of_time` ($\top$) are built-in.

The example specification file describes the granularity of hours as a regular partitioning of minutes. Further, it asserts that there are sixty minutes in an hour. In the granularity lattice, this information is represented by a solid edge connecting minutes and hours, labeled with a *60*. What may be surprising to some readers is that the specification does not assume that there are sixty seconds in a minute. This is because there may be sixty-one seconds in a minute due to leap second insertions. Instead, the specification file states that an irregular mapping exists between minutes and seconds. Irregular mappings are C functions that the calendar provides. For every irregular mapping, a calendar must have both an "upward" mapping function (e.g., `seconds_to_minutes`) and a "downward" mapping function (e.g., `minutes_to_seconds`). The name of the irregular

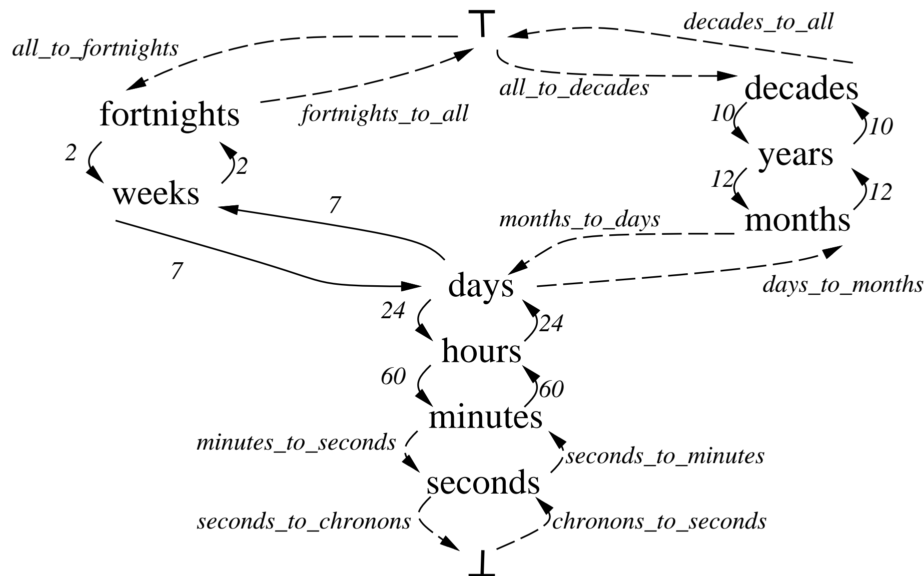


Figure 7.4: The Gregorian calendar granularity lattice with mappings between granularities

mapping function, e.g., `seconds_to_minutes`, is inferred from the source and target granularities. Alternatively, the calendar specification file could explicitly list the name of the mapping function with a separate `MAPPING` clause as follows.

```
MAPPING days TO months USING leap_month_mapping;
```

Although the `seconds_to_minutes` function is simple (we describe elsewhere how to process Gregorian calendar dates with leap seconds [Dyreson & Snodgrass 1994E]), some irregular mappings may be quite complex. The `days_to_months` mapping must accommodate months that have different numbers of days, as well as leap years.

In the example Gregorian calendar specification, note that the granularity of decades has a different anchor point than the other granularities.[1] If the anchor point for decades were the Gregorian origin, midnight January 1, A.D. 1, then the ten years between (inclusive) 1971 and 1980 would be a decade rather than the ten years between (inclusive)

[1] We assume that the definitions of the anchor points, `gregorian_origin` and `gregorian_year_boundary_2000` in terms of time-line clock chronons are given elsewhere in the calendar specification file (presumably before the granularities are described).

1970 and 1979. The latter period, but not the former, fits the common definition of a decade. Consequently, the anchor point for decades cannot be the Gregorian calendar origin, but must be a decade (and year) boundary (such as the start of the year A.D. 2000). When the anchor points differ between two granularities, semantic checking is performed to determine if the anchor point of the coarser granularity is on a granule boundary of the finer granularity. In general, the granularity descriptions are checked when the specification file is parsed during database construction. The irregular functions cannot be checked for errors and are assumed to be valid.

During construction of the lattice, pairs of granularities are checked for *congruence*. Two granularities, perhaps from different calendars, are congruent if they partition the time-line into identical sets, even though they may have different anchor points. For example, suppose that Gregorian and Julian days partition the time-line similarly, but that these two granularities have different anchor points. Then the 23rd Gregorian day is not the same day as the 23rd Julian day; instead it is the 100023th Julian day (assuming that the Gregorian day anchor point is 100000 days later than the Julian day anchor point). Congruent granularities are very useful for saving space in timestamps (see Section 7.4.1). In the granularity lattice, they are connected by regular mapping edges labeled with a *1*. Conceptually, granularities may be tested for congruence by enumerating the set of time-line clock chronons, converting each chronon to the given granularity (using the *scale* operation described below), and seeing whether it belongs to the same or a different partition (modulo partition labels). Various simple optimizations can be applied to this brute-force approach.

7.3.2 Temporal constants

Instant, interval, and span constants are syntactically delimited by special characters, "`||`", "`[]`", and "`%%`", respectively (or their SQL-92 equivalents, e.g., "`TIMESTAMP''`", "`PERIOD'[]'`", and "`INTERVAL''`", respectively). A calendar translates whatever comes between the delimiters into an instant, interval, or span timestamp. We assume that the calendar also decides the granularity of that timestamp. For example, it is natural to interpret that the instant constant `|June 12, 1994|` is given to the granularity of days.

We believe that calendars should provide this interpretation, however, we do not require them to do so. Consequently, the calendar may decide that the constant has a granularity of years (losing some information), or seconds (creating some information). For the constants presented in this dissertation, we always assume the “natural” interpretation.

7.3.3 Column Definitions

In SQL-92, the create table statement adds relation schemas to the system catalog. In a temporal extension to SQL-92, columns (attributes) of instant, interval, or span timestamps can be defined as part of the create table statement as follows.

```
CREATE TABLE Vacations (Name CHAR[30], Time PERIOD);
CREATE TABLE Flight_Departures (FlightNum INTEGER,
                                Time DATE);
```

Details are presented elsewhere [Snodgrass et al. 1994].

Lacking from these column definitions, however, is a way to specify the timestamp granularity. We propose to allow the user to specify a *range* and granularity at column definition. Conceptually, range is how much time is represented. For example, to define an instant timestamp that can store times known to the granularity of a second that are within 18,000 years of the granularity anchor point, we propose to use

```
TIMESTAMP(%18000 years%,seconds)
```

The span constant in the column definition is parsed by a specific calendar (calendar scoping rules are described elsewhere [Snodgrass et al. 1994]). The specified granularity, however, is a system-wide name. In an SQL-92 implementation, the name is a user-defined, nullary function symbol that is made known to the database when the granularity lattice is built. The default granularity is `seconds`. The default range is `%120 years%`. The rationale behind these defaults is given in Section 7.4.1. The following declaration

specifies an instant timestamp that can store times within thirty-six billion years of the granularity anchor to the granularity of a microsecond.

```
TIMESTAMP(%36 billion years%,microseconds)
```

The syntax for column definition is similar to SQL-92's datetime definition and provides a great deal of flexibility, as can be imagined [Melton & Simon 1993]. Like SQL-92, we assume that a column definition establishes a data-type that is the same for every value in that column, e.g., in a column of integer type, every value is an integer (or a null value). All timestamps in a column are stored to the same granularity. In our example database, all flight times are stored to the granularity of a minute, rather than some being stored to finer granularities, such as a second or millisecond. Times known to coarser granularities (e.g., a day) can be stored by making the indeterminacy explicit (e.g., the flight leaves between the first and last minutes during that day). Also, the granularity of a timestamp can be stored with the schema rather than in the timestamp. This results in smaller timestamp formats, as described further in Section 7.4.1.

It is important to note that the syntactic extensions for granularity coexist with those for indeterminacy discussed in Chapter 6.

Creating an interval-timestamped column is similar to creating an instant-timestamped column. The interval data-type definition has both a range and a granularity, as does the instant declaration. The instants that start and end the interval share the same range and granularity. For example,

```
PERIOD(%18000 years%,microseconds)
```

establishes an interval that has bounding instants with a granularity of microseconds and a range of 18,000 years.

Specification of span columns is similar to that of instant columns. The span data type declaration uses both a range and a granularity, as does the instant declaration. For example,

```
INTERVAL(%40 years%,years)
```

would establish a span with a granularity of `years` and a range of forty years. The range is interpreted slightly differently for a span. The range limits the maximum duration of the span, rather than the maximum distance from the calendar origin. In this example, spans stored in this column can be no longer than forty years.

7.3.4 Granularity in Operations

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

Consider a binary temporal operation, such as `OVERLAPS` or addition, involving operands at differing granularities. There are several possible semantics for the operation.

- Give a mismatched granularity error. We feel that this option is too inflexible since operations over differing granularities will be common.

- Perform the operation at the granularity of the first operand, as proposed by the SQL-92 language standard [Melton & Simon 1993]. The advantage of this strategy is that the granularity of the operation can be controlled by swapping the operands. The disadvantage is that symmetric operations are no longer symmetric, e.g., `A OVERLAPS B` is not the same thing as `B OVERLAPS A`, since the operation is performed to and the result given in the granularity of the first operand.

- Perform the operation to the finer granularity. In order to perform the operation at the finer granularity, both operands must be rendered in that granularity. For example, to compare an instant at the granularity of an hour to an instant at the granularity of a day, the day instant must be converted to hours prior to performing the operation. The problem is that the conversion creates an indeterminate instant. That is, if we know that an instant is located sometime during a particular day, then we cannot assert that it is located sometime during a particular hour, rather we only know that it is located sometime during a 24-hour period.

- Perform the operation to the coarser granularity.

We will describe user-level operations that can easily support these various semantics (plus others). As the default semantics we adopt the second semantics from above to remain consistent with SQL-92. But a user can utilize the operations that we describe below to implement any desired semantics.

All the semantics rest on operations that move times within the granularity lattice. Converting from a finer to a coarser granularity moves a time "up" the granularity lattice. The conversion is called a *scale* operation and is described in the next section.

As an example, consider an operation that tests the overlap of an instant known to the granularity of a minute and an interval known to the granularity of a day. To implement the coarse granularity semantics, the instant known to the granularity of a minute is "scaled" to the day that contains that minute. The overlap test is then performed at the granularity of an day. Informally, the entire operation is akin to asking, "is this instant located during the same day as some portion of the interval?" Note that the semantics of an operation fundamentally depends on the granularity of the operands. At the granularity of a year, the overlap would test "is this instant located during the same year as some portion of the interval?" At the granularity of a minute, the overlap would test "is this instant located during the same minute as some portion of the interval?" In a database that supports indeterminacy, a user may also use plausibility and credibility phrases to pose question such as "is it *possible* that this instant is located during the same minute as some portion of the interval?"

7.3.5 Scale

We propose to add an operation, called *scale*, that moves a time "up" (or "down") the granularity lattice. The informal syntax of scale is

$$\texttt{scale(}\ \langle \textit{operand} \rangle\texttt{,}\langle \textit{granularity} \rangle\ \texttt{)}$$

The first argument to scale is an operand. The original granularity of this operand is locally overridden by the scale operation. A temporal operand can either be a constant, a column variable, or the result of an expression. If the operand is a constant, the granularity of

that constant is given by the calendar that parses the constant. If the operand is a column variable, the granularity of that variable is given by the column definition (specified in the schema). Finally, if the operand is an expression, the granularity of the operand is the granularity of the result of that expression. The second argument to scale is the target granularity. For example,

```
scale(Flight_Departures.Time,days)
```

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

Scale is a very simple operation. However, the internal mechanics of scale are complex. Before considering the semantics of scale, we give some examples that illustrate scale's simple behavior.

```
scale(|June 1, 1994|, centuries) = |20th|
scale(|June 1, 1994|, years) = |1994|
scale(|June 1, 1994|, months) = |June 1994|
scale(|June 1, 1994|, days) = |June 1, 1994|
scale(|June 1, 1994|, hours)
         = |0 hours June 1, 1994 ~ 23 hours June 1, 1994|
scale(|June 1, 1994|, minutes)
         = |00:00 June 1, 1994 ~ 23:59 June 1, 1994|
scale(|June 1, 1994|, seconds)
         = |00:00:00 June 1, 1994 ~ 23:59:59 June 1, 1994|
```

Notice how the result of the scale switches from determinate to indeterminate as the target granularity moves from coarser to finer granularities. This is a key feature of scale. Scaling a determinate instant from a finer to a coarser granularity results in a determinate instant. But scaling a determinate instant from a coarser to a finer granularity results in an indeterminate instant.

Scale uses the regular and irregular mappings between granularities to convert a time from one granularity to another. the mappings are recorded in the granularity lattice.

The mechanics of scale depends on whether scale moves a time from a finer to a coarser granularity or from a coarser to a finer granularity. We discuss the finer to coarser scaling first, focusing on scaling determinate instants.

Suppose that we wish to scale an instant, $time_f$, from a finer granularity, f, to a coarser granularity, c, resulting in the instant, $time_c$. Let $diff_{f,c}$ be the difference between the granularity anchor points of f and c, expressed in the finer granularity, f, and $map_{f_to_c}$ be the regular or irregular mapping from the finer to the coarser granularity. Initially we assume that there are no "intervening" granularities between f and c; instead they are directly related by $map_{f_to_c}$. Then

$$\texttt{scale}(time_f, c) = map_{f_to_c}(time_f - diff_{f,c}) = time_c.$$

If $map_{f_to_c}$ is a regular mapping, then the mapping function is a simple integer division. For example, suppose we want to scale a time from minutes to hours. Since minutes and hours have the same anchor points,

$$\texttt{scale}(time_f, \texttt{hours}) = time_f \textbf{ div } 60 = time_c.$$

This operation scales any minute in $\{0, \ldots, 59\}$ to hour 0, any minute in $\{60, \ldots, 119\}$ to hour 1, etc.

Now suppose that granularity f' lies between f and c in the granularity lattice. Then

$$\texttt{scale}(time_f, c) = map_{f'_to_c}(map_{f_to_f'}(time_f - diff_{f,f'}) - diff_{f',c}) = time_c.$$

So to scale from minutes to days, the scale function would be

$$\begin{aligned}
&\texttt{scale}(time_f, \texttt{days}) \\
&\qquad = \texttt{scale}(\texttt{scale}(time_f, \texttt{hours}), \texttt{days}) \\
&\qquad = (time_f \textbf{ div } 60) \textbf{ div } 24 \\
&\qquad = time_c.
\end{aligned}$$

In general, by composing scale operations, a time may be moved several granularities in the lattice. Scale operations can be composed since they are functions.

Times may also be moved down the granularity lattice. The mechanics of scaling a determinate instant from a coarser to a finer granularity are given below.

$$\texttt{scale}(time_c, f) = (map_{c_to_f}(time_c - diff_{c,f})) \sim (map_{c_to_f}((time_c + 1) - diff_{c,f}) - 1)$$

If $map_{c_to_f}$ is a regular mapping, then the mapping function is a simple integer multiplication. For example, suppose we want to scale a time from hours to minutes. Since hours and minutes have identical anchor points,

$$\begin{aligned} &\texttt{scale}(time_c, \texttt{minutes}) \\ &\qquad = time_c \times 60 \sim ((time_c + 1) \times 60) - 1 \\ &\qquad = time_c \times 60 \sim (time_c \times 60) + 59. \end{aligned}$$

This operation scales hour 0 to $0 \sim 59$ minutes, hour 1 to $60 \sim 119$ minutes, etc.

To scale an indeterminate instant, the upper and lower supports are scaled separately. When scaling from a coarser to a finer granularity, each support is scaled to a pair of times. In such cases, the period of indeterminacy is maximized by choosing the minimum time for the lower support and the maximum time for the upper support.

```
scale(|June 1994 ~ July 1994|, days)
            = |June 1, 1994 ~ July 31, 1994|
```

Since all times are related at $\top$ and $\bot$ in the lattice, an instant at any granularity can be converted to an instant at any other granularity by moving up and down the lattice. The general rule to scale from one granularity to another is to move the time down to the greatest lower bound of the two granularities, and then up to the target granularity [Wang et al. 1993]. An up move results in an overall "loss" of information. For example, in scaling from microseconds to seconds, the number of microseconds within that second (6 digits of information) is removed from the timestamp. A down move will convert determinate to indeterminate information, refining the information content of the timestamp. Scale never "creates" information. No composition of scale operations will ever relocate an instant elsewhere on the time-line nor will it ever result in a timestamp that has a period of indeterminacy shorter than that of the original timestamp.

7.3.6 Scaling Intervals and Spans

To scale an interval, the instants that start and end the interval are scaled separately. Scaling spans, however, is slightly more complex.

A span is an unanchored duration. A span of `%1 day%` represents a duration that when added to an instant at the granularity of days, will displace that instant by one day. So,

`|December 30, 1994| + %1 day% = |December 31, 1994|`

and

`|December 31, 1994| + %1 day% = |January 1, 1995|.`

But a span of `%1 day%` also represents a duration that when added to an instant at the granularity of months, could displace that instant by 0 or 1 months. In the above example, the instant could be moved from the month of December to the month of January. Note that the span of `%1 day%` could also displace an instant into the next year.

The problem is that a granularity is an anchored partitioning, whereas a span is unanchored. Imagine taking a span and placing it anywhere along a time-line that is partitioned into granules. Depending upon where we place the span, it will cross more or fewer granules as shown in Figure 7.5. Even the smallest span will cross at least one granule boundary.

The consequence of the unanchored nature of spans is that whenever a span is scaled, an indeterminate span will result, even when a determinate span is scaled from a finer to a coarser granularity. Below, we give some examples to illustrate the unexpected behavior of scaling a span.

```
scale(%1 day%, centuries) = %0 centuries ~ 1 century%
scale(%1 day%, years) = %0 years ~ 1 year%
scale(%1 day%, months) = %0 months ~ 1 month%
scale(%1 day%, days) = %1 day%
scale(%1 day%, hours) = %1 hour ~ 47 hours%
```

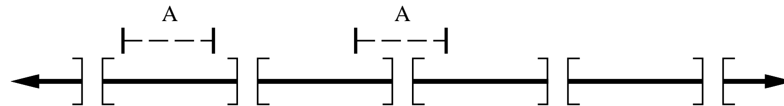


Figure 7.5: Span A can be placed within a single granule or spanning two granules

```
scale(%1 day%, minutes) = %1 minute ~ 2879 minutes%
```

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

7.3.7 Scaling Mass Functions

The probability mass function gives the probability that the instant is located within a given granule. Since a scale operation modifies the size and number of granules in a period of indeterminacy, it also changes the mass function. Each mass function is described, in the implementation, as a function at the level of chronons (actually, the implementation only *approximates* this granularity, but the approximation is discussed elsewhere). In scaling from a finer to a coarser granularity, the mass of each fine granule is added to the mass of all the other fine granules that belong to a given coarse granule. For example, assume that an indeterminate instant with a seven day period of indeterminacy (from Sunday through Saturday) and a *uniform* mass function is scaled to the granularity of weeks. In the resulting instant, the probability that the instant is located during each day, a probability of $\frac{1}{7}$, is accumulated to give the probability that the instant is located during the given week, a probability of 1. In scaling from a coarser to a finer granularity, the mass of each coarse granule is, in effect, "dispersed." The mass functions are defined at the level of chronons. Any granularity above that level aggregates the mass at each chronon into the mass at a granule. In scaling from a coarser to a finer granularity, the mass is reaggregated at the finer granularity.

7.3.8 Cast

A scale operation that converts an instant from a coarser to a finer granularity produces an indeterminate instant. But, for various reasons, a user may not want an indeterminate result. Instead, a user might want an operation, like scale, that always produces a determinate timestamp, even though that result might not be strictly consistent with the recorded data. To meet these user needs, we propose a new operation, called *cast*, that allows one to "create" information.

The cast operation is similar to scale but always produces a determinate timestamp. In those situations where scale would return an indeterminate timestamp, cast simply chooses a single possibility, always the first possibility, from the indeterminate result. For example, to cast a determinate instant from a coarser to a finer granularity, cast first scales the instant, resulting in an indeterminate instant. From that indeterminate instant, it selects the first instant in the period of indeterminacy. It returns this single instant, which is modeled by a determinate timestamp, as the result. In effect, for any instant timestamp, cast assumes that the timestamp always models the first instant at all finer granularities. To draw an analogy with fixed-precision numbers, cast is a "pad with zeros" operation. The number 3.45 casted to five decimal places would be 3.45000.

We could define an operation similar to cast that chooses the last instant in a period of indeterminacy, but this operation is not needed to implement the SQL-92 semantics; although it could easily be added.

The syntax of cast is similar to that of scale.

$$\texttt{cast(}\ \langle \textit{operand} \rangle \texttt{,} \langle \textit{granularity} \rangle\ \texttt{)}$$

The semantics of cast is also similar to that of scale, with one exception: when casting from a coarser to a finer granularity only the first time is produced. Examples of cast will make this distinction clear.

```
cast(|June 1, 1994|, centuries) = |20th|
cast(|June 1, 1994|, years) = |1994|
cast(|June 1, 1994|, months) = |June 1994|
```

```
cast(|June 1, 1994|, days) = |June 1, 1994|
cast(|June 1, 1994|, hours)
                        = |0 hours June 1, 1994|
cast(|June 1, 1994|, minutes)
                        = |00:00 June 1, 1994|
cast(|June 1, 1994|, seconds)
                        = |00:00:00 June 1, 1994|
```

Compare these results with the examples of scale given in the previous section. No indeterminate instants are produced. An cast on an indeterminate instant may, however, still produce an indeterminate instant, for example

```
cast(|June 1994 ~ July 1994|, days)
                        = |June 1, 1994 ~ July 1, 1994|
```

Unlike the scale, the cast does not maximize the period of indeterminacy.

7.3.9 Enforcing SQL-92 Semantics

The SQL-92 semantics for temporal operations is enforced by implicitly inserting cast operations, as needed, to convert the second operand to the granularity of the first operand. Cast is used rather than scale since SQL-92 does not have indeterminacy. The query processor inserts the cast operations during the query-rewrite phase.

Although we adopt SQL-92 semantics in this chapter, any of the possible semantics (save generating the mismatched granularity error) can be implemented by inserting scale and cast operations. For example, to implement the coarse granularity semantics, the finer operand is converted to the granularity of the coarser operand by inserting a scale operation on the finer operand. Another option would be to introduce a compile or run-time “flag” to toggle among the various semantics.

It is important to note that an individual user who desires a different temporal semantics can explicitly insert scale or cast operations. User-specified granularity conversions override query-rewrite insertions.

7.3.10 Defaults

The default scale (cast) on an operand may be globally overridden by a SET statement. We propose that the user can set the global scale in a set statement, e.g.,

```
SET SCALE microseconds # Set global scale to microseconds
SET CAST microseconds # Set global cast to microseconds
```

Setting a global scale (cast) adds a `scale()` (`cast()`) operation to every operand, excepting those make explicit calls to `scale()` or `cast()`. This has the effect of performing all operations at a given granularity. Since both kinds of set statements modify all the operands in a query, the most recently executed set scale or cast takes precedence.

Initially there is no global scale (cast), the scale (cast) of operands are given by the operands. Once set, the global scale and cast can be removed as follows:

```
SET SCALE AS DEFAULT
```

7.3.11 Processing of Example Query

We use the techniques developed in previous sections to process the example query given in Section 7.1. First, the query is parsed under coarser granularity semantics and rewritten to the following query.

```
SELECT *
FROM Vacations, Flight_Departures
WHERE Vacation = 'Thanksgiving' AND
       (scale(Flight_Departures.Time,days)
        OVERLAPS Vacations.Time)
```

The granularity lattice is consulted to determine how to scale from minutes to days. Scaling from minutes to days first requires scaling from minutes to hours and then from

Flight_Departures(Flight#,Time)

Flight#	Time
53	Nov. 20
200	Nov. 25
653	Nov. 27
658	Nov. 30

Figure 7.6: Flight times scaled to days

hours to days. Both mapping are regular and the anchor differences are zero, so

$$\begin{aligned}&\texttt{scale(}\mathit{Flight_Departures.Time}\texttt{,days)}\\&\quad=\texttt{scale(scale(}\mathit{Flight_Departures.Time}\texttt{,hours),days)}\\&\quad=(\mathit{Flight_Departures.Time}\ \mathbf{div}\ \texttt{60)}\ \mathbf{div}\ \texttt{24}.\end{aligned}$$

The effect of the scale on each time is depicted in Figure 7.6. The scaled time of flight# 200 and of flight# 653 overlaps November 24 - November 28, so only these two tuples appear in the answer.

7.4 Implementation

In this chapter we have introduced timestamps that model temporal information at various granularities and two operations that convert times between granularities. The added modeling capabilities carry an associated space and time cost. In this section we quantify the cost of supporting mixed granularities. In a previous chapter, we described simple and compact timestamp formats that store times. These formats remain unchanged, however, range and granularity dictate the size of the format to use. We also outline the implementation of scale and cast. Although these operations may appear expensive, we present several optimization strategies that mitigate the expense.

7.4.1 Impact of Granularity on Timestamp Formats

The only change to the timestamp formats discussed in Chapter 5 is that the data field stores the granule number rather than the chronon number. The granule number is a count, in granules, of the distance from the granularity anchor point to the instant. Since all timestamps in a column have the same granularity, the granularity is stored with the schema rather than stored with the timestamp, thus eliminating a field to store the name of the granularity.

In a timestamp definition, the range controls the maximum permissible count. For example,

```
INSTANT(%18000 years%,seconds)
```

defines a timestamp that can have a count of at most 36,000 years ($\pm 18,000$ years) worth of seconds, or approximately 1.2 trillion seconds. In general, the number of bits needed to store the maximum count varies, depending on the range and granularity. For a large range and small granularity, the maximum allowable count is quite large, while for a small range and large granularity, the maximum count is small. Consequently, the size of the field to store the count, and in turn, the size of an instant timestamp, varies. The following is a rough guideline for determining the size (in bits) of the field to store the count,

$$\lceil \mathbf{log}_2(\mathbf{max}(\texttt{scale}(range, granularity)))\rceil + 1.$$

The max function chooses the longest possible span from the indeterminate span returned by scale [Dyreson & Snodgrass 1994C]. Since the count could be positive or negative, one bit is added to the size to account for the sign bit. The size of the count field is further adjusted to keep the format on a 32-bit word boundary. Table 7.1 shows some examples of instant specifications that might arise in practice (the span `%all_of_time%` is approximately 36 billion years in our model of time). As a comparison, the SQL-92 `TIMESTAMP` format storing the maximum allowable range (9999 years) to the granularity of seconds requires 19 *positions*, which may be conveniently stored as BCD digits in two and a half words [Melton & Simon 1993]. Note that the defaults on an instant timestamp

definition, a granularity of `seconds` and a range of `%120 years%`, builds a one-word format (the default is essentially the UNIX format).

Congruent granularities can be used to limit the size of timestamps. A user that wants to store times in the current decade to the granularity of a second can use a one word format by using a granularity, congruent to seconds, but with an anchor point at the current decade boundary. By relocating the anchor point via congruent granularities, the user can use one word timestamps for most applications.

7.4.2 Scale and Cast

Since cast is based on scale, we describe only the implementation of scale.

Scale is performed in the "inner-loop" of query processing, potentially done many times during a query. Each scale costs (possibly) one subtraction (for the anchor difference) and one "expensive" suboperation. For a regular mapping from a finer to a coarser granularity the expensive suboperation is a division. On some machines (e.g., Sun-4s) division is micro-coded as repeated subtraction, often costing much more than addition or multiplication. For a regular mapping from a coarser to a finer granularity the expensive suboperation is a multiplication (note that the second multiplication can be done with an addition). For an irregular mapping, it is a C function invocation, which probably uses a division or a multiplication. In this section, we present four optimization strategies that are designed to minimize the cost of the "expensive suboperation."

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

The second optimization exploits the fact that multiplication is much cheaper than division (on many machines). This optimization applies only to temporal comparisons, but we anticipate that comparisons will be the most common kind of temporal operation. All comparisons, including `OVERLAPS`, are expressed as formulas involving the *Before* operation (a $<$ ordering) and logical connectives [Snodgrass 1987]. Consider a *Before* between

Timestamp Definition	Count Size $\Rightarrow$ Format Size
`INSTANT (%18000 years%,days)`	20 bits $\Rightarrow$ one word
`INSTANT (%10000 years%,seconds)`	39 bits $\Rightarrow$ two words
`INSTANT (%18000 years%,microseconds)`	60 bits $\Rightarrow$ two words
`INSTANT (%all_of_time%,seconds)`	60 bits $\Rightarrow$ two words
`INSTANT (%all_of_time%,nanoseconds)`	90 bits $\Rightarrow$ three words

Table 7.1: Sizes of some common instant timestamps

two determinate instants, α and β. If the granularity of α, G_α, is finer than the granularity of β, G_β, then α is before β at the coarser granularity (i.e., $\texttt{scale}(\alpha, G_\beta)\ Before\ \beta$), if and only if α is before β at the finer granularity (i.e., $\alpha\ Before\ \texttt{cast}(\beta, G_\alpha)$). So a temporal comparison with a scale on one operand can be transformed into a comparison with an cast on the other operand. This program transformation trades a scale from a finer to a coarser granularity (a division) for an cast from a coarser to a finer granularity (a multiplication). The query processor can choose the cheaper operation (in this case the cast), but must factor into the decision how many times the cast or scale is executed. If α and β are column variables and there are far fewer timestamps in α's column, then the transformation will not improve performance since many more casts of β will be performed than scales of α.

A third optimization is to introduce a direct link into the granularity lattice for a highly optimized mapping function. For example, if the database implementor knows that casting years to seconds will be a common operation, a direct link with the name of the optimized mapping function can be inserting into the granularity lattice during its construction. In casting years to days, the run-time engine will use this direct link rather than the composition of years to months, months to days, days to hours, hours to minutes, and minutes to seconds, which costs three regular and two irregular mappings in total. Elsewhere we show that casting years to days requires only eight microseconds on a Sun-4 IPC [Dyreson & Snodgrass 1994E].

The final optimization is to use a *lazy caching* strategy to avoid recomputing previously scaled times. The caching strategy is based on the observation that times in a column of timestamps are often clustered rather than distributed uniformly over the entire timeline (random sampling could be used to detect the clustering). Consequently, there are probably many cases where several instants at the finer granularity scale to the same instant at the coarser granularity. For example, instants in a column of employee birthdates will be clustered between 1934 and 1974 (most employees are between twenty and sixty years old). Assume that these birthdates, stored to the granularity of days, are compared to a column at the granularity of years (e.g., in computing a bar graph of employee ages). In a large corporation, it is probably the case that several employees were born in the same year. To avoid recomputing the cast of years to days (introduced by a previously discussed optimization), we can cache previously computed casts using a small array (there are only forty years between 1934 and 1974). The viability of the caching strategy is a trade-off between the cost of building and maintaining the cache and the cost of cache misses.

7.4.3 Empirical Results

To quantify the actual cost of supporting queries on mixed granularities, we programmed the example query, under a variety of optimization strategies, as a series of calls in the MULTICAL system. The call sequences are shown in Figure 7.7. The variables `f` and `v` are the column variables for the `Flight_Departures.Time` and `Vacations.Time`, respectively. The `unpack` operations parse the timestamp flags to distinguish determinate from indeterminate and special instants. We ignored the "Thanksgiving" selection and coded the `OVERLAPS` as a conjunction of *Before* operations with no short circuit evaluation. The first sequence (from left to right) is an overlap with no support for mixed granularities. Testing this sequence will give us a base cost against which we can compare the cost of modeling and using information at mixed granularities. The second sequence scales minutes to days, using the algebraic optimization. We will also test the unoptimized sequence; that code is not shown. The third sequence combines the algebraic simplification with the program transformation that trades a scale for an cast.

We compiled all four tests using the GNU C compiler, version 2.4.5, with compiler optimizations fully enabled. We then ran the tests several million times on a dedicated Sun-4 IPC (a twelve "mips" machine). The results we obtained are as follows. For the no granularity test, each predicate evaluation took approximately 10 microseconds, for the unoptimized test with one scale, 48 microseconds, for the algebraic optimization test, 27 microseconds, and for the program transformation test, 14 microseconds.

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

7.5 Related Work on Granularity

Our work can be viewed as an extension of Anderson's pioneering research on a model of time [Anderson 1982]. Anderson pointed out the need to model times at multiple granularities. Clifford and Rao further developed Anderson's framework by adding a "granularity chain" (a complete ordering of granularities) and "downward" granularity conversions between times [Clifford & Rao 1987]. Wiederhold, Jajodia, and Litwin made Clifford and Rao's theoretical work more concrete by proposing a specific semantics for temporal comparisons at mixed granularities [Wiederhold et al. 1991]. Their proposed semantics is similar to the finer granularity semantics mentioned in this chapter. Recently, Wang, Jajodia, and Subrahmanian generalized the "granularity chain" to a lattice and proposed semantics for moving times "up" and "down" the lattice [Wang et al. 1993]. Their downward conversion "creates" information. For example, if we record that John earned a salary of $50,000 in 1991, then at the granularity of days, John earned $\frac{\$50{,}000}{365}$. However, this inference is not supported by the stored data since John may have changed pay scales several times during the year; the inference invokes an extra uniformity as-

```
{ The no granularity sequence }
    unpack_event(f);
    unpack_interval(v);
    c1 := Before(f,v.to);
    c2 := Before(v.from,f);
    if (c1 and c2) then
        add f,v to result
```

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

```
{ The fully optimized sequence }
    unpack_event(f);
    unpack_interval(v);
    v.from := v.from × 1440;
    v.to := ((v.to + 1) × 1440) − 1;
    c1 := Before(f,v.to);
    c2 := Before(v.from,f);
    if (c1 and c2) then
        add f,v to result
```

Figure 7.7: MULTICAL calls for example queries

sumption to create a daily account of John's salary. They also presented semantics for temporal comparisons at mixed granularities.

None of these papers address the issue of cost or the integration of granularities from multiple calendars; the latter three also lack indeterminacy. Each is based on the notion that an instant at a given granularity is a "segment" of the time-line. In these frameworks, suppose that we convert two instants located in the same hour and stored at the granularity of an hour to the granularity of a minute. We then query whether they overlap. We will always obtain an affirmative answer because the two instants are identical time-line segments.

In contrast, we treat all instants as indeterminate. When two instants located in the same hour are scaled to a finer granularity, two similar indeterminate instants result. But each indeterminate instant retains the semantics of the original instant in that it records that the instant is located sometime during the hour (with the upper and lower supports expressed in the finer granularity). The user can then query whether these instants "definitely" overlap (never) or "possibly" overlap (always). It is our position that indeterminacy is necessary to support "downward" granularity mappings and to correctly model instants.

7.6 Chapter Summary

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

In this chapter, we described a granularity as a partitioning of the time-line. We then showed that granularities naturally form a hierarchy, or more precisely, a lattice, in that some granularities are finer or coarser with respect to others. Although granularities come from many different calendars, all granularities are related in a single, system-wide lattice. Next, we described how the granularity lattice is built. During construction, "up" and "down" mapping functions are inserted between some pairs of granularities. These

mapping functions are used by *scale* and *cast* to move times from one granularity to the next in the lattice. Judicious use of scale and cast can implement a variety of semantics for temporal operations. We adopted coarse granularity semantics and show that it can be enforced by implicitly inserting scale operations during query analysis. The scale operation does not create information; rather it exploits the relationship between granularity and indeterminacy to refine the information content of a timestamp. A determinate instance, stored to a particular granularity, becomes indeterminate when scaled to a finer granularity. Support for indeterminacy permits conversions between granularities which heretofore might have been considered incomparable, such as weeks and months. Finally, we explored the cost of our framework. We presented four optimizations that can be easily applied during query analysis and showed that for the example query these optimizations reduced the predicate evaluation overhead to a reasonable level. Our conclusion is that a simplistic approach to implementing multiple granularities is quite expensive, but that via the combination of the straight forward optimizations that we presented the overhead of the additional expressive power of mixed granularities is quite small.

CHAPTER 8

NOW AND INDETERMINACY

Now is an English noun meaning "at the present time" [Sykes 1964]. *Now* is also a distinguished timestamp value in many temporal data model proposals. The precise semantics of this value is investigated elsewhere [Clifford et al. 1994B]. In this chapter, we examine the role of indeterminacy in providing a richer modeling capability for *now*. *Now* is a common timestamp value. Most real-world activities modeled by temporal databases must automatically maintain and update the current database state. For example, banks need to keep customer accounts current, stores need to keep track of which goods are in stock, and companies need to maintain data on who is currently employed. *Now* is important to accurately modeling all three enterprises.

8.1 *Now* in Valid-time

In the valid-time dimension, a common use of *now* is to indicate that a fact is valid until the current time [Ariav et al. 1984, Bassiouni & Llewellyn 1992, Elmasri et al. 1990, Gadia & Yeung 1988, Navathe & Ahmed 1989, Sarda 1990, Tansel 1990, Yau & Chat 1991]. In conventional databases, such facts are the only ones that are directly supported by the data model. For example, suppose that Jane began working as a welder for the factory on June 1. Figure 8.1 shows the relevant tuple from the factory's employment history (the **EMPLOYEE** relation). Jane started working as a welder on June 1, as indicated by the "from" attribute (for the examples in this chapter we assume a timestamp granularity of one day). The value *now*, appearing as the "to" time in Jane's employment tuple, represents a currently unknown future time when Jane will stop working for the factory. The result of a query that requests current welders includes Jane.

The informal semantics of this value is that Jane is an employee until we learn otherwise. As the current time inexorably advances, the value of *now* also changes to

Employee(Name, Position)

Name	Position	Valid time (from)	(to)
Jane	welder	June 1	*now*

Figure 8.1: Jane's employment tuple

reflect the new current time. Some authors have called this concept *until changed* instead of *now* [Wiederhold et al. 1993, Wiederhold et al. 1991], but the meaning is the same.

Suppose that instead of using the variable *now* as the "to" time in that tuple, we use a ground time, i.e., a particular date. We start by recording a "to" time of June 1. Then as time advances and Jane remains a welder, the "to" time on Jane's tuple must be updated each day to record when she worked. Hence, the "to" time would be updated to June 2, then to June 3, etc. While this representation is faithful to our knowledge at any point in time, it is unrealistic to assume that the "to" time will be continuously updated as time advances. It is also unclear who should do the updating, as the database has no indication of which timestamp values are stable and which are continuously changing. For these reasons, it is useful to use the variable *now*.

One problem with this use of *now* is the database appears to explicitly record that Jane will *not* be employed tomorrow. Assume, for the purpose of this discussion, that today is July 9. A query that asks who will be employed tomorrow (i.e., July 10) will not have Jane in the answer, since the "to" time of Jane's tuple is *now*, or in this case, July 9. Yet if nothing changes, in the database or in reality, and we execute the identical query (i.e., who was employed on July 10) on July 11, we will get a different answer, since Jane will be included. We call this problem the *pessimistic assumption*.

Some temporal data models avoid this problem by limiting valid time to the past, that is, to times before *now* [Gadia & Yeung 1988, Tansel 1990]. For many applications this limitation is much too restrictive. Other data models (e.g., [Ben-Zvi 1982, Snodgrass 1987, Snodgrass 1993, Thirumalai & Krishna 1988]) address this problem by using *forever* or ∞ as the "to" time, as shown in Figure 8.2. *Forever* is the largest representable timestamp value, that is, the one furthest in the future. This value admits that we do

Employee(Name, Position)

Name	Position	Valid time (from)	(to)
Jane	welder	June 1	forever

Figure 8.2: Jane's employment tuple with a large right boundary

not know when Jane will depart the university, and so assumes that she will be working forever. One limitation of this fix is that it is overly optimistic: forever is a long time! In SQL and in IBM's DB2, forever is about 8,000 years from the present [Date & White 1990, Melton & Simon 1993]; in our more liberal proposal, it is approximately 18 *billion* years from the present time [Dyreson & Snodgrass 1993B]. Hence, to assert that Jane will be employed forever is most assuredly incorrect (others have also noted that a "to" time of ∞, or *forever* has erroneous implications for the future [Navathe & Ahmed 1989]). A related limitation is that when Jane departs the university, *forever* must be revised with the date of her departure; but the revised date will be an entirely separate time, unrelated to, and inconsistent with, *forever*. The act of changing a time to an unrelated time is typically a corrective act applied to a previously incorrect time, rather than simply a refinement of previous information as new information is gathered.

An alternative way to view this problem is that there is a difference between the *actual* and *expected* times of a fact. On a day-to-day basis, we expect Jane to remain employed. A database that uses *forever* as the "to" time of her employment tuple (very optimistically) records her expected employment, while a database that uses *now* (very conservatively) records only her actual employment, the time she has worked to the current time. However, with indeterminacy, we can represent the actual and expected times using a single kind of time value.

Another problem shared by both of these approaches (a "to" time of either *now* or *forever*) is that they implicitly contain a very strong assumption, that we term the *punctuality assumption*, about the integrity of a valid-time database. The tuples shown in Figures 8.1 and 8.2 represent the fact that Jane is employed until the database indicates otherwise. That is, these tuples assume that the factory's database is a true, exact, up-to-

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	July 6
Jane	*possibly* employed as a welder	July 6	*now*

Figure 8.3: Meaning of Jane's tuple, if today is July 9 and the bound is 3 days

date model of the world. To remain consistent with the world, the instant Jane is fired, her unemployment must be manifested in the database. Otherwise, queries will return an incorrect result. If Jane was fired yesterday (i.e., July 8), but the database has yet to be updated, a query requesting the current employees executed on either relation will erroneously include Jane.

In many relations, the valid time is always earlier than the time of the transaction that stored the information, but within a well-specified, finite bound (we will shortly consider cases where valid time is later than transaction time) [Jensen & Snodgrass 1994, Jensen & Snodgrass 1992]. Typically, when an employee is hired or fired, it is not until several hours or days after the incident that the database record of that employee is updated to indicate the new employment status. For instance, perhaps Jane was fired on July 8, but it is not until July 11 that her tuple is actually updated to reflect her correct status. If the bound on the relationship between transaction execution time and valid time is known to be at most three days (all updates are made within three days), then a query that asks if Jane was employed four or more days ago can always determine Jane's correct employment status. Given such an assumption, one could interpret the meaning of Jane's tuple in Figure 8.1 as of today (July 9), as shown in Figure 8.3. This effectively replaces the unrealistic punctuality assumption with a weaker yet more reasonable *bounded assumption*. This interpretation may solve the problem posed by the punctuality assumption, but has the limitation of introducing a possibly employed status (how should that be handled?), and does not address the first problem, that of Jane still being employed tomorrow.

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	July 31 $\sim$ August 31

Figure 8.4: Jane has a fixed-term appointment, represented with valid-time indeterminacy

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	July 31 $\sim$ *forever*

Figure 8.5: Jane's employment will be for at least two months

8.2 *Now* Remodeled

By using indeterminate instants, we can more accurately record our knowledge of Jane's employment with the factory. Instead of using *now* as the "to" time in Jane's tuple, we can use an indeterminate instant. Which indeterminate instant to use depends on our knowledge of the situation. If Jane was hired as a limited-term employee, to work between two and three months, we could record this information as shown in Figure 8.4. Here two time bounds, July 31 and August 31, delimit the "to" indeterminate instant. If we knew only that the term would be *at least* two months, we would use the representation shown in Figure 8.5. If the factory has a mandatory retirement policy, we could decrease the indeterminacy considerably, as shown in Figure 8.6. If we removed the guarantee that Jane will work at least two months, we arrive at the representation shown in Figure 8.7.

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	July 31 $\sim$ Jan. 1, 2028

Figure 8.6: Assuming a mandatory retirement

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	June 1 $\sim$ Jan. 1, 2028

Figure 8.7: No guaranteed minimum term

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	July 6 $\sim$ Jan. 1, 2028

Figure 8.8: The situation as of July 9

Indeterminate instants address the first problem, the pessimistic update assumption, providing evidence that Jane might still be employed in the future. They also remove the problem of incompleteness in the non-timestamp attributes (e.g., *possibly* employed, as shown in Figure 8.3), and ensure that new knowledge acquired later, such as the information that Jane left the university on August 10, is not inconsistent with currently stored information, but rather is a refinement of that information.

Although indeterminate instants remove the pessimistic assumption, they require instantaneous updates. To illustrate the latter problem, assume that today is July 9, and that Jane departed today. Assume also that her departure has not yet been recorded in the database. The state of the Jane's tuple in the database should not be that of Figure 8.7, but rather that shown in Figure 8.8 (again, assuming at most a three day lag in recording a fact in the database). The state on July 10 is shown in Figure 8.9 (note how the indeterminacy in the "to" instant has decreased ever so slightly). On July 11, the indeterminacy disappears as we learn of Jane's departure. Each successive state is consistent with that preceding it, and each accurately records our current knowledge of Jane's status. But how can we automatically support this successive refinement of states?

To accurately represent our continuously changing knowledge about Jane's employment, we need to combine the best features of *now* and indeterminate instants, into a new

Employee(Name, Position)

Name	Position	Valid time (from)	(to)
Jane	welder	June 1	July 7 $\sim$ Jan. 1, 2028

Figure 8.9: The situation as of July 10

kind of instant, which we call a *now-relative indeterminate instant*. An example is shown in Figure 8.10. Here, the lower bound of the "to" timestamp is an expression involving the construct *now* and a span, in this case, three days, indicating the punctuality of updates.

A now-relative indeterminate instant captures both the actual and expected times associated with a fact. For instance, in the tuple given in Figure 8.10, Jane's actual employment history is delimited by the lower bound on the "to" time while her expected employment is delimited by the upper bound. The lower bound expresses, on a day-to-day basis, our changing knowledge of when Jane was employed while the upper bound expresses our expectation of how long she will remain employed. For example, if the reference time is July 9, the now-relative indeterminate instant shown in Figure 8.10 would be interpreted as the non-relative indeterminate instant "July 6 $\sim$ Jan. 1, 2028." Using a now-relative indeterminate instant ensures that continual updates are not required (in a sense, the updates are lazy and non-persistent), while capturing all of our knowledge of exactly when Jane was employed by the factory.

So what happens to the probability mass function as *now* approaches the upper bound? The probability mass function is "shrunk" by using *Shrink_s* as the period of indeterminacy gets shorter. For example, assume that we store the tuple shown in Figure 8.10 on June 1, and that we associate with that tuple the "normal" distribution. If 10% of the time between June 1 and Jan. 1, 2028 has elapsed, then the valid time of the tuple is $\textit{Shrink_s}(10, |\text{June 1} \sim \text{Jan. 1, 2028}|, \text{normal})$. Note that to shrink correctly, we must store the initial lower bound with the timestamp.

There is one wrinkle to this scheme, if the now-relative indeterminate instant is the "to" time of an indeterminate interval, then the lower bound on the instant must be between the upper bound on the "from" event and the upper bound on the "to" event. For instance,

Employee(Name, Position)

Name	Position	Valid time (from)	Valid time (to)
Jane	welder	June 1	(*now* − 3 days) ∼ Jan. 1, 2028

Figure 8.10: Using a now-relative indeterminate instant

the "to" lower bound of Jane's employment tuple is constrained to be sometime between June 1 and Jan. 1, 2028. If today is May 9, then the lower bound is June 1 and the tuple indicates that we *expect* Jane to be employed from June 1 to Jan. 1, 2028. If today is Jan. 1, 2050, then the upper bound is Jan. 1, 2028 and the tuple indicates that Jane was *actually* employed From June 1 to Jan. 1, 2028. In short, now-relative indeterminate instants capture the semantics of predictive updates.

Now-relative indeterminate instants are able to model the evolutionary character of temporal databases. A real-life prediction situation is either confirmed or proven false as time progresses. Similarly, as the reference time increases, values in the *possible* extensionalization of a tuple evolve into *definite* values or are removed from the database. Consider the tuple of Figure 8.10. If the reference time is May 9, then Jane will be *possibly* employed every day between June 1, 1994 and Jan. 1, 2028. However, for reference time April 1, 2028, the database records that Jane has *definitely* been employed every day between June 1, 1994 and Jan. 1, 2028. In summary, not only do now-relative indeterminate instants relax the strict punctuality assumption as now-relative determinate instants do, but they also support future queries and capture the semantics of predictive updates.

In the next section, we demonstrate that now-relative indeterminate instants can be stored in the same representation as determinate instants and non-relative indeterminate instants, with the result that they impose little space overhead.

8.3 Timestamp Representation

This chapter has proposed a new timestamp that must be represented: a now-relative indeterminate instant. In this section we extend the existing timestamp formats to represent

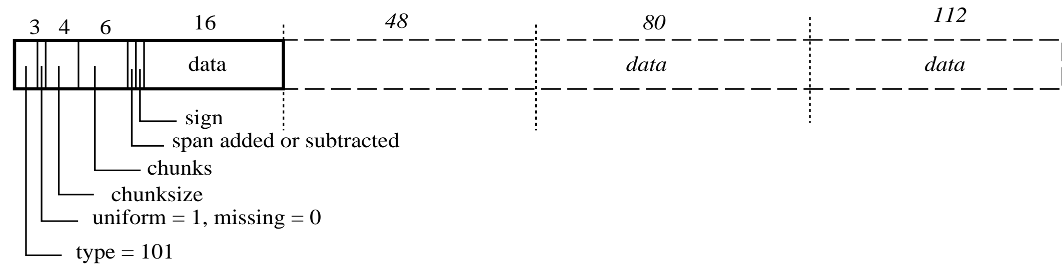


Figure 8.11: The now-relative indeterminate instant format

the new timestamp value, and show that now-relative indeterminate formats have a minor impact on storage costs. We extend only the instant formats; intervals are represented as a pair of bounding instants.

A now-relative indeterminate instant is quite complex. It is of the form "*now* $\pm$ *span* $\sim$ *upper bound*." A now-relative indeterminate instant format has an upper bound, a span, and the variable *now*. To properly scale nonuniform distributions, the initial lower bound must also be stored, i.e., the time at which the tuple was first stored. Surprisingly we can typically fit all of this information in the same space that a determinate instant requires, eight bytes in the common case. The key to the storage is that the span, representing the punctuality of updates, is the same for every tuple in a relation, hence it can be stored in the schema (in those rare cases where tuples are mixed with different update behaviors, the relation can be horizontally fragmented, the tuples in each fragment have the same update punctuality).

We employ a variation of a chunked indeterminate format that has essentially the same format, but a different interpretation. The new format is shown in Figure 8.11. A now-relative indeterminate instant has a different type field than an indeterminate instant since there is no other way to distinguish between the two formats. The upper bound is explicitly encoded in the data field, while the span is stored in the schema. The initial lower bound does not have to be stored for the standard distributions, since these distributions are not changed by shrinking. The sign bit positions the upper bound before or after the granularity anchor point, just as does the sign bit for determinate instants.

If the chunking yields an inadequate precision, or the user wishes to associate a distribution other than the missing or uniform distribution then more storage will be required. In this case, we use variants of other indeterminate instant formats. These variants are essentially the same as the indeterminate formats, but have a different type value. For brevity, we omit these variants.

As a point of comparison, the SQL-92 `TIMESTAMP` format, which has a range of only 10,000 years and a granularity of only one second, and which does not incorporate either indeterminacy or now-relativity, requires twenty positions (eighty bits). For most users, our eight byte format should suffice.

8.4 Summary

Using indeterminate and now-relative indeterminate instants can provide a far richer semantics for *now*. Now-relative indeterminate (and determinate) instants relax the strict punctuality assumption, replacing it with a more reasonable bounded assumption while indeterminate instants naturally support the uncertainty in future queries and allow for predictive updates. Both of these positive aspects are combined in now-relative indeterminate instants.

We showed that the new kinds of instants needed to combat the semantic difficulties associated with *now* have a compact representation, just two or three words in most cases, with virtually no space overhead compared with other timestamp representations.

CHAPTER 9

CONCLUSIONS

In this document, we extended a valid-time relational database to support valid-time indeterminacy, or "don't know when" information. Indeterminacy is a common kind of incomplete temporal information and has several sources, the most prevalent being granularity mismatches or conversions. We described a new temporal value, an indeterminate instant, that is capable of modeling imprecise event occurrence times, such as that an event occurred sometime between June 3, 1994 and June 10, 1994. An instant is determinate if it is known when (i.e., during which chronon) it is located. But if we only know that the instant is located sometime during a set or range of chronons, we call it an indeterminate instant. The indeterminacy refers to the location of the instant, not whether the instant exists. Indeterminate instants do not model the situation where it is unknown if an instant exists. An indeterminate instant is represented by a period of indeterminacy, describing the set of possible times for when the instant is located, and a probability mass function, giving the likelihood of each time in the period of indeterminacy. When a user creates an indeterminate instant, she may have no information about the underlying probability distribution. To handle this situation we introduced the capability of representing a distribution that is "missing," which represents a complete lack of knowledge about the probability mass function. We also described an indeterminate interval timestamp (to model intervals such as "sometime during the 70s through sometime during the 80s") and an indeterminate span timestamp (to model durations such as "three to five days"). The three new data-types were added to the TQuel and TSQL2 data models.

One important assumption we make throughout is that tuples are row-independent, with no information shared between indeterminate tuples. All the other database approaches that we are aware of that utilize probabilities to model various flavors of incompleteness make this assumption as well because computing dependent probabilities in the

inner loop of query processing is just too expensive. We also assume that the indeterminate instants can be modeled by contiguous sets of possible chronons. We do not support non-contiguous sets which could model indeterminate events such as "it happened yesterday morning or this morning," although an appropriate probability mass function (assigning zero probability to times other than this morning or yesterday morning, if this is within implementation bounds) could be used to model this particular situation. We exploit both of these assumptions to achieve efficiency in representation and in query processing. The contiguous set assumption allows us to represent a period of indeterminacy with a lower and an upper bound rather than having to represent each chronon in the set separately, while the independence assumption is necessary to rapid computation of the probabilistic ordering.

Query level support for indeterminacy rests on two controls on the retrieval process, range credibility and ordering plausibility. Range credibility changes the information available to query processing. It eliminates some possible but unlikely intervals from an indeterminate interval until the desired quality of information is reached. Ordering plausibility controls the construction of an answer to a query using the pool of credible information. We added both controls to the syntax and semantics of TQuel and TSQL2. As with the introduction of any new data type, permissible operations on indeterminate instants, intervals, and spans must be clearly specified. Operations on timestamps fall into four broad categories: comparison, arithmetic, input, and output [Dyreson & Snodgrass 1993B]. The class of operations impacted most by indeterminacy are comparisons, and, in particular, the temporal ordering operator, *Before*.

The semantics of *Before* without indeterminacy is based on a well-defined ordering of the valid-time instants in the underlying relations [Snodgrass 1987]. In the determinate semantics, *Before* is the "$\leq$" relation on event times. Every temporal expression consisting of *Before* operations and logical connectives refers to this ordering to determine if the expression is satisfied. A set of determinate instants has a single temporal ordering. Given a temporal expression, this ordering either satisfies the expression or fails to satisfy it.

A set of indeterminate instants, however, could have many possible temporal orderings. For example, one temporal ordering of the instants in the *Received* relation given on

page 18 is e_1, e_2, e_3, e_4. But the ordering e_1, e_3, e_2, e_4 is also possible. There are still other possible orderings. Some of these temporal orderings are *plausible* while others are *implausible*. We propose to permit the user to specify which orderings are plausible by setting an appropriate *ordering plausibility* value. We stipulate that a temporal expression is satisfied if there exists a plausible ordering that satisfies it.

The probability distribution information is related to the ordering plausibility value in the following fashion. Consider a set of instants, $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, in a temporal expression. For any pair of instants, α_i and α_j, compared by the expression, the probability that α_i is *Before* α_j in all possible extensions of the database cannot be less than the ordering plausibility. Essentially, we propose to treat each *Before* as an isolated test (in terms of probability calculations). The ordering plausibility value expresses the user's confidence in each test.

Operations on indeterminate values with a distribution that is missing are restricted. A tuple with a distribution that is missing only participates in queries which ask for the definite or possible answer (as these answers make no use of the underlying mass function).

We also augmented the create and modify statements in both TQuel and TSQL2 to specify which relations incorporate valid-time indeterminacy and to identify which timestamp format to use. The update statements (append, delete, and replace) can also be extended in an analogous manner. The TSQL2 syntax given here, including augmented schema specification, retrieval, update, and set statements, has been adopted in the consensus temporal query language TSQL2 [Dyreson & Snodgrass 1994F].

One of our goals in adding support for valid-time indeterminacy to TQuel and TSQL2 was to leave unchanged both the meaning and performance of all statements without indeterminacy. We call this property *query reducibility*. Support for indeterminacy is an extension of the determinate semantics and the implementation rather than a replacement.

While it is easy to posit an abstract model with a large set of weighted possible times, the actual time and space cost of such a model must be carefully considered. We note that other "probabilistic" database proposals [Barbará et al. 1989, Barbará et al. 1992,

Cavallo & Pittarelli 1987, Gelenbe & Hebrail 1986] do not address the issue of cost; perhaps because the researchers anticipated few weighted alternatives.

We showed how indeterminate instants with a uniform or unknown probability distribution can be represented in only 64 bits in most cases; for user-defined distributions the common representation is only 96 bits. The probability mass function is much too large to store with each timestamp, instead only a pointer to a mass function is stored. We optimized storage of the common mass function pointers.

We implemented the operations required by the extended semantics, and demonstrated that the implementation is efficient. The most difficult operation to implement efficiently was *Before*. *Before* is typically executed in the "inner loop" of query evaluation, performed possibly many times for each participating tuple. In this tight inner loop, *Before* must compute the probabilistic ordering between two instants, a potentially costly computation. But we successfully used approximation techniques to mitigate the cost of *Before*. We use a tree-like data structure to store an approximation to a user-given probability mass function. A "pivoting" algorithm then computes the probability that one instant is before another. Although the approximation of a mass function introduces some error into the "pivoting" computation, we showed that this error is bounded and that the size of the error is under the control of the database implementor (the implementor chooses the "precision" and "coarseness" of the approximation, and also sets the number of pivots to perform). There is a tradeoff between the size of the error and the speed of the pivoting algorithm. The speed of the pivoting algorithm can be improved by relaxing the error, however, the overall speedup on real-world data is slight. We experimented with error bounds of 1% and 10%. Except for rare situations, the run-time cost was essentially equivalent. We also compared the determinate *Before* with the indeterminate *Before*. Support for indeterminacy appears to essentially double the "in memory" cost of query evaluation (this analysis ignores the potentially far greater cost of disk I/O in query evaluation, which remains unchanged).

Support for indeterminacy also enables a richer modeling of granularity and of *now*. Granularity and indeterminacy are two sides of the same coin. A time at a given granularity is indeterminate with respect to all finer granularities. The practical impact of this

observation is that when a user chooses to convert an instant timestamp at a coarse granularity to one at a fine granularity with no loss of information, an indeterminate instant results. Indeterminacy is essential to implementing operations that move times around the granularity lattice.

Now is a timestamp value representing the current time. A now-relative indeterminate instant is a span, representing the the punctuality of updates, the special value *now*, indicating that the temporal information is current and changes as time advances, and an upper bound, limiting when the information is current. For example, suppose that an employee is currently employed, but will not work beyond the year 1995. We could represent this information using the now-relative indeterminate instant $|\ now \sim \texttt{1995}|$ as the "to" time of the employment interval. The lower bound of the "to" time (i.e., *now*) captures, on a day-to-day basis, our changing knowledge of the employee is employed while the upper bound (i.e., 1995) expresses our expectation of how long the employee will remain employed.

In sum, the approach that we espouse here has an intuitive semantics, is orthogonal to those proposed by others to handle value incompleteness and generalized events, refines previously proposed techniques to handle multiple granularities of time, increases the modeling capabilities of a temporal database, and has a practical implementation. The result is an expressive extension to TQuel and TSQL2. The extension is also "transparent" to the user who does not use the added query language and data model support for indeterminacy. The extended semantics and implementation reduce to the previous determinate semantics and implementation under the default credibility and plausibility.

This dissertation can be extended in various directions. Time and space are analogous, yet different, problem domains [Snodgrass 1992]. Perhaps there is a spatial analog to valid-time indeterminacy and the techniques developed here can be applied to "spatial indeterminacy." Can the representation and use of indeterminate instants can be naturally extended to indeterminate "points?"

"Best fit" queries are another avenue of exploration. A best fit query relaxes strict query constraints to accept tuples that come close to satisfying the query constraints, without actually satisfying query constraints. Often the answer set is ranked by some

"closeness measure." For example, suppose we are interested in finding out which airline flights are scheduled to depart around 9 AM. We could issue a best fit query for flights that are scheduled to depart at 9 AM. Perhaps no flights leave exactly at 9 AM; a best fit query would find flights that come closest to 9 AM first, followed by those that leave much earlier or much later. Indeterminacy can be used to support best fit queries. In the airline flight example, 9 AM could be automatically interpreted as the indeterminate instant | `6 AM` $\sim$ `12 AM` | with a normal distribution. A best fit query would then ask for flights that overlap this indeterminate instant, ranking the results by the highest ordering plausibility that satisfies the overlap.

Valid-time indeterminacy also engenders some novel query optimization problems since it is a new kind of information. For example, the *Before* operation costs slightly more for indeterminate instants than for determinate instants. If the database has only determinate instants, limiting the number of *Before* operations may have little impact on the evaluation of a temporal expression, but with indeterminate instants, a possible optimization is to limit the number of *Before* operations. As a simple example, consider the temporal expression, "δ `overlap(first(`γ, β`))` $\wedge$ α `precede` β." Evaluating the second conjunct requires a single call of *Before* while the first conjunct requires at least three *Before* operations. Evaluating the second conjunct and then the first (if needed) is a better evaluation strategy.

Another important issue is validation. This dissertation advocates a method for storing and querying temporally incomplete information in the hope that users will find this method practical and beneficial to their applications. A comprehensive study of the utility of the methods advocated here is addressing users' desires will validate or refute this hope. Such a study is beyond the scope of this dissertation.

APPENDIX A

TSQL2 BNF FOR INDETERMINACY

A.1 New or Modified Syntax for Indeterminacy Constructs

The organization of this section follows that of the SQL-92 document. The syntax is listed under corresponding section numbers in the SQL-92 document. All new or modified syntax rules are marked with a bullet ("•") on the left side of the production.

Where appropriate, we provide disambiguating rules to describe additional syntactic and semantic restrictions. We assume that the reader is familiar with the SQL-92 proposal, and that a copy of the proposal is available for reference.

A.1.1 Section 5.2 <token>

Six reserved words were added.

⟨*reserved word*⟩ ::=

- | `INDETERMINATE`
- | `CREDIBILITY`
- | `PLAUSIBILITY`
- | `GENERAL`
- | `NONSTANDARD`
- | `DISTRIBUTION`

A.1.2 Section 5.3 <literal>

No new syntax is introduced, but the allowable datetime, interval, and period literals is expanded to support indeterminate values.

Additional General Rules:

- Let A and B be valid <datetime value>s, representing the datetimes C and D. Let E be a string consistent with the *distribution_format* property, which can include references to the field *distribution_name*. If the value of the *indeterminate_datetime* property, with the *determinate_datetime1* field replaced with A, the *determinate_datetime2* field replaced with B, and the *distribution* field replaced with E, is identical to the <datetime value>, then the value represented by the <datetime value> is the indeterminate datetime with lower support C, upper support D, and distribution as names in E.

- Let A and B be valid <interval value>s, representing the intervals C and D. Let E be a string consistent with the *distribution_format* property, which can include references to the field *distribution_name*. If the value of the *indeterminate_interval* property, with the *determinate_interval1* field replaced with A, the *determinate_interval2* field replaced with B, and the *distribution* field replaced with E, is identical to the <interval value>, then the value represented by the <interval value> is the indeterminate datetime with lower support C, upper support D, and distribution as names in E.

A.1.3 Section 6.1 <data type>

The productions for the non-terminals ⟨*datetime type*⟩, ⟨*interval type*⟩ and ⟨*period type*⟩ are augmented with the following.

⟨*datetime type*⟩ ::=

- [⟨*indeterminate data type*⟩] `DATE` [⟨*time precision and scale*⟩]
- [⟨*indeterminate data type*⟩] `TIME` [⟨*time precision and scale*⟩]
  [`WITH TIME ZONE`]
- [⟨*indeterminate data type*⟩] `TIMESTAMP` [⟨*time precision and scale*⟩]
  [`WITH TIME ZONE`]

⟨*interval type*⟩ ::=

- [⟨*indeterminate data type*⟩] INTERVAL [⟨*time precision and scale*⟩]

⟨*period type*⟩ ::=

- [⟨*indeterminate data type*⟩] PERIOD [⟨*period precision*⟩]
  [WITH TIME ZONE]

The production, ⟨*indeterminate data type*⟩ is added.

⟨*indeterminate data type*⟩ ::=

- [NONSTANDARD] [GENERAL] INDETERMINATE

Additional syntax rules:

1. The default distribution is standard (not NONSTANDARD).
2. The default indeterminate datetime is compact (not GENERAL).
3. The default datetime is determinate (not INDETERMINATE).
4. The size of the timestamp format allocated depends on the kind of timestamp selected and the user-specified precision. Enough space must be allocated to the data fields to accommodate the precision of the timestamp (precision rules are described elsewhere). The default indeterminate timestamp format is the chunked with standard distributions format. By specifying GENERAL the user chooses to use one of the general, indeterminate timestamp formats. By specifying NONSTANDARD the user chooses to use one of the nonstandard timestamp formats.

A.1.4 Section 6.3 <table reference>

To the production for ⟨*table reference*⟩ is placed a ⟨*corr*⟩ non-terminal, which itself includes an optional credibility phrase.

⟨*table reference*⟩ ::=

- ⟨*table source*⟩ [[`AS`] ⟨*corr*⟩ { ⟨*corr*⟩ } ...]
- | ⟨*derived table*⟩ [`AS`] ⟨*corr*⟩ { ⟨*corr*⟩ } ...

  | ⟨*joined table*⟩

⟨*corr*⟩ ::=

- ⟨*correlation*⟩ [`WITH CREDIBILITY` ⟨*integer*⟩]

Additional general rules:

1. The credibility is a value between 0 and 100 (inclusive).
2. If the credibility phrase is missing, the default credibility is 100 or as specified by the user with a set statement.

A.1.5 Section 7.6 <where clause>

To the production for ⟨*where clause*⟩ is added the plausibility phrase.

⟨*where clause*⟩ ::=

- `WHERE` ⟨*search condition*⟩ [`WITH PLAUSIBILITY` ⟨*integer*⟩]

Additional general rules:

1. The plausibility is a value between 1 and 100 (inclusive). A value of 1 implies a non-zero plausibility less than 1.
2. If the plausibility phrase is missing, the default plausibility is 100 or as specified by the user with a set statement.

A.1.6 Section 11 <schema definition>

We add to the production for ⟨*schema element*⟩ to allow dynamic definition of distributions.

⟨*schema element*⟩ ::=

- | ⟨*create distribution statement*⟩

⟨*create distribution statement*⟩ ::=

- `CREATE` [{ `GLOBAL` | `LOCAL` } `TEMPORARY`] `DISTRIBUTION` ⟨*distribution name*⟩ `USING` ⟨*table name*⟩

⟨*alter distribution statement*⟩ ::=

- `ALTER DISTRIBUTION` ⟨*distribution name*⟩ `USING` ⟨*table name*⟩

⟨*drop distribution statement*⟩ ::=

- `DROP DISTRIBUTION` ⟨*distribution name*⟩

Additional general rules:

1. The distribution must conform to implementation-dependent distribution constraints, otherwise an exception is raised.
2. The ⟨*create distribution statement*⟩ establishes a new distribution name.
3. Altering a distribution effectively destroys the old distribution and replaces it with a new distribution having the indicated table descriptor.

A.1.7 Section 12.5 <SQL procedure statement>

To this class of statements are added the default plausibility and credibility statements.

⟨*SQL session statement*⟩ ::=

- | ⟨*set credibility statement*⟩
- | ⟨*set plausibility statement*⟩
- | ⟨*create distribution statement*⟩
- | ⟨*alter distribution statement*⟩
- | ⟨*drop distribution statement*⟩

⟨*set credibility statement*⟩ ::=

- SET CREDIBILITY { ⟨*integer*⟩ | AS DEFAULT }

⟨*set plausibility statement*⟩ ::=

- SET PLAUSIBILITY { ⟨*integer*⟩ | AS DEFAULT }

Additional syntax rules:

1. The most recent invocation of a ⟨*set credibility statement*⟩ or a ⟨*set plausibility statement*⟩ takes precedence.
2. If both the ⟨*set credibility statement*⟩ and the ⟨*set plausibility statement*⟩ are omitted, then the defaults, 100 and 100, respectively, are assumed.

A.2 New or Modified Syntax for Granularity

The organization of this section follows that of the SQL92 document. The syntax is listed under corresponding section numbers in the SQL92 document. All new or modified syntax rules are marked with a bullet ("•") on the left side of the production.

Where appropriate, we provide disambiguating rules to describe additional syntactic and semantic restrictions. We assume that the reader is familiar with the SQL92 proposal, and that a copy of the proposal is available for reference.

A.2.1 Section 5.3 <literal>

A literal is added.

⟨*time granularity*⟩ ::=

- ⟨*identifier*⟩

Additional syntax rules:

1. The available ⟨*time granularity*⟩s are implementation dependent, but must include `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

A.2.2 Section 6.8 ⟨*datetime value function*⟩

The following productions are added to the ⟨*datetime value function*⟩ non-terminal.

⟨*datetime value function*⟩ ::=

- | `SCALE` ⟨*left paren*⟩ ⟨*datetime value expression*⟩ `AS` ⟨*time granularity*⟩ ⟨*right paren*⟩

⟨*period value expression*⟩ ::=

- | `SCALE` ⟨*left paren*⟩ ⟨*period value expression*⟩ `AS` ⟨*time granularity*⟩ ⟨*right paren*⟩

Additional general rules:

1. Local invocation of a scale function overrides the global default.

A.2.3 Section 6.10 ⟨*cast specification*⟩

Casting to different granularities is allowed, by adding to the options of the ⟨*cast target*⟩.

⟨*cast target*⟩ ::=

⟨*domain name*⟩
| ⟨*data type*⟩
• | ⟨*time granularity*⟩

Additional syntax rules:

1. The table showing allowable data conversions is augmented to add the time granularity (G) cast target.

⟨*data type*⟩ of *SD* / ⟨*data type*⟩ of *TD*

	EN	AN	VC	FC	VB	FB	D	T	TS	YM	DT	P	TE	IS	G
EN	Y	Y	Y	Y	N	N	N	N	N	M	M	N	N	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N
C	Y	Y	M	M	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
B	N	N	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N
D	N	N	Y	Y	N	N	Y	N	Y	N	N	Y	Y	Y	Y
T	N	N	Y	Y	N	N	N	Y	Y	N	N	Y	Y	Y	Y
TS	N	N	Y	Y	N	N	Y	Y	Y	N	N	Y	Y	Y	Y
YM	M	Y	Y	Y	N	N	N	N	N	Y	N	N	N	N	Y
DT	M	Y	Y	Y	N	N	N	N	N	N	Y	N	N	N	Y
P	N	N	Y	Y	N	N	N	N	N	M	M	Y	Y	N	Y
TE	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y	Y
IS	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y	Y

2. If *SD* is D, T, TS, YM, DT, P, TE, or IS and *TD* is G then the conversion results in a value of the data type *SD* at the underlying granularity *TD*.

A.2.4 Section 6.14 <datetime value expression>

To the general rules for the non-terminal ⟨*datetime value expression*⟩ are added the following.

Additional general rules:

1. The following is added to Rule 3.

   The semantics of <datetime value expression>s involving <period term>s is calendar-dependent. If the underlying granularities of both are supplied by the tt SQL92 calendar, then the semantics are as follows. (Original Rule 3 goes here.)

2. Operands are coerced to the global scale or cast specified in the last `SET SCALE` or `SET CAST` command prior to the operation. If no such command was issued or the defaults are specified, then operands are scaled as needed to enforce ledt-operand semantics.

3. The range of intermediate results is the maximum allowed by the implementation.

A.2.5 Section 6.15 <interval value expression>

The following production is added to the ⟨*interval value expression*⟩ non-terminal.

⟨*interval value expression*⟩ ::=

- | `SCALE` ⟨*left paren*⟩ ⟨*interval value expression*⟩ `AS` ⟨*time granularity*⟩ ⟨*right paren*⟩

The general rules for the non-terminal ⟨*interval value expression*⟩ are augmented. Additional general rules:

1. Local invocation of a scale function overrides the global default.

2. The following is added to Rule 6.

   If <datetime value expression> is specified, the semantics is calendar-dependent. If the underlying granularities of both the <datetime value expression> and the <datatime term>, as well as the <period qualifier> are supplied by the `SQL92` calendar, then the semantics are as follows. (Original Rule 6 goes here.)

3. The granularity of the resulting type of the `SCALE` operation is ⟨*time granularity*⟩.

A.2.6 Section 10.1 <interval qualifier>

This is significantly generalized to allow implementation-defined granularities. The ⟨*non-second datetime field*⟩ non-terminal is removed, ⟨*timestamp qualifier*⟩ and ⟨*period qualifier*⟩ are added, and the following non-terminals are modified.

⟨*start field*⟩ ::=

- ⟨*time granularity*⟩ [⟨*left paren*⟩
  ⟨*interval leading field precision*⟩ ⟨*right paren*⟩]
- | ⟨*left paren*⟩ ⟨*interval string*⟩ ⟨*interval qualifier*⟩ ⟨*right paren*⟩

⟨*end field*⟩ ::=

- ⟨*time granularity*⟩ [⟨*left paren*⟩
  ⟨*interval fractional seconds precision*⟩ ⟨*right paren*⟩]

⟨*single datetime field*⟩ ::=

- ⟨*time granularity*⟩ [⟨*left paren*⟩ ⟨*interval leading fixed position*⟩
  [⟨*comma*⟩ ⟨*interval trailing field position*⟩] ⟨*right paren*⟩]

⟨*timestamp qualifier*⟩ ::=

- [⟨*start field*⟩ `TO`] ⟨*end field*⟩
- | ⟨*single datetime field*⟩

⟨*period qualifier*⟩ ::=

- [⟨*start field*⟩ `TO`] ⟨*end field*⟩
- | ⟨*single datetime field*⟩

The general rules are significantly generalized to remove many fairly arbitrary restrictions.

A.2.7 Section 11.10 <alter table statement>

The ⟨*alter table statement*⟩ is augmented with the following alternatives.

⟨*alter table statement*⟩ ::=

- | ⟨*scale valid definition*⟩
- | ⟨*cast valid definition*⟩

The following productions are added.

⟨*scale valid definition*⟩ ::=

- `SCALE VALID AS` ⟨*time precision and scale*⟩

⟨*cast valid definition*⟩ ::=

- `CAST VALID AS` ⟨*time precision and scale*⟩

Additional syntax rules:

1. Let T be the table identified in the containing ⟨*alter table statement*⟩.
2. T shall be a valid-time table.

Additional general rules:

1. The temporal element of each tuple of T is converted to the new precision and scale, using a cast or scale operation.

A.2.8 Section 12.5 <SQL procedure statement>

The production for the non-terminal <SQL session statement> is changed to include default session-level scale and cast specification commands.

⟨*SQL session statement*⟩ ::=

- | <set scale statement>
- | <set cast statement>

⟨*set scale statement*⟩ ::=

- SET SCALE { ⟨*time granularity*⟩ | AS DEFAULT }

⟨*set cast statement*⟩ ::=

- SET CAST { ⟨*time granularity*⟩ | AS DEFAULT }

The SET PROPERTIES statement is extended to optionally associate properties with a particular granularity.

⟨*set properties statement*⟩ ::=

SET PROPERTIES
[FOR CHARACTER SET [DEFAULT | NATIONAL | <character set>]]
- [FOR { ⟨*granularity name*⟩ | ⟨*calendar name*⟩ }]
WITH <property spec>

Additional syntax rules:

1. The most recent invocation of a ⟨*set scale statement*⟩ or a ⟨*set cast statement*⟩ takes precedence.

2. If both the ⟨*set scale statement*⟩ and the ⟨*set cast statement*⟩ are omitted (or specified *as default*, then coarser-granularity semantics is assumed.

3. Case:

   - If neither ⟨*granularity name*⟩ nor ⟨*calendar name*⟩ is specified, then the properties for all granularities are altered.

- If ⟨*granularity name*⟩ is specified, then only the properties for that granularity are altered.
- If ⟨*calendar name*⟩ is specified, then only the properties for the granularities defined by that calendar are altered.