Supporting Valid-time Indeterminacy

Curtis E. Dyreson and Richard T. Snodgrass

TR-7

A TIMECENTER Technical Report

Title	**Supporting Valid-time Indeterminacy**

Author(s)	Curtis E. Dyreson and Richard T. Snodgrass

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, Polytechnic University, New York, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors, The Rune alphabet (second phase) has 16 letters. They all have angular shapes and lack horizontal lines because the primary storage medium was wood. However, runes may also be found on jewelry, tools, and weapons. Runes were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

In *valid-time indeterminacy* it is known that an event stored in a database did in fact occur, but it is not known exactly *when* the event occurred. In this paper we extend the SQL data model and query language to support valid-time indeterminacy. We represent the occurrence time of an event with a set of possible instants, delimiting when the event might have occurred, and a probability distribution over that set. We also describe query language constructs to retrieve information in the presence of indeterminacy. These constructs enable users to specify their *credibility* in the underlying data and their *plausibility* in the relationships among that data. A denotational semantics for SQL's select statement with optional credibility and plausibility constructs is given. We show that this semantics is *reliable*, in that it never produces incorrect information, *maximal*, in that if it were extended to be more informative, the results may not be reliable, and *reduces* to the previous semantics when there is no indeterminacy. Although the extended data model and query language provide needed modeling capabilities, these extensions appear initially to carry a significant execution cost. A contribution of this paper is to demonstrate that our approach is useful and practical. An efficient representation of valid-time indeterminacy and efficient query processing algorithms are provided. The cost of support for indeterminacy is empirically measured, and is shown to be modest. Finally, we show that the approach is general, by applying it to the temporal query language constructs being proposed for SQL3.

Keywords: Incomplete information; indeterminacy; probabilistic information; SQL; temporal database; TSQL2; valid-time database.

1   Overview

Most databases record the history of an enterprise. Such databases associate with each 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 many 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 hour recorded on a system with timestamps in the granularity of a second happened sometime *during* that hour but during which second is unknown.

- *Dating techniques* — Many dating techniques, such as Carbon-14 dating [GC66], are inherently imprecise.

- *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 a student in the first grade was born. The student's date of birth could be recorded in the database as either unknown (she was born between the beginning and the end of time) or imprecise (she was born between five and seven years ago).

- *Clock measurements* — Every clock measurement has some imprecision [Pet91].

This paper proposes the *valid-time indeterminate* data model and its implementation. The model adds valid-time indeterminacy to SQL [MS93].

We identified several design goals to be met in extending SQL with indeterminacy. First, we wanted the syntactic extensions to be minimal, yet highly expressive. Timestamps should include a representation for

1

valid-time indeterminacy and users should be able to control, via query language constructs, the amount of indeterminacy present in derived information. Second, we wanted the extensions to be upward compatible with SQL. In the absence of valid-time indeterminacy the syntax and semantics should reduce to that of SQL. Third, the semantics should be simple and intuitive. In particular, information should not be 'invented' during queries. And finally, it was critical that valid-time indeterminacy have little impact on the performance of the DBMS, either in space to store indeterminate timestamps or in query evaluation time.

Unfortunately, these design goals conflict, as discussed further in Section 11. Earlier work required a three-valued [Sno82] or four-valued logic [Sch87]. Probabilistic approaches (e.g., [BGMP92]) are highly expressive, but have a query evaluation cost which is proportional to the number of probable alternatives. The number of such alternatives can easily number in the millions for a single indeterminate time, and thus probabilistic approaches are impractical if directly applied in a temporal context. More recent work models indeterminacy by using a constraint network. At best, using a restricted network, query evaluation complexity has been shown to be cubic [BCTP95] in the number of constraints. Our approach uses probabilistic weights, yet achieves a linear query evaluation complexity, with a nominal increase in storage cost.

The next section introduces an example requiring the storage of valid-time indeterminate information; this example will be used throughout the paper. We then examine the representation of valid-time indeterminacy. After that, we explore what it means to retrieve information from a database with valid-time indeterminacy. We outline syntactic and semantic extensions to SQL to support retrieval of valid-time indeterminate information, and argue their correctness. We then show how valid-time indeterminacy can be implemented. Although retrieving valid-time indeterminate information may appear to be expensive, we demonstrate that an efficient implementation exists. The final sections apply these same notions to a temporal query language, trace related work, summarize our approach, and discuss future work. Proofs for all theorems can be found in the Appendix.

2   Motivating Example

An example database is shown in Figure 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. This relation includes a period timestamp[1]; the other two relations include instant timestamps. For each relation we assume an underlying timestamp granularity of one day[2].

Valid-time indeterminacy naturally arises in both base relations and derived relations. The *During* attribute of the *In_Production* base relation is an indeterminate period. This is because the granularity of the *In_Production* relation is a month. A month is an indeterminate value that represents a set of possible days. Production on an airplane started on some day in the indicated month, but we cannot be sure which one. For example, production on the Centurion with serial number AB33 started sometime between (inclusive) March 1 and March 31. 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 educated guesswork. 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 *When*

---

[1]This paper will use the SQL3/Temporal [Mel96] term *period* to denote an anchored duration of time, e.g., the year 1995, in place of the term *interval* [JCE+94] which appears more commonly in the literature, as *interval* is used in SQL to denote an *unanchored* duration of time, e.g., one year.

[2]For expository purposes only, we adopt a non-SQL format for temporal constants, e.g., May 1 rather than 1996-05-01.

Sent_by_Boeing(Lot_Num,Part,When)   *Sent_by_Cessna(Lot_Num,Part,When)*

Lot_Num	Part	When
23	wing strut	May 6
24	engine	June 4

Lot_Num	Part	When
30	wing strut	May 26
31	wing strut	June 9

In_Production(Model, Serial_Num, During)

Model	Serial_Num	During
Centurion	AB33	[March 1 $\sim$ March 31 – June 1 $\sim$ June 30]
Cutlass	Z19	[June 1 $\sim$ June 30 – July 1 $\sim$ July 31]
Centurion	AB34	[June 1 $\sim$ June 30 – August 1 $\sim$ August 31]
Caravan	FA2K	[April 1 $\sim$ April 30 – May 1 $\sim$ May 31]

Received(Warehouse, Lot_Num, Part, When)

Warehouse	Lot_Num	Part	When	
Boeing	23	wing strut	May 10 $\sim$ May 29	e_1
Cessna	30	wing strut	May 30 $\sim$ June 18	e_2
Boeing	24	engine	June 8 $\sim$ June 27	e_3
Cessna	31	wing strut	June 13 $\sim$ July 2	e_4

Figure 1: An example database

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 Boeing warehouse arrived on one of the days in the set $\{\text{June 8, June 9, ..., June 27}\}$, but we have no reason to favor one day over another.

Queries can make use of indeterminate information in the database. 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). In SQL 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 [Lip79]. 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 (we formalize these bounds in Section 5.3). 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 number 23 from the Boeing 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 number 30 from the Cessna warehouse is in the possible answer to the query. It is possible that this shipment arrived prior to June 10 (and also possible that it did not). Similarly, lot number 24 from the Boeing warehouse possibly arrived prior to June 10. The first shipment from the Boeing warehouse is also in the possible answer because a definite answer is also a possible answer, but not vice-versa.

The Definite Answer	*Warehouse*	*Lot_Num*	*Part*	*When*
	Boeing	23	wing strut	May 10 ∼ May 29

The Probable Answer	*Warehouse*	*Lot_Num*	*Part*	*When*
	Boeing	23	wing strut	May 10 ∼ May 29
	Cessna	30	wing strut	May 30 ∼ June 18

The Possible Answer	*Warehouse*	*Lot_Num*	*Part*	*When*
	Boeing	23	wing strut	May 10 ∼ May 29
	Cessna	30	wing strut	May 30 ∼ June 18
	Boeing	24	engine	June 8 ∼ June 27

Figure 2: Answers to example queries

Between the possible and definite limits lie other answers. For instance, assume that it is equally likely for each day in {June 8, June 9, ..., June 27} that lot number 24 arrived. 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 number 24 arrived prior to June 10. However, it is *probable* that both lot number 30 (0.55 probability, 11 chances out of 20) and lot number 23 did arrive (1.00 probability). The definite, possible, and "probable" answer to the temporal selection are portrayed in Figure 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 conditions specified in the query. We provide separate controls on the indeterminacy for each stage.

Correlation credibility (potentially) changes the information available to query processing by replacing each indeterminate time value with a relevant determinate time value. A typical replacement is the *expected value* or probabilistic mean. For example, the expected start of production for the Centurion with serial number AB33 is March 15 (the expectation of the uniform distribution over a sequence of values is the half-way point in that sequence). In SQL with indeterminacy the user can express this preference by selecting an appropriate correlation credibility value. The chosen correlation credibility potentially modifies every time value in the associated relation, removing the indeterminacy.

Ordering plausibility controls the construction of an answer to the query. 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 "do they definitely 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 Cessna warehouse was received during production of the Centurion with serial number AB33, but one cannot be absolutely sure that it did.

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 SQL allows the user to control both kinds of indeterminacy. Correlation credibility replaces indeterminacy in the data while ordering plausibility governs the probability of relationships among the data.

4

3 Extending the Data Model with Indeterminacy

In this section we discuss how to represent indeterminate instants, periods, and intervals in the data model. In Section 7 we discuss how these representations are implemented.

3.1 Model of the Time-line

We briefly summarize the simple, standard model of time that we adopt for this paper. The model is presented in detail elsewhere [DSS95].

Time has a standard geometric metaphor. In this metaphor, time itself is a line segment (assuming a bounded universe); a point on the time-line is called an *instant*; the time between two instants is known as a *time period* (period for short); and a length, or unanchored segment, of the time-line is an *interval*. The time-line segment is partitioned into a finite number of smaller segments each of which is called a *chronon* [Ari86, CR87, JCE$^+$94]. A chronon is the smallest amount of time that can be represented in the implementation. The chronons are consecutively labeled with the integers in the sequence $0, \ldots, N$, where N is the number of values that a timestamp can represent. The domain of time values is the set of chronons and two special values, $-\infty$ and $+\infty$, which are used to represent the *beginning of time* instant and the *ending of time* instant, respectively.

3.2 Indeterminate Instants

An instant is a point on the time-line. An instant is *determinate* if it is known when (i.e., during which particular 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, rather than to the existence of the instant.

An indeterminate instant is described by a *lower support*, an *upper support*, and a *probability mass function (p.m.f.)* [DS93]. 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 $\sim$ May 29.

3.2.1 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

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

5

Like most other probabilistic approaches in databases [BGMP90, BGMP92, DS96, CP87, FR96, GH86, KS93b, KS93a, Zim92] no provisions are made for joint or dependent probabilities. An indeterminate instant, α, is denoted using the notation, $(\alpha_* \sim \alpha^*, P_\alpha)$.

3.2.2 Mass Function Sources

The probability mass function for an indeterminate instant is supplied by the user. 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 hour and recorded on a system with timestamps in the granularity of a second happened sometime *during* that hour, but during which particular second is unknown, and there is no there is no a priori reason to favor one second over another.

- *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 [Pet91].

Dey and Sarkar provide several additional means of determining the underlying mass function [DS96].

In some case, a user just may not know the underlying mass function because that information is unavailable or the mass function might exceed the implementation capacities of the system (Section 8 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 [BGMP90, BGMP92], 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 an 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.

3.3 Indeterminate Periods and Intervals

A *determinate period* is the time between two instants[3]. In our model of time, it is represented by a sequence of chronons, denoted by the starting and terminating chronons in the sequence.

---

[3]While the period data type is not in SQL-92[MS93], it is included in SQL3 [Mel96].

A period bounded by indeterminate instants (called the *starting* and *terminating instants*) is termed an *indeterminate period*. An indeterminate period could start during any member of the set of possible chronons of the starting instant. Likewise, the indeterminate period 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 period begins or ends.

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

We turn now from the data model to the query semantics.

4   Syntactic Extensions to SQL

In this section we summarize the syntactic extensions to SQL to support the storage and retrieval of valid-time indeterminate information from a database. Full coverage of the syntactic extensions can be found elsewhere [Sno95]. In the next section we provide a formal semantics for these constructs.

To support valid-time indeterminacy, three syntactic extensions to SQL are needed: (1) to indicate that a temporal attribute is indeterminate, (2) to specify the correlation credibility, and (3) to specify the ordering plausibility.

The first syntactic extension, to indicate that an attribute is indeterminate, involves the schema specification statements. In the create table statement a user may add either the modifier `INDETERMINATE` or `INDETERMINATE COMPACT` after an instant, period, or interval attribute specification to specify that the value may be indeterminate. The two modifiers toggle between alternative storage strategies for indeterminate timestamps, discussed in Section 7 (the compact version is a less expressive, smaller timestamp). We also add an optional "with" phrase to the end of the create table statement that allows the user to specify *standard* or *nonstandard* mass functions. These two categories of mass functions are also discussed in Section 7. The default is `WITH STANDARD DISTRIBUTION`. To the alter table statement we provide clauses that allow any of these aspects to be changed. Below are some examples of the extended create table and alter table statements.

```
CREATE TABLE Received(Warehouse  CHARACTER(30),
                      Lot_Num     INTEGER, Part CHARACTER(40),
                      When        INDETERMINATE DATE);
CREATE TABLE In_Production(Model  CHARACTER(30),
                      Serial_Num  CHARACTER(10),
                      During      INDETERMINATE PERIOD(DATE));
ALTER TABLE  Received ALTER COLUMN When TO NONSTANDARD DISTRIBUTION;
```

The second syntactic extension supports correlation credibility. The from clause in the select statement declares the relations over which the query is to be evaluated, and associates correlation name(s) with each relation. The specified correlation credibility is denoted via a `WITH CREDIBILITY` phrase, and is applied to each time value (instant, period, or interval) in the relation associated with the specified correlation name. The credibility is one of the following four values.

1. `INDETERMINATE` – Retain all the indeterminacy by leaving the time values unchanged.

2. `EXPECTED` – Replace each indeterminate time value with the expected value. Select this to compute the expected result for a query.

7

3. `MAX` – Replace each indeterminate time value with the lower support (except for an instant that starts a period, in which case use the upper support). For periods, this value eliminates all the indeterminacy. For instants, it uses the earliest possible instant, while for intervals, the shortest possible interval is chosen.

4. `MIN` – Replace each indeterminate time value with the upper support (except for an instant that starts a period, in which case use the lower support). For periods, this value converts all the indeterminate information to determinate information. For instants, it chooses the latest possible instant, while for intervals, the longest possible interval is used.

The credibility phrase is optional and has an initial default value of `INDETERMINATE`. This default value can be changed using a `SET CREDIBILITY` statement.

The third syntactic extension concerns the ordering plausibility. The ordering plausibility is the plausibility in the where predicate among the instants, periods, and intervals that participate in the predicate. The default ordering plausibility is specified using a `SET PLAUSIBILITY` statement. The default ordering plausibility can be over-ridden in a select statement by appending a `WITH PLAUSIBILITY` phrase to the end of the where clause. The ordering plausibility is an integer value between 1 and 100 (inclusive), and has an initial default value of 100. An ordering plausibility of 1 indicates that any possible answer is desired (i.e., the where predicate can be satisfied by any possible extension); an ordering plausibility of 100 requests the definite answer (i.e., the where clause must be satisfied by all possible extensions).

An example query illustrating the various constructs is shown in Figure 3. Intuitively, the query will determine, within the specified plausibility and credibility levels, which wing strut shipments were received during production of each Centurion. The from clause specifies that all information, regardless of its credibility, from the *In_Production* relation should be used (via the specified credibility of `INDETERMINATE`). The where clause selects pairs of Centurion and wing strut tuples that overlap with a plausibility of 60. Finally, the target list determines when the shipment of possibly defective parts was received. When this query is applied to the database shown in Figure 1, the relation shown in Figure 4 is computed. We will discuss in detail in Section 5.8 how this query is evaluated to obtain the indicated result.

5 Semantic Extensions to SQL

In this section we extend the semantics of SQL to support indeterminacy. The presentation focuses on SQL's select statement. We first provide a brief review of the semantics of the select statement. We then extend the semantics to support indeterminacy. The evaluation of a select in the extended semantics has a possible and a definite interpretation, as well as other interpretations that lie between those bounds. We show that the possible interpretation is both *reliable*, in that it does not invent information, and *maximal*, insofar as it cannot be strengthened to produce more results. However, the indeterminate semantics does not demonstrate that indeterminacy can be efficiently implemented. Consequently we introduce an operational semantics. The operational semantics provides all the necessary support for indeterminacy with three changes to the SQL semantics. First it redefines the temporal ordering relation, *Before*. Second, it introduces a 4-sorted domain for the evaluation of where clause predicates. And third, it adds a *Replace* operator to effect correlation credibility. Each of the changes incorporates the determinate semantics (as the default). Hence, the semantics of existing queries is left unchanged. We show that the operational semantics correctly implements the indeterminate semantics.

5.1 Review of SQL Semantics

In this section we present a simplified semantics for the select statement. Our goal in this presentation is to highlight those aspects of the statement that will be impacted by valid-time indeterminacy, a theme which

```
SET PLAUSIBILITY 60
SELECT r.Warehouse, r.Lot_Num, p.Serial_Num, r.When
   FROM Received AS r WITH CREDIBILITY INDETERMINATE,
        In_Production AS p WITH CREDIBILITY INDETERMINATE
   WHERE p.Model = "Centurion" AND r.Part = "wing strut"
        AND r.When OVERLAPS p.During
```

Figure 3: An example query

Warehouse	Lot_Num	Serial_Num	When		
Boeing	23	AB33	May 10	$\sim$	May 29
Cessna	30	AB33	May 30	$\sim$	June 18
Cessna	31	AB34	June 13	$\sim$	July 2

Figure 4: Result of the example query

we will develop in subsequent sections.

We will use the notation, $[\![\, x\,]\!]_{SQL}$, to denote the meaning of the syntactic SQL construct x. The top-level of a denotational semantics for the select statement, applied to a database d, is given below.

$$[\![\, \text{SELECT} \ \langle\text{target list}\rangle \ \text{FROM} \ \langle\text{from list}\rangle \ \text{WHERE} \ \langle\text{predicate}\rangle \,]\!]_{SQL}(d) =$$
$$[\![\ \langle\text{target list}\rangle \,]\!]_{SQL}([\![\, \text{WHERE} \ \langle\text{predicate}\rangle \,]\!]_{SQL}([\![\ \langle\text{from list}\rangle \,]\!]_{SQL}(d)))$$

The select statement first applies the from clause to the database. This clause computes the Cartesian product of the relations specified in the $\langle\text{from list}\rangle$. The meaning of the from clause is

$$[\![\ \langle\text{from list}\rangle \,]\!]_{SQL}(d) = [\![\ \langle\text{relation}_1\rangle, \ldots, \langle\text{relation}_n\rangle \,]\!]_{SQL}(d) =$$
$$[\![\ \langle\text{relation}_1\rangle \,]\!]_{SQL}(d) \times \ldots \times [\![\ \langle\text{relation}_n\rangle \,]\!]_{SQL}(d)$$

where

$$[\![\ \langle\text{relation}_i\rangle \,]\!]_{SQL}(d) = r_i, \ r_i \ \in \ d, \text{and } r_i \text{ is named } \langle\text{relation}_i\rangle.$$

The result computed by the from clause, an intermediate relation r, is then used as an argument for the where clause. This clause selects those tuples that satisfy the $\langle\text{predicate}\rangle$ in the where clause.

$$[\![\, \text{WHERE} \ \langle\text{predicate}\rangle \,]\!]_{SQL}(r) = \{t \mid t \in r \land [\![\ \langle\text{predicate}\rangle \,]\!]_{SQL}(t)\}$$

Note that the correlation names appearing in $\langle\text{predicate}\rangle$ will need to be mapped to the attributes in the tuple t. This is generally done by passing the symbol table as a second argument to $[\![\ ]\!]_{SQL}$. For simplicity, we omit that argument. We will restrict the presentation to predicates that are logical formulæ constructed from comparison operations and boolean connectives, and will rely upon the readers background knowledge of SQL to supply the meaning of each SQL $\langle\text{predicate}\rangle$, e.g., the standard Boolean logic applies:

$$[\![\ \langle\text{predicate}\rangle \ \text{AND} \ \langle\text{predicate}\rangle \,]\!]_{SQL}(t) = [\![\ \langle\text{predicate}\rangle \,]\!]_{SQL}(t) \land [\![\ \langle\text{predicate}\rangle \,]\!]_{SQL}(t).$$

In the final step, the output of the where is projected onto the desired domains specified by the $\langle\text{target list}\rangle$.

$$[\![\ \langle\text{target list}\rangle \,]\!]_{SQL}(r) = \pi_{[\![\ \langle\text{target list}\rangle \,]\!]_{SQL}}(r)$$

Here we also ignore the mapping of correlation names to attributes of r.

5.2 Supporting the Syntactic Extensions

Since the current SQL semantics does not support indeterminacy the extended syntax can only be supported in a new, extended semantics, which we denote as $[\![\ ]\!]_{ind}$. The extensions presented in Section 4 include two additional controls on the indeterminate information: the correlation credibility and the ordering plausibility. These values will appear as additional parameters, δ and γ, respectively, to $[\![\ ]\!]_{ind}$. Both values can be specified using SET DEFAULT statements. The meanings of the SET DEFAULT statements are given below. Assume S is any SQL statement.

$$[\![\,\texttt{SET DEFAULT CREDIBILITY } \delta';\, S\,]\!]_{ind}(\delta, \gamma, d) = [\![\,S\,]\!]_{ind}(\delta', \gamma, d)$$
$$[\![\,\texttt{SET DEFAULT PLAUSIBILITY } \gamma';\, S\,]\!]_{ind}(\delta, \gamma, d) = [\![\,S\,]\!]_{ind}(\delta, \gamma', d)$$

The default values can be over-ridden within the select statement itself. Here we show how to over-ride the plausibility default; in Section 5.7 we show how to over-ride the credibility default.

$$[\![\,\texttt{SELECT } \langle\textit{target list}\rangle \texttt{ FROM } \langle\textit{from list}\rangle \texttt{ WHERE } \langle\textit{predicate}\rangle \texttt{ WITH PLAUSIBILITY } \gamma'\,]\!]_{ind}(\delta, \gamma, d) =$$
$$[\![\,\texttt{SELECT } \langle\textit{target list}\rangle \texttt{ FROM } \langle\textit{from list}\rangle \texttt{ WHERE } \langle\textit{predicate}\rangle\,]\!]_{ind}(\delta, \gamma', d)$$

The initial default credibility is INDETERMINATE and initial default plausibility is 100.

5.3 An Overview of the Indeterminate Semantics

In this section we give an overview of the indeterminate semantics and discuss several properties that a semantics involving incomplete information should possess. In later sections we show that our semantics does indeed have these essential properties.

The indeterminate semantics for the select statement is outlined below. This semantics has the same structure as the SQL semantics.

$$[\![\,\texttt{SELECT } \langle\textit{target list}\rangle \texttt{ FROM } \langle\textit{from list}\rangle \texttt{ WHERE } \langle\textit{predicate}\rangle\,]\!]_{ind}(\delta, \gamma, d) =$$
$$[\![\, \langle\textit{target list}\rangle\,]\!]_{ind}(\gamma, [\![\,\texttt{WHERE } \langle\textit{predicate}\rangle\,]\!]_{ind}(\gamma, [\![\, \langle\textit{from list}\rangle\,]\!]_{ind}(\delta, d)))$$

The SQL and indeterminate semantics differ in two ways. First, the indeterminate semantics has additional parameter(s), but note that δ is utilized only in the from list, and γ is utilized only in the where predicate and target list. Second, the select statement has a different meaning in the indeterminate semantics. Let's consider what the meaning should be, intuitively.

A select statement applied to a database containing only complete information and evaluated under SQL semantics has a single interpretation. In contrast, a retrieval from a database containing incomplete information has at least two interpretations. One interpretation is that the query selects information that *possibly* matches the retrieval constraints. The second interpretation is that the query selects information that *definitely* matches the retrieval constraints. Which interpretation is adopted is specified by the user via syntactic constructs in the query. It is important however, to guarantee that a query avoids producing *impossible* results. That is, a query should be constrained to compute a subset of the possible interpretation, and a superset of the definite interpretation. We formalize this by introducing the concept of a completion.

An indeterminate instant can be thought of as a set of possible instants, one of which is the "actual" instant, but which one is unknown. Each of the possible instants represents a different, complete description of reality. Each possibility is termed a *completion* of the instant[4]. The following definition captures this intuition.

---

[4]Gadia introduced the completion of a temporal tuple and of a temporal relation [GNP92]. Our definition extends this notion to indeterminate instants, periods and intervals and to conventional tuples, relations and databases containing such values.

Definition 1 (Completion of an indeterminate instant) *Let $\alpha = (\alpha_* \sim \alpha^*, P_\alpha)$. A completion of α is α_i, where α_i is a determinate instant such that $\alpha_* \leq \alpha_i \leq \alpha^*$. The set of all completions for an instant α is denoted $\mathbf{C}(\alpha)$.* ∎

Completions of periods and intervals also exist. A completion of an indeterminate period is one in which its delimiting (indeterminate) instants are both replaced by their completions. A completion of an indeterminate interval is one of the possible durations. The concept can also be generalized to apply to tuples, relations, and databases. A completion of x, be it a tuple, relation, or database, is x_c, where x_c is the same as x but with each indeterminate instant, period and interval replaced by a completion of that value. The set of all completions for an entity x is denoted $\mathbf{C}(x)$.

In the indeterminate semantics, the possible interpretation of the select statement is attained by using a plausibility of 1, while the definite interpretation is given by adopting a plausibility of 100. We focus on the where clause to show the difference in these two interpretations. The definite interpretation of the where clause is given below.

$$[\![\texttt{WHERE } \langle predicate \rangle]\!]_{ind}(100, r) = \{t \mid t \in r \ \wedge \ \forall t' \in \mathbf{C}(t)(\ [\![\ \langle predicate \rangle]\!]_{SQL}(t') \ )\}$$

The definite interpretation selects only those tuples that are selected by the SQL semantics (note the use of $[\![\ \langle predicate \rangle]\!]_{SQL}$) in *every* completion of the tuple.

The possible interpretation differs only slightly. It selects only those tuples that are selected by the SQL semantics in *some* completion of the tuple.

$$[\![\texttt{WHERE } \langle predicate \rangle]\!]_{ind}(1, r) = \{t \mid t \in r \ \wedge \ \exists t' \in \mathbf{C}(t)(\ [\![\ \langle predicate \rangle]\!]_{SQL}(t') \ )\}$$

This semantics is *reliable*, in the sense that it never produces incorrect information[5]. For a semantics to be reliable, the result of a where clause on a completion of a relation r in the SQL semantics should be consistent with the result in the indeterminate semantics on the relation r.

Theorem 1 $[\![\texttt{WHERE } \langle \text{predicate} \rangle]\!]_{ind}(1, r)$ *is reliable, that is, for any where clause, W,*

$$\forall r' \in \mathbf{C}(r) \, [[\![W]\!]_{SQL}(r') \in \mathbf{C}([\![W]\!]_{ind}(1, r))] \ .$$

Proof: Proofs are given in the Appendix. ∎

The semantics is also *maximal*, in that if the semantics were extended to be more informative, i.e., allow more completions, then the result may no longer be reliable. From the previous theorem, we know that $[\![W]\!]_{ind}(1, r)$ contains all the needed completions. We need to determine that it contains no extraneous completions.

Theorem 2 $[\![\texttt{WHERE } \langle \text{predicate} \rangle]\!]_{ind}(1, r)$ *is maximal, i.e., for any where clause, W,*

$$\forall c \in \mathbf{C}([\![W]\!]_{ind}(1, r))[\exists r' \in \mathbf{C}(r) \, (c = [\![W]\!]_{SQL}(r'))] \ .$$

Note that these two theorems in concert demonstrate that, for all where clauses W and indeterminate relations r,

$$\mathbf{C}([\![W]\!]_{ind}(1, r)) = \bigcup_{r' \in \mathbf{C}(r)} [\![W]\!]_{SQL}(r').$$

Observe that if the database has only complete information, there is only one completion, and effectively just a single interpretation of a query since the possible and definite interpretations are equivalent. So

---

[5]Here we adopt the notions of reliability, and later, maximality, used by Gadia [GNP92].

the credibility and plausibility have no effect whatsoever on existing databases, which contain only complete information, and the semantics of extant SQL queries and databases is unchanged by extensions to support indeterminacy. This important property is termed *temporal upward compatibility* [BBJS96]. In the context of indeterminacy, it is also equivalent to stating that the indeterminate data model is a *generalization* [GNP92] of the determinate (SQL) data model.

The possible and definite interpretations are just two of the many interpretations available in the indeterminate semantics. Other interpretations result from choosing other credibility and plausibility values. These interpretations are related. Increasing the plausibility setting in a select statement yields a "more definite" interpretation. It is essential however, to ensure that these other interpretations will not generate spurious results. We guarantee this by demonstrating (in Section 5.9) that evaluating a select statement in the indeterminate semantics is monotonic in the plausibility. Each result at a higher plausibility setting is a subset of the result at a lower plausibility.

5.4 Supporting Ordering Plausibility

Both the possible and definite interpretations of a query in the indeterminate semantics are stated in terms of the SQL semantics. Unfortunately, a straightforward implementation of these interpretations would be highly impractical, since it would require computing the predicate over every possible completion of a tuple in the inner loop of query processing. There could be many completions for each tuple, depending on the duration of the period of indeterminacy. As an example, there are seven billion completions of a single indeterminate period that has delimiting instants with a period of indeterminacy of one day, assuming a chronon size of one second.

Consequently, we introduce an operational semantics, $[\![\ ]\!]_{op}$, that implements the indeterminate semantics. We later show that the operational semantics is efficient, satisfies the goals outlined above, and is fully consistent with the indeterminate semantics, $[\![\ ]\!]_{ind}$.

5.4.1 Probabilistic Ordering

The ordering plausibility (primarily) impacts the meaning of the where clause. The semantics of the where predicate without indeterminacy is based on a well-defined ordering of the instants in the underlying relations. Every temporal predicate (e.g., OVERLAPS) and temporal constructor (e.g., PERIOD($\langle$*instant value*$\rangle$, $\langle$*instant value*$\rangle$)) refers to the ordering given by *Before* to determine the truth value of the predicate or the instant or period returned by the constructor. For example consider the following OVERLAPS predicate.

$$[\![\ \langle instant_1 \rangle \text{ OVERLAPS PERIOD}(\langle instant_2 \rangle, \langle instant_3 \rangle) \ ]\!]_{SQL} =$$
$$Before([\![\ \langle instant_2 \rangle\ ]\!]_{SQL}, [\![\ \langle instant_1 \rangle\ ]\!]_{SQL}) \wedge Before([\![\ \langle instant_1 \rangle\ ]\!]_{SQL}, [\![\ \langle instant_3 \rangle\ ]\!]_{SQL})$$

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, due to the many completions of those instants. 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 between pairs of instants that satisfies each predicate in the expression. This semantics reduces to that of the determinate case when there is only one ordering.

In the SQL semantics, *Before* is the "$\leq$" relation on the representation of instants (i.e., on chronons).

$$Before(\alpha, \beta) = \alpha \leq \beta$$

$$\beta$$

α		e_1	e_2	e_3	e_4
	e_1	1.00	1.00	1.00	1.00
	e_2	0	1.00	.86	.96
	e_3	0	.16	1.00	.73
	e_4	0	.05	.27	1.00

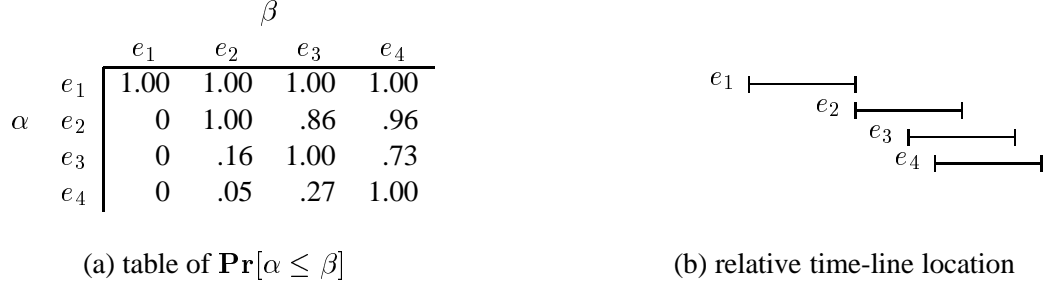(a) table of $\mathbf{Pr}[\alpha \leq \beta]$ 　　　　　　　(b) relative time-line location

Figure 5: $\mathbf{Pr}[\alpha \leq \beta]$ for the indeterminate instants in *Received*

In the operational indeterminate semantics, the temporal ordering is instead based on the probability that one instant is before another. We define that probability first, and then give the new ordering operation.

Definition 2 (Probabilistic ordering) *For any two indeterminate instants, α and β, the probability that α is before β is $\mathbf{Pr}[\alpha \leq \beta] = \sum_{i \leq j} \mathbf{Pr}[\alpha = i] \times \mathbf{Pr}[\beta = j] \quad i, j \in \{0, \ldots, N\}$.* ■

Figure 5(a) shows the value of $\mathbf{Pr}[\alpha \leq \beta]$ (to two decimal places) for each pair of instants in the relation *Received*, e.g., $\mathbf{Pr}[e_2 \leq e_3] = .86$. Those instants are placed on a time-line in Figure 5(b).

The probabilistic ordering 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 $\mathbf{Pr}[(\alpha \leq \beta \leq \eta)]$.

To handle indeterminate instants in a flexible manner, we define a new operator, *Before$_I$*, that includes an additional 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.

Definition 3 (Indeterminate *Before*) *Let α and β be a pair of indeterminate instants.*

$$Before_I(\alpha, \beta, \gamma) = \begin{cases} \{\} & \text{if } \neg(\mathbf{Pr}[\alpha \leq \beta] \times 100) \geq \gamma) \wedge \neg(\mathbf{Pr}[\beta < \alpha] \times 100) \geq \gamma) \\ \{True\} & \text{if } (\mathbf{Pr}[\alpha \leq \beta] \times 100) \geq \gamma) \wedge \neg(\mathbf{Pr}[\beta < \alpha] \times 100) \geq \gamma) \\ \{False\} & \text{if } \neg(\mathbf{Pr}[\alpha \leq \beta] \times 100) \geq \gamma) \wedge (\mathbf{Pr}[\beta < \alpha] \times 100) \geq \gamma) \\ \{True, False\} & \text{if } (\mathbf{Pr}[\alpha \leq \beta] \times 100) \geq \gamma) \wedge (\mathbf{Pr}[\beta < \alpha] \times 100) \geq \gamma) \end{cases}$$ ■

In the operational semantics, *Before$_I$* is defined on the power set of the standard boolean domain (a four-sorted domain, cf. [Bel77]). There are four possible outcomes for *Before$_I$* on a pair of instants. In the first case, the relationship does not hold, nor does its negation, and *Before$_I$* evaluates to the empty set. In the second, the relationship holds but its negation does not and *Before$_I$* evaluates to $\{True\}$. In the third case, the relationship does not hold but its negation does. *Before$_I$* evaluates to $\{False\}$. The fourth case covers the other possibility, that both the relationship and its negation hold simultaneously. *Before$_I$* evaluates to $\{True, False\}$.

Note that in the operational semantics *Before$_I$* treats ordering probabilities that are between 0 and 0.01 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 0.01 whenever its value is between 0 and 0.01.

Definition 4 (Refined definition of probabilistic ordering) *Let α and β be a pair of indeterminate instants.*

$$\mathbf{Pr}'[\alpha \leq \beta] = \begin{cases} \mathbf{Pr}[\alpha \leq \beta] & \text{if } \mathbf{Pr}[\alpha \leq \beta] > 0.01 \\ 0 & \text{if } \mathbf{Pr}[\alpha \leq \beta] = 0 \\ 0.01 & \text{otherwise} \end{cases}$$ ■

13

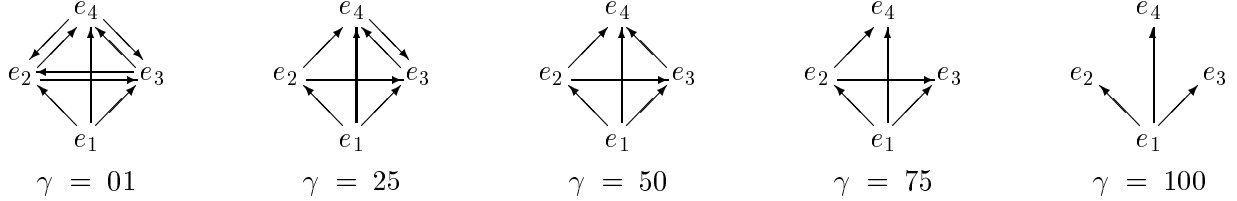$\gamma\ =\ 01$ $\gamma\ =\ 25$ $\gamma\ =\ 50$ $\gamma\ =\ 75$ $\gamma\ =\ 100$

Figure 6: Ordering the events in *Received* depends on γ

This function replaces **Pr** in the definition of *Before$_I$*. With this substitution, to evaluate every possible ordering, however improbable, an ordering plausibility of 1 suffices.

The *missing* mass function is treated specially. If one (or more) of the instants being ordered has the missing mass function, then the mass is assumed to be distributed in such a way that there is a small, but non-zero probability, ϵ, of ordering the two instants. For instance, if we introduce an instant e_5 that has an overlapping period of indeterminacy with e_2, but a missing mass function, then $\mathbf{Pr}'[e_2 \leq e_5] = \epsilon$ and $\mathbf{Pr}'[e_5 \leq e_2] = \epsilon$. Consequently, in the semantics, an instant with a missing mass function behaves exactly like a *null value*, in that we stipulate that the participation of such an instant in a *Before$_I$* operation makes the *Before$_I$* evaluate to the empty set (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$_I$* will be return $\{True, False\}$ for an ordering plausibility of 1.

The ordering relation, *Before$_I$*, 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 6. 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.

5.4.2 Predicates, Logical Formulæ and Constructors

We are now in a position to supply the operational semantics for the where clause predicate. We will assume, without loss of generality, that this predicate is a logical formulæ composed of *Before* operations on pairs of instants; a complete set of temporal predicates can be constructed with such formulæ [All83].

$$[\![\ \langle predicate_1 \rangle \ \texttt{AND} \ \langle predicate_2 \rangle \ ]\!]_{op}(\gamma, r) = [\![\ \langle predicate_1 \rangle \ ]\!]_{op}(\gamma, r) \bigcap [\![\ \langle predicate_2 \rangle \ ]\!]_{op}(\gamma, r)$$
$$[\![\ \langle predicate_1 \rangle \ \texttt{OR} \ \langle predicate_2 \rangle \ ]\!]_{op}(\gamma, r) = [\![\ \langle predicate_1 \rangle \ ]\!]_{op}(\gamma, r) \bigcup [\![\ \langle predicate_2 \rangle \ ]\!]_{op}(\gamma, r)$$
$$[\![\ \texttt{NOT} \ \langle predicate \rangle \ ]\!]_{op}(\gamma, r) = \{x \mid \neg x \in [\![\ \langle predicate \rangle \ ]\!]_{op}(\gamma, r)\}$$

Below, we illustrate the semantics with several examples using the instants in Figure 5.

$$[\![e_2 \leq e_3]\!]_{op}(r, 100) = Before_I(e_2, e_3, 100) = \{\} \quad [\![\texttt{NOT} \ (e_2 \leq e_3)]\!]_{op}(r, 100) = \{\}$$
$$[\![e_2 \leq e_3]\!]_{op}(r, 50) = \{True\} \quad\quad [\![\texttt{NOT} \ (e_2 \leq e_3)]\!]_{op}(r, 50) = \{False\}$$
$$[\![e_2 \leq e_3]\!]_{op}(r, 1) = \{True, False\} \quad\quad [\![\texttt{NOT} \ e_2 \leq e_3]\!]_{op}(r, 1) = \{True, False\}$$
$$[\![e_2 \leq e_3 \ \texttt{AND} \ e_1 \leq e_4]\!]_{op}(r, 100) = \{\} \quad [\![\texttt{NOT} \ (e_2 \leq e_3 \ \texttt{AND} \ e_1 \leq e_4)]\!]_{op}(r, 100) = \{\}$$
$$[\![e_2 \leq e_3 \ \texttt{AND} \ e_1 \leq e_4]\!]_{op}(r, 50) = \{True\} \quad [\![\texttt{NOT} \ (e_2 \leq e_3 \ \texttt{AND} \ e_1 \leq e_4)]\!]_{op}(r, 50) = \{False\}$$
$$[\![e_2 \leq e_3 \ \texttt{AND} \ e_1 \leq e_4]\!]_{op}(r, 1) = \{True\} \quad [\![\texttt{NOT} \ (e_2 \leq e_3 \ \texttt{AND} \ e_1 \leq e_4)]\!]_{op}(r, 1) = \{False\}$$
$$[\![e_2 \leq e_3 \ \texttt{OR} \ e_1 \leq e_4]\!]_{op}(r, 100) = \{True\} \quad [\![\texttt{NOT} \ (e_2 \leq e_3 \ \texttt{OR} \ e_1 \leq e_4)]\!]_{op}(r, 100) = \{False\}$$
$$[\![e_2 \leq e_3 \ \texttt{OR} \ e_1 \leq e_4]\!]_{op}(r, 50) = \{True\} \quad [\![\texttt{NOT} \ (e_2 \leq e_3 \ \texttt{OR} \ e_1 \leq e_4)]\!]_{op}(r, 50) = \{False\}$$

14

$$[\![\, e_2 \leq e_3 \ \texttt{OR} \ e_1 \leq e_4 \,]\!]_{op}(r, 1) = \{True, False\} \quad [\![\, \texttt{NOT} \ (e_2 \leq e_3 \ \texttt{OR} \ e_1 \leq e_4) \,]\!]_{op}(r, 1) = \{True, False\}$$

A key difference between the operational semantics and the SQL semantics is that the operational semantics uses *Before$_I$* rather than *Before*. Here is the operational semantics for the OVERLAPS example given previously.

$$[\![\, \langle instant_1 \rangle \ \texttt{OVERLAPS PERIOD(} \ \langle instant_2 \rangle \texttt{,} \ \langle instant_3 \rangle \texttt{)} \,]\!]_{op}$$
$$= Before_I([\![\, \langle instant_2 \rangle \,]\!]_{SQL}, \ [\![\, \langle instant_1 \rangle \,]\!]_{SQL}, \gamma) \bigcap Before_I([\![\, \langle instant_1 \rangle \,]\!]_{SQL}, \ [\![\, \langle instant_3 \rangle \,]\!]_{SQL}, \gamma)$$

Elsewhere we show how constructors can be extended in the presence of indeterminacy [DS93].

We reiterate that in the probabilistic ordering all instants are assumed to be independent. So in the expression "$Before_I(e_2, e_3, \gamma)$ AND $Before_I(e_3, e_4, \gamma)$" the two *Before$_I$* operations operations are separately evaluated, returning a set of boolean values that is subsequently intersected. While the strategy of separate evaluation of conjuncts is consistent with the determinate semantics, it is important to realize that it is not equivalent to computing $(\mathbf{Pr}[(e_2 \leq e_3 \leq e_4)] \times 100) \geq \gamma$.

5.4.3 Indeterminate Semantics of the Where Clause

The meaning of the where clause in the indeterminate semantics can now be given in terms of the operational semantics.

$$[\![\, \texttt{WHERE} \ \langle predicate \rangle \,]\!]_{ind}(\gamma, r) = \{t \mid t \in r \wedge True \in [\![\, \langle predicate \rangle \,]\!]_{op}(\gamma, t)\}$$

Note that the computation of the predicate over the completions of each underlying tuple has been replaced with set operations over the result of the $\mathbf{Pr}'$ function. Admittedly this function appears to also be expensive to compute, as it is $O(m^2)$ per tuple, where m is the number of chronons in the periods of indeterminacy of the instants to which $\mathbf{Pr}'$ is applied (m can be very large). However, we show in Section 8 that the $\mathbf{Pr}'$ computation can be approximated in constant time. With this result, the operational semantics for the evaluation of a temporal expression consisting of temporal predicates and boolean connectives has the same complexity as that of the determinate semantics: $O(n)$, where n is the number of predicates in the query, independent of both the number of chronons and the number of completions of indeterminate values of the tuple.

5.5 *Set_Before*

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

$$Set\_Before(\gamma, \ \alpha, \ \beta) \iff \forall x \in \alpha \ \forall y \in \beta \ Before_I(\gamma, \ x, \ y)$$

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. *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 of instants is earlier, or it could be that both are earlier. In the indeterminate semantics, *First*

operates on sets of instants.

$$First(\gamma,\,\alpha,\,\beta) \;\;=\;\; \begin{cases} \alpha \bigcup \beta & \text{if } (Set\_Before\,(\gamma,\,\alpha,\,\beta) \wedge Set\_Before\,(\gamma,\,\beta,\,\alpha)) \\ \alpha & \text{if } Set\_Before\,(\gamma,\,\alpha,\,\beta) \\ \beta & \text{if } Set\_Before\,(\gamma,\,\beta,\,\alpha) \\ \eta - \delta & \text{otherwise, where} \\ & \qquad \eta = \alpha \bigcup \beta \text{ and} \\ & \qquad \delta = \{x|\; x \in \eta \wedge \neg\exists y \in \eta\; (Before_I(\gamma,\,y,\,x))\} \end{cases}$$

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. 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 $Before_I$ 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*.

$$\begin{array}{ll} First(50,\,\{e_2\},\,\{e_3\}) = \{e_2\} & (\alpha \text{ is first}) \\ First(100,\,\{e_2\},\,\{e_1\}) = \{e_1\} & (\beta \text{ is first}) \\ First(1,\,\{e_2\},\,\{e_3\}) = \{e_2\} \text{ and } \{e_3\} & (\text{both } \alpha \text{ and } \beta \text{ are first}) \\ First(100,\,\{e_2\},\,\{e_3\}) = \{e_2,\,e_3\} & (\alpha \text{ and } \beta \text{ are incomparable}) \end{array}$$

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 6), for example:

$$\begin{array}{l} First(100,\,\{e_1\},\,First(100,\,\{e_2\},\{e_3\})) = \{e_1\} \\ First(100,\,\{e_2\},\,First(100,\,\{e_3\},\{e_1\})) = \{e_1\}. \\ First(100,\,First(100,\,\{e_2\},\{e_1\}),\,First(1,\,\{e_3\},\,\{e_4\})) = \{e_1\} \end{array}$$

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:

$$\begin{array}{l} First(1,\,\{e_1\},\,First(1,\,\{e_2\},\{e_3\})) = \{e_1\} \\ First(1,\,\{e_2\},\,First(1,\,\{e_3\},\{e_1\})) = \{e_1\} \\ First(1,\,First(1,\,\{e_2\},\{e_1\}),\,First(1,\,\{e_3\},\,\{e_4\})) = \{e_1\}. \end{array}$$

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

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

16

Contrast this with the determinate semantics for the overlap constructor:

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

One final aspect should be mentioned. Within the where clause, or in the target list, as discussed below, periods can appear as attribute values or as the result of a constructor. In both cases, for a period p, if $Before_I(p_{from}, p_{to}, \gamma)$, then the period value should be replaced with the value NULL. Additionally, any indeterminate predicate applied to a NULL argument should yield the empty set.

5.6 The Target List

The indeterminate semantics of the target list is quite simple.

$$[\!\![\ \langle target\ list \rangle\ ]\!\!]_{ind}(\gamma, r) = \pi_{[\!\![\ \langle target\ list \rangle\ ]\!\!]_{ind}(\gamma)}(r)$$

The only wrinkle is that any constructors in the target list, e.g., `PERIOD()`, will utilize the ordering plausibility, γ.

5.7 Supporting Correlation Credibility

Correlation credibility extends the from clause with an optional credibility phrase. In general correlation credibility is used to replace indeterminate time values with determinate time values. The replacement strategy to use depends upon the credibility value and whether the time value replaced is an instant, period, or interval. Below we define a *Replace* function that effects the replacement for every time value in a tuple.

Definition 5 (*Replace*) *Let tuple* $t = (\bar{X}, \alpha_1, \ldots, \alpha_n)$ *where* $\bar{X}$ *are non-temporal values and* $\alpha_1, \ldots, \alpha_n$ *are time values. Then*

$$Replace(\delta, t) = (\bar{X}, \mathbf{R}(\alpha_1), \ldots, \mathbf{R}(\alpha_n)).$$

R *is the replacement strategy. Table 1 lists the replacement strategies for each combination of credibility value and kind of time value, assuming that the time value is* $(\alpha_* \sim \alpha^*, P_\alpha)$ *and* $\mathbf{E}[\alpha]$ *is the expected value.*

As an example, consider the tuple

$t = $ (Centurion, AB33, [March 1 $\sim$ March 31 – June 1 $\sim$ June 30]).

For this period timestamped tuple the four credibility values yield the following results.

Replace(`INDETERMINATE`, t) = t
Replace(`EXPECTED`, t) = (Centurion, AB33, [March 15 – June 15])
Replace(`MAX`, t) = (Centurion, AB33, [March 31 – June 1])
Replace(`MIN`, t) = (Centurion, AB33, [March 1 – June 30])

Using the *Replace* function, we are in a position to define the meaning of the from clause in the indeterminate semantics, including the specification of how the default credibility can be over-ridden for a particular correlation variable.

$[\!\![\ \langle from\ list \rangle\ ]\!\!]_{ind}(\delta, d) = $
$\quad [\!\![\ \langle from_1 \rangle, \ \ldots, \ \langle from_n \rangle\ ]\!\!]_{ind}(\delta, d) = $
$\quad\quad [\!\![\ \langle from_1 \rangle\ ]\!\!]_{ind}(\delta, d) \times \ldots \times [\!\![\ \langle from_n \rangle\ ]\!\!]_{ind}(\delta, d)$

where each of the $\langle from_i \rangle$'s can be either of two constructs.

	instant	period start	period end	interval
INDETERMINATE	α	α	α	α
EXPECTED	$\mathbf{E}[\alpha]$	$\mathbf{E}[\alpha]$	$\mathbf{E}[\alpha]$	$\mathbf{E}[\alpha]$
MIN	α_*	α_*	α^*	α_*
MAX	α^*	α^*	α_*	α^*

Table 1: Replacement strategies

$$[\![\ \langle \textit{relation}\rangle\ ]\!]_{ind}(\delta, d) = \{t' \mid t \in \langle \textit{relation}\rangle \wedge t' = Replace(\delta, t)\}$$
$$[\![\ \langle \textit{relation}\rangle\ \texttt{WITH CREDIBILITY}\ \delta'\ ]\!]_{ind}(\delta, d) = \{t' \mid t \in \langle \textit{relation}\rangle \wedge t' = Replace(\delta', t)\}$$

When the $\langle \textit{from list}\rangle$ is given a credibility of $\delta = \texttt{INDETERMINATE}$, it retains all of the indeterminacy present in the temporal values of d, not altering them at all. Hence,

$$[\![\ \langle \textit{from list}\rangle\ ]\!]_{ind}(\texttt{INDETERMINATE}, d) = [\![\ \langle \textit{from list}\rangle\ ]\!]_{SQL}(d).$$

5.8 Result of the Example Query

At this point, the indeterminate semantics of the select statement has been specified. As an example, we provide the tuple calculus semantics for the query given in Figure 3 on page 9.

$$
\begin{aligned}
[\![\ Q\ ]\!]_{ind}(d) \ =\ & \{(r.Warehouse,\ r.Lot\_Num,\ p.SerialNum,\ r.When)\mid \\
& r \in Replace(\texttt{INDETERMINATE},\ Received) \\
& \wedge p \in Replace(\texttt{INDETERMINATE},\ In\_Production) \\
& \wedge p.Model = {}'\text{Centurion}' \wedge r.Part = {}'\text{wingstrut}' \\
& \wedge\ True \in \{\textit{Before}_I(p.During_{from},\ r.When,\ 60) \\
& \qquad\qquad \cap \textit{Before}_I(r.When,\ p.During_{to},\ 60)\}\}
\end{aligned}
$$

Here the Cartesian product and projection operators have been expressed in the tuple calculus.

If this query is applied to the database given in Figure 1 on page 3, it will result in three tuples, shown in Figure 4. First, the time values in the underlying relations are unchanged because the query uses a correlation credibility of INDETERMINATE. 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 $(\text{May }30 \sim \text{June }18,\ \text{uniform})$ is before $(\text{June }1 \sim \text{June }30,\ \text{uniform})$ 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 Cessna warehouse.

5.9 Correctness

The intuitive semantics ($[\![\ ]\!]_{ind}$), motivated in Section 5.3, applies the predicate to each completion and thus is impractical, whereas the operational semantics ($[\![\ ]\!]_{op}$) can be efficiently implemented. The correctness of the operational semantics hinges on two requirements. The first is that the two semantics agree where the intuitive semantics is defined, that is, for the possible and definite interpretations. The possible interpretation has an ordering plausibility of $\gamma = 1$; the definite interpretation has $\gamma = 100$. The interpretations concern only the ordering plausibility; they apply at any correlation credibility.

Theorem 3 $[\![\;]\!]_{ind}(\gamma, r)$ *and* $[\![\;]\!]_{op}(\gamma, r)$ *are equivalent for* $\gamma = 1$ *and* $\gamma = 100$.

The second requirement is that the operational semantics be monotonic: as the plausibility increases, the result must move from the possible interpretation towards the definite interpretation. This ensures that the semantics at intermediate plausibilities is consistent with that of the possible interpretation.

Theorem 4 $[\![\,S\,]\!]_{ind}(\delta, \gamma, r)$ *is monotonic in* γ.

6   Implementation Overview

Changes to the semantics to support valid-time indeterminacy induce changes in the implementation. These changes are isolated to the representation of instants, intervals, and periods, and to the new or modified temporal operators: *Before$_I$*, *Set_Before*, and *Replace*. 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. Some of the modified operators, e.g., *Before$_I$*, 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.

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, periods, and intervals. We then show how the new operators can be implemented efficiently, focusing on the probabilistic ordering used in *Before$_I$*.

7   Indeterminate Timestamp Formats

Valid-time indeterminate instants, periods, and intervals model new kinds of temporal information. To represent indeterminate temporal values, new temporal data types, or *timestamps* [JCE$^+$94], are needed. In this section we briefly describe the indeterminate timestamps. We present only the indeterminate instant timestamps; the interval and period timestamps are natural extensions of the instant timestamps [DS95b].

The instant format described here builds upon the determinate instant format; a full description of this format is given elsewhere [DS95b]. A determinate instant timestamp combines a type tag (to indicate the kind of instant, e.g., determinate or indeterminate) with a signed integer representing a distance (in chronons or, more precisely, in *granules*[6]) from the time-line *origin* or *anchor-point*. The type tag occupies three bits and the signed integer either 29, 61, or 93 bits depending upon the maximum range and granularity of the timestamp. The 64-bit 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. It is important to point out that the range and granularity of an instant timestamp are stored in the schema rather than in the timestamp.

Indeterminate instants cannot use the determinate timestamp because an indeterminate instant is more than a single time: it is two times and a probability mass function. To represent indeterminate instants, we add four new formats. These four formats are a combination of *compact* or *general* with *standard* or *nonstandard* distributions. The combinations are explained in detail below.

---

[6]A granule is a coarse-grained grouping of chronons, e.g., chronons can be grouped into days, years, or months. The timestamp stores the distance in terms of granules so that large ranges of times, e.g., 10 million years, can be stored compactly by counting in coarse granules, e.g., in millenia.

FORMAT	Starting Time	Terminating Time	Probability Distribution
determinate	explicit	implicit	implicit
compact, standard	explicit	implicit	implicit
compact, nonstandard	explicit	implicit	explicit
general, standard	explicit	explicit	implicit
general, nonstandard	explicit	explicit	explicit

Table 2: Encodings in the indeterminate formats

Each indeterminate timestamp format has the three basic parts needed to describe an indeterminate instant: a lower support, an upper support, and a probability mass function. These three parts are encoded in the timestamp either implicitly or explicitly. Table 2 indicates for each format whether the representation is explicit or implicit. For example, the determinate format has an explicit lower support, but an implicit upper support (identical to the lower support) and an implicit probability mass function (the mass function is *missing*).

The compact indeterminate formats implicitly encode the upper support. The implicit encoding consists of a *chunk-size* and a number of *chunks*. 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 chunking scheme was developed to meet the expectation that regular periods of indeterminacy, e.g., N hours, N days, or N years, will be the norm. If the format is not compact it is said to be *general* and the upper support is stored explicitly as a count of granules from the anchor point.

If the chunk-size and number of chunks can be stored in two small fields, then chunking is a very efficient method of encoding a terminating time. In our formats, the chunk-size is a four-bit field and the number of chunks a seven-bit field, or eleven bits in toto. Since the chunk-size is four bits, only a limited number of chunk-sizes are available. One of the duties of the database implementor is to specify chunk-size tables, one for each supported granularity. Common chunk-sizes are seconds, hours, days, and weeks. The space efficiency of the chunking 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.

The timestamp representation of a probability mass function is the name of a mass function (a 16-bit identifier). Only the name of the mass function, e.g., uniform, normal, etc., is stored with the timestamp; the actual mass function is stored separately as described further in Section 8.1.2. Instants with a uniform distribution or a distribution that is missing, are termed the *standard* distributions. Since these distributions will likely be common, we optimized representation their representation as a single bit rather than a 16-bit identifier (to toggle between uniform and missing).

The user specifies the kind of timestamp, compact or general, and the kind of distribution, standard or nonstandard, to use when defining or altering a temporal attribute as discussed in Section 4. The design of the indeterminate timestamp formats optimizes representation of the common mass functions (the standard mass functions cost only a single bit). 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 four-bit nibbles) is 56 bits. Our indeterminate compact timestamp with the same range and granularity as the SQL-92 datetime format requires only 64 bits. The smallest indeterminate timestamp is just 32 bits (a compact, standard format with a range of 2^{18} granules and a chunked period of indeterminacy). The largest is 208 bits (a general, nonstandard format with an upper and lower support within 2^{93} granules of the granularity anchor point).

8   Implementation of Operators

In this section we discuss the implementation of *Before$_I$*, *Set_Before*, and *Replace*.

8.1   Implementation of *Before$_I$*

We observed in Section 5 that the semantics of temporal constructors and predicates such as OVERLAPS and PERIOD are ultimately based on *Before$_I$*. If the instants being compared by *Before$_I$* are determinate then *Before$_I$* is the "$\leq$" relation on the domain of time values (the integers extended with special values representing the beginning and end of time). Indeterminate instants complicate the implementation of *Before$_I$*. In the indeterminate semantics it may be necessary to compute the probability that one instant is before another. This is a potentially costly computation. We show below how this computation can be made efficient.

8.1.1   The Common Interface

The interface to the *Before$_I$* routine is given in Figure 7. 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$_I$* is trivially true since an instant is always equal to itself. *Before$_I$* 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 periods of indeterminacy will be common. If no special case applies then the ordering probability, $\mathbf{Pr}'[\alpha \leq \beta]$, must be calculated.

8.1.2   Probability Mass Function Representation

In this section we describe a data structure to store a probability mass 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 pre-computed and its values cached, say, in an array. The latter strategy is best for a probability mass function. *Before$_I$* 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.

To attain reasonable storage costs, the probability mass function is *approximated*. We assume that the domain of the mass function is normalized to [0,1] and that 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. The mass function is sampled at C evenly-spaced points. C is called the *coarseness* of the approximation. The sample points are $\{\frac{1}{2C}, \frac{3}{2C}, \ldots, \frac{2C-1}{2C}\}$. To approximate the mass function, for each sample point, the rods to the left and right of the point are recorded. Typically there is also a rod that covers the point, but this rod is not counted (this is the error in the approximation). The coarseness is usually 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 8.

The rod and point method of approximating a probability mass function has some limitations. The coarseness and precision restrict the variety of functions that can be approximately represented. 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 functions are possible. Also, mass functions that have a mass of more than $\frac{1}{P}$ spread over less

function *Before$_I$*(**in** α, β : instant; **in** γ : integer) : set of boolean;
    begin
        if α *is* β **then return** {True}
        else if $\alpha^* \leq \beta_*$ **then return** {True}
        else if $\beta^* < \alpha_*$ **then return** {False}
        else return *PROB_α_LEQ_β*(α, β, γ)
    end;

Figure 7: Interface to *Before$_I$*

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 instance, 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 8 there are no rods to the left and two rods to the right of the first point. The example shows that the number of rods left and right of a sample point will often sum to $P - 1$ since the rod covering the node is uncounted. The tree for the approximated uniform mass function is shown in Figure 9(a).

In the tree shown in Figure 9, 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 recording where each rod ends. Observe that if both children have the same count, then no rod ends within the subtree rooted at the parent (and all nodes in the subtree will have the same count). 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 pruned tree for the example distribution is shown in Figure 9(b). The specially marked nodes have dashed line borders.

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. Each interior node is two $\log_2(C)$-bit pointers while each leaf node is two $\log_2(P)$-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 distribution tree efficiently stores the approximated probability mass function, but the approximation impacts the computation of $\mathbf{Pr}'[\alpha \leq \beta]$, changing the problem to a rod counting problem.

8.1.3 An Overview of Computing $\mathbf{Pr}'[\alpha \leq \beta]$

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 two rows, which we shall call the α-row and the β-row, are parallel to each other as shown in Figure 10. As shown in the figure, the length of each rod can vary. The rod counting problem is to count the pairs of rods, one rod from
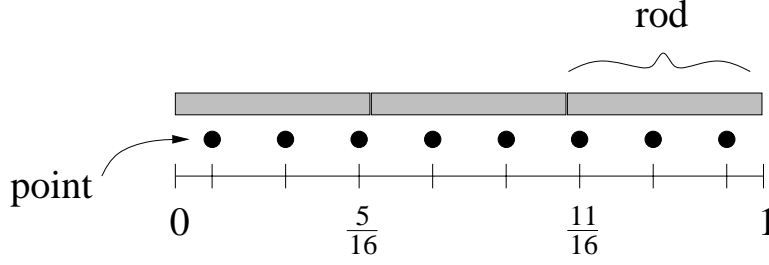
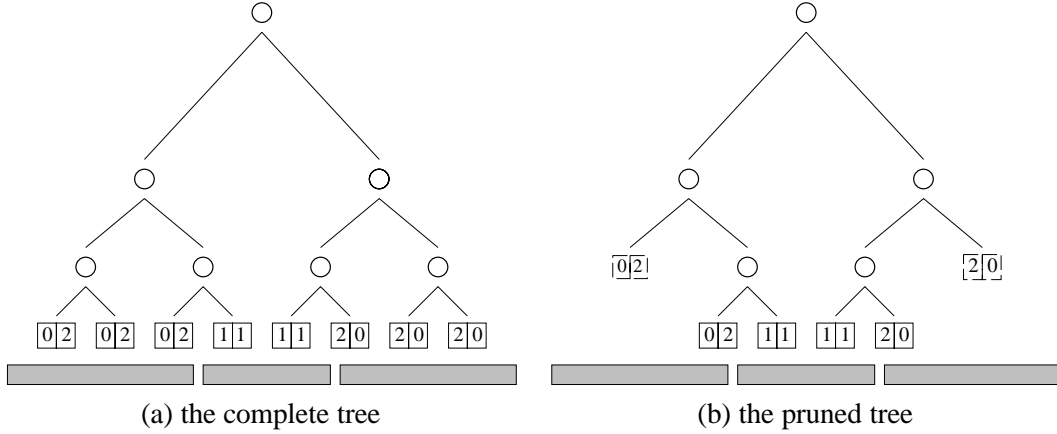Figure 8: The approximated uniform mass function with $P = 3$ and $C = 8$

(a) the complete tree (b) the pruned tree

Figure 9: The tree for the approximated uniform mass function with $P = 3$ and $C = 8$

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 $\mathbf{Pr}'[\alpha \leq \beta]$.

The rod counting problem is complicated by the fact that several rods may be located within a single chronon, so some rods in β could be strictly before those in α yet may still contribute to $\mathbf{Pr}'[\alpha \leq \beta]$. For the purpose of computing $\mathbf{Pr}'[\alpha \leq \beta]$ 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 also 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 $\mathbf{Pr}'[\alpha \leq \beta]$. 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, 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 probability ($\leq \frac{1}{P^2}$) of this case and thus undercounts $\mathbf{Pr}'[\alpha \leq \beta]$. Below 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 10. 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.

23

The second step is to identify where the left-end of the pivot belongs in the ordering of β's rods. The left-end of the pivot divides β's row of rods into three parts: those before the left-end of the pivot, β_{before}, those after the left-end, β_{after}, and, perhaps, a rod that overlaps the left-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 $\mathbf{Pr}'[\alpha \leq \beta]$. If the running sum (scaled by 100) exceeds the plausibility, γ, then the algorithm terminates since the plausibility has been met and $Before_I$ 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 $\mathbf{Pr}[\beta < \alpha]$. If the running sum (scaled by 100) exceeds $100 - \gamma$, then the algorithm terminates since $Before_I$ 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 α_{before} and those in $\beta_{before} \bigcup \beta_{overlap}$, as well as the relationships between the the rods in $\alpha_{after} \bigcup pivot$ and those in β_{after}. Each of these subproblems is solved recursively in the next "round" of the algorithm.

8.1.4 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 $\mathbf{Pr}'[\alpha \leq \beta]$ or $\mathbf{Pr}[\beta < \alpha]$ 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 5 *The kth pivot will count $P^2/2^{\lfloor \log_2(k) \rfloor + 1}$ pairs.*

For example, 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.

As we pointed out earlier, the approximation by rods and points leads to an undercounting of $\mathbf{Pr}'[\alpha \leq \beta]$. However, this undercounting is small.

Theorem 6 *The undercount is less than $\frac{2}{P}$.*

8.1.5 Implementation Details for Computing $\mathbf{Pr}'[\alpha \leq \beta]$

The code for the "pivoting" algorithm is shown in Figure 11. The counting stops when the count of pairs exceeds the needed number of true pairs or false pairs (it simultaneously solves for both $\mathbf{Pr}'[\alpha \leq \beta]$ and $\mathbf{Pr}[\beta < \alpha]$), or when all the possible pivots have been tried (an undercount has occurred and we assume that $\mathbf{Pr}'[\alpha \leq \beta]$ 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.
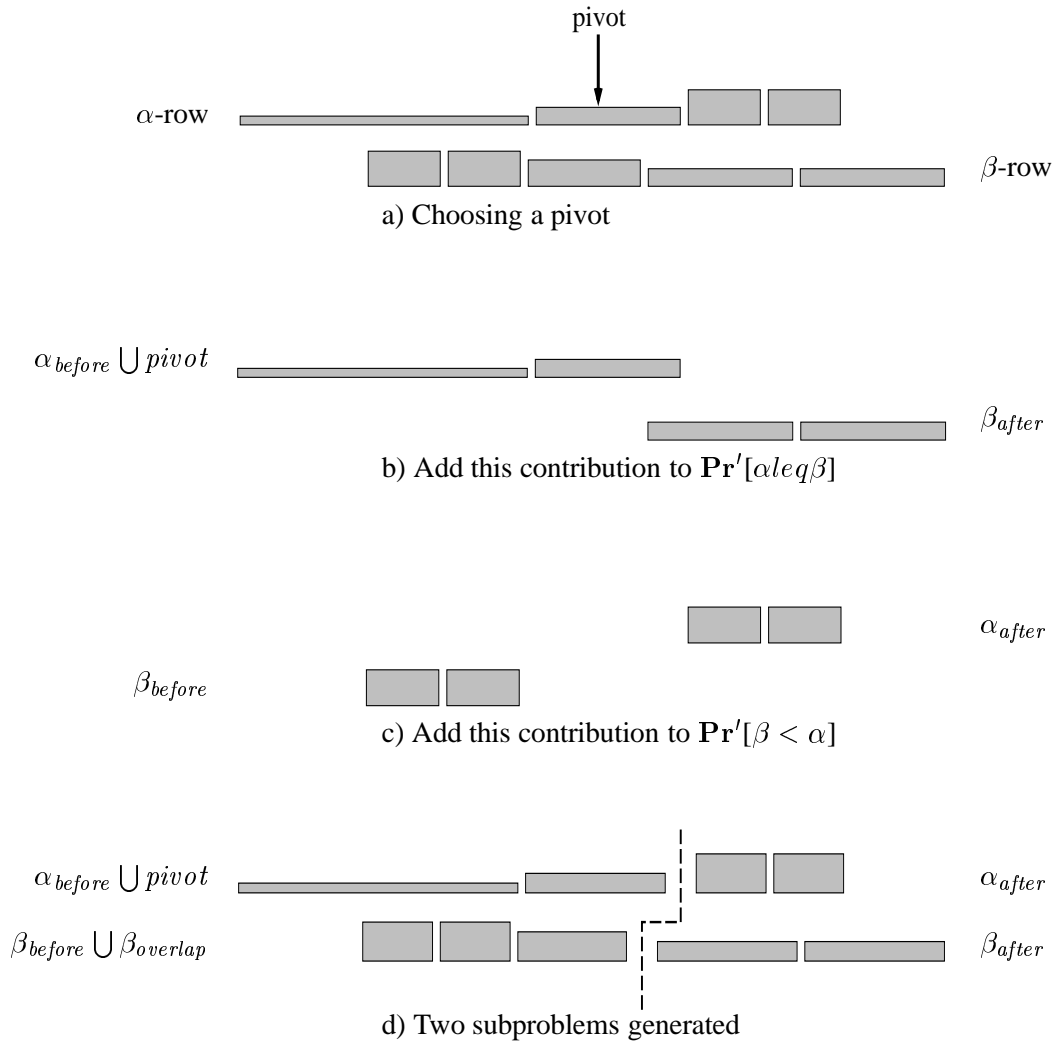
a) Choosing a pivot

b) Add this contribution to $\mathbf{Pr}'[\alpha leq \beta]$

c) Add this contribution to $\mathbf{Pr}'[\beta < \alpha]$

d) Two subproblems generated

Figure 10: A rod counting operation

8.2 The *Replace* **Function**

The *Replace* function changes an indeterminate time value in one of four ways. It could replace the value with the lower support, replace it with the upper support, replace it with the expected value, or leave the value unchanged. The straightforward pseudo-code for *Replace* for an instant is given in Figure 12. The code for starting and terminating period time values, as well as for intervals, is similar, and omitted for brevity. In this pseudo-code we assume that the expected value for the probability mass function has been pre-computed for the normal interval [0,1]. This value is cached and used by the *Replace* function to compute the expected value for a non-normal interval, i.e., a period of indeterminacy.

8.3 *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 $Before_I$ those in another by performing logical *and*s between rows in the boolean table of previously computed $Before_I$ results.

function *PROB_α_LEQ_β*(**in** α, β : event; **in** γ : integer) : set of boolean;

    const

        P : integer = 256; C : integer = 65536;

        α_{tree}, β_{tree} : probability_mass_function_tree(C,P);

        plausibility_map : array[1..100] **of** $1..\lfloor P^2/100 \rfloor$;

    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) $\vee$ (*false_pairs* < 0) **then return**;

            if (($\alpha_{to} - \alpha_{from}$) = 0) $\vee$ (($\beta_{to} - \beta_{from}$) = 0) **then return**;

        /* Calculate pivot */

            pivot $\leftarrow \alpha_{from} + ((\alpha_{to} - \alpha_{from})$ **div** 2);

        /* Figure out α's contribution */

            $\alpha_{before} \leftarrow pivot - \alpha_{from}$;

            $\alpha_{after} \leftarrow \alpha_{to} - pivot$;

        /* Figure out the chronon in which the pivot ends using binary search */

            $\alpha_{mid} \leftarrow$ binary_search(α_{tree}, *pivot*);

        /* Find the rod in β just after the pivot's chronon using binary search */

            leaf $\leftarrow$ binary_search(β_{tree}, α_{mid});

        /* Figure out β's contribution */

            $\beta_{before} \leftarrow$ *leaf.left_rods* $- \beta_{from}$;

            $\beta_{after} \leftarrow \beta_{to} -$ *leaf.right_rods*;

        /* How much is the total contribution? */

            true_pairs $\leftarrow$ *true_pairs* $- ((\alpha_{before} + 1) \times \beta_{after})$;

            false_pairs $\leftarrow$ *false_pairs* $- (\alpha_{after} \times \beta_{before})$;

        /* Continue counting */

            ROD_COUNTING(α_{from}, $\alpha_{from} + \alpha_{before}$, β_{from}, $\beta_{from} + \beta_{before}$);

            ROD_COUNTING($\alpha_{to} - \alpha_{after}$, α_{to}, $\beta_{to} - \beta_{after}$, β_{to});

        end;

    begin

        true_pairs $\leftarrow$ *plausibility_map*[γ];

        false_pairs $\leftarrow P^2 -$ *true_pairs*;

        ROD_COUNTING($1, P, 1, P$);

        if (*true_pairs* ≤ 0) **return** {True};

        else if (*false_pairs* ≤ 0) **return** {False};

        else return {};

    end;

Figure 11: The pivoting algorithm

```
function ReplaceInstant(in δ : credibility; in α : instant) : instant;
    const
        α_tree : probability_mass_function_tree(C, P);
    var
        j : integer;
    begin
        case δ
            INDETERMINATE: return α;
            MAX: return α*;
            MIN: return α*;
            EXPECTED: return α* + (α_expected × (α* − α*))
    end; { ReplaceInstant }
```

Figure 12: The *ReplaceInstant* algorithm

8.4 Impact of Indeterminacy on the Determinate Implementation

In parallel with the theorem of reducibility given in Section 5.9, conventional SQL queries on determinate relations will incur no additional execution overhead under the new semantics, and executing such queries on indeterminate relations will add little overhead since $Before_I$'s for a plausibility of 100 are very efficient.

9 Empirical Analysis of the Implementation

We implemented the indeterminate operations in a prototype system for temporal support called MULTI-CAL [SSD+92]. MULTICAL is written in the C programming language[7]. We complied the code using the GNU C compiler, version 2.7.2, with compiler optimization fully enabled. We used a precision of 2^8 and a coarseness of 2^{16} in the code for the indeterminate $Before_I$. These values limit the maximum error in the pivoting algorithm to less than 1%. We also implemented $Before_I$ with a maximum possible error of 10%. This version of $Before_I$ performs (at most) 14 pivots as discussed in Section 8.1.4.

We tested the performance of each operation in isolation first. All tests were performed on a DEC-Alpha 3000 Model 400 (a 133.33 MHZ machine). The timings for each test were collected using the atom tool [SE94], which allowed us to count machine cycles. Table 3 shows the results for each operation. The execution times shown in the table only include the cycles actually spent within the function, and exclude the cost of the function call itself. 'NA' denotes "not applicable." The *Replace* and determinate *Before* operations are relatively cheap. The worst case for *Replace* occurs with the EXPECTED credibility since it involves a floating point multiplication. The best and worst case behavior of $Before_I$ vary significantly.

To further examine $Before_I$'s behavior, we devised several additional tests for the indeterminate $Before_I$. These tests were designed to capture both the worst case and the expected case performance of the pivoting algorithm. The worst case for $Before_I$ happens when the two indeterminate instants span the same chronons and have uniform distributions. We tested this worst case performance of $Before_I$ on a pair of instants, each of which has a period of indeterminacy of one million chronons. The results are given in Figure 13. The graph plots the execution time (in machine cycles) of $Before_I$ 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 $Before_I$. The average worst case $Before_I$ operation with 1% error, across all plausibilities, is

---

[7]C code for the operations discussed in the previous section, as well as for all the experiments discussed here, is available via the WWW at http://www.cs.jcu.edu.au/~curtis/htmls/indeterminacy.html.

Operation	Determinate Cost	Indeterminate Best Case Cost	Indeterminate Worst Case Cost
Determinate *Before*	73	NA	NA
Indeterminate *Before$_I$* - 1% error	NA	77	86930
Indeterminate *Before$_I$* - 10% error	NA	77	4960
Replace	NA	33	239
Set_Before	NA	0.4	0.4

Table 3: Timings on indeterminate operations (in machine cycles)
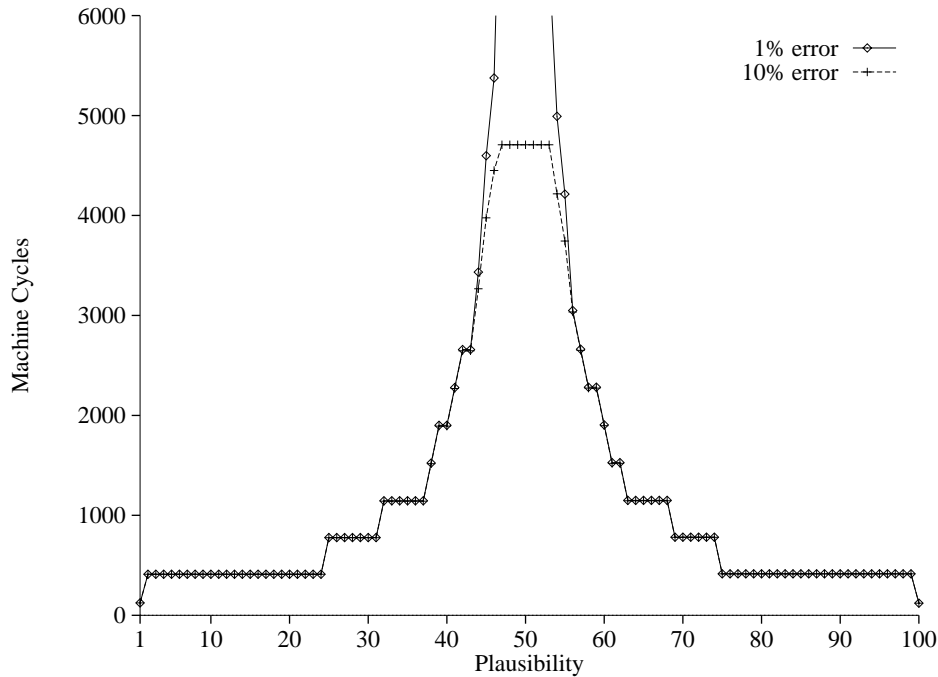
Figure 13: Worst-case performance of *Before$_I$*

approximately 2781 machine cycles, with a high of 86930 machine cycles at a plausibility of 50, and a low of 77 machine cycles. With a maximum error of 10%, the average worst case for *Before$_I$* is somewhat less, 1246 machine cycles, with a high of 4960 machine cycles 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 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 14 shows the results. The z-axis is the cost (in machine cycles) of a single call to *Before$_I$* (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$_I$* is very cheap. If the instants overlap, *Before$_I$* only exhibits poor performance along a central ridge. Note that the graph in Figure 13 is a slice of the graph in Figure 14 at the "even" point along the "relative position" axis.
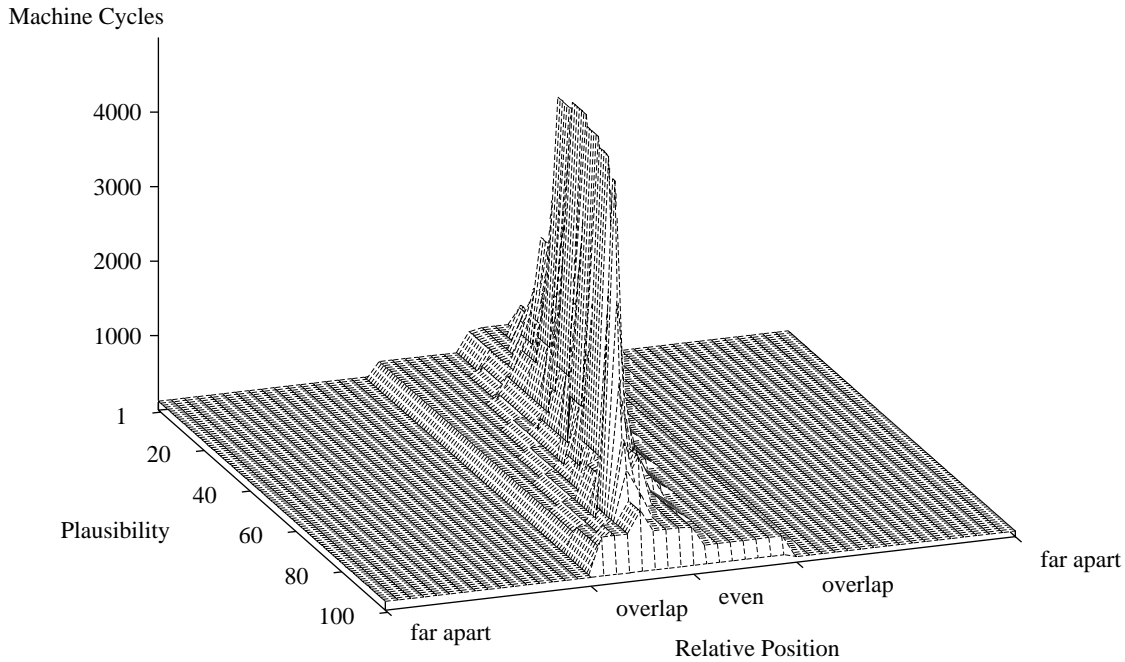
Figure 14: Sliding one instant relative to another performance of $Before_I$

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 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 $Before_I$ on every possible combination of instants (10^2 possible combinations) at plausibilities ranging from 1 to 100. Instants were not compared to themselves. We rerandomized the location of the instants between each test (i.e., per one hundred comparisons). The results are depicted in Figure 15. The graph in Figure 13 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 when the size of the random space is small is the cost of $Before_I$ significant.

To this point we had not determined how much more expensive we expect $Before_I$ to be than $Before$. To measure the relative cost of $Before_I$ we reran the "random placement of instants" test described above on both $Before$ and $Before_I$. But this time we let the size of the random chronon space vary between 1 and
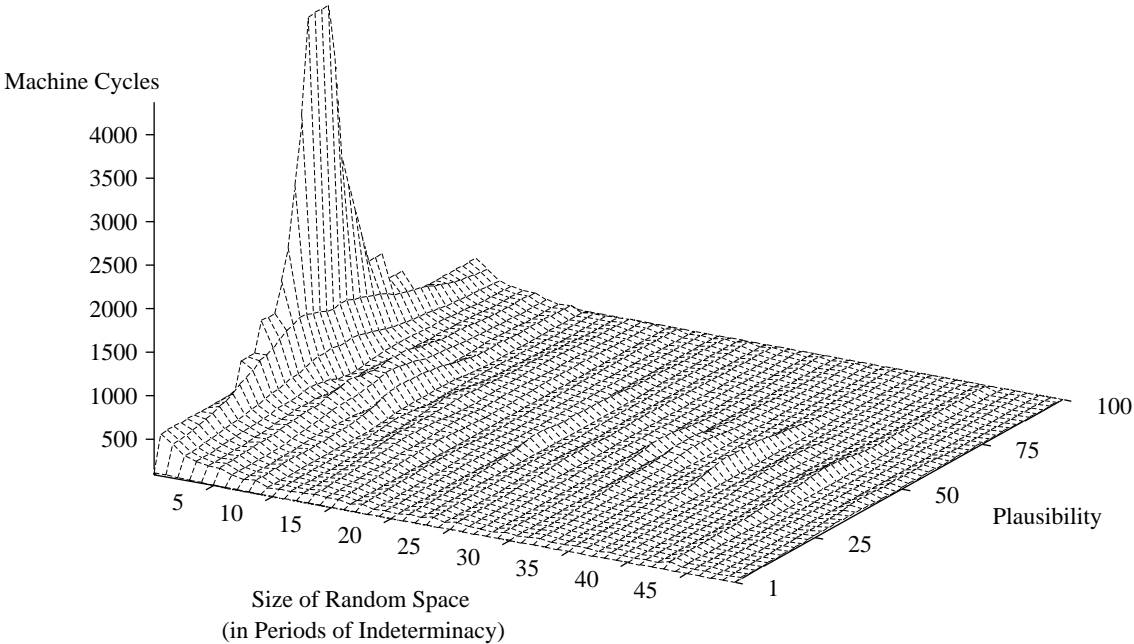
Figure 15: The cost of comparing ten instants randomly placed in a chronon space of varying size

93 days rather than 1 and 50. For each day, we averaged the cost of the *Before$_I$* operation across all the plausibilities, 1 to 100. The results are plotted in Figure 16. 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), *Before$_I$* is approximately one and one-half times as expensive as *Before*. This difference will remain approximately the same as we further increase the size of the random space.

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 performed in most queries. To measure *Before$_I$* and the other operations in context, we designed a test of a complete query. We hand-compiled the example SQL query given in Figure 3 on page 9 into the series of MULTICAL calls shown in Figure 17. The determinate and indeterminate sequences of calls differ only slightly; the differences are highlighted in italics in the indeterminate sequence. We used the compact indeterminate timestamp formats, which are more expensive to unpack, but have the same space cost as the determinate timestamps used in the experiment. Note that each sequence does $O(n^2)$ unpacks, i.e., all possible combinations of instants and periods are unpacked. An alternative query processing strategy is to unpack each instant and period and cache the results, costing only $O(n)$ unpacks. We also programmed this sequence of calls, which we refer to as the "optimized unpack sequence." For all the sequences we suppressed all input, output, and disk I/O, as these expensive operations tend to dominate the cost of other operations, and also because those costs are identical whether indeterminacy is present or absent.
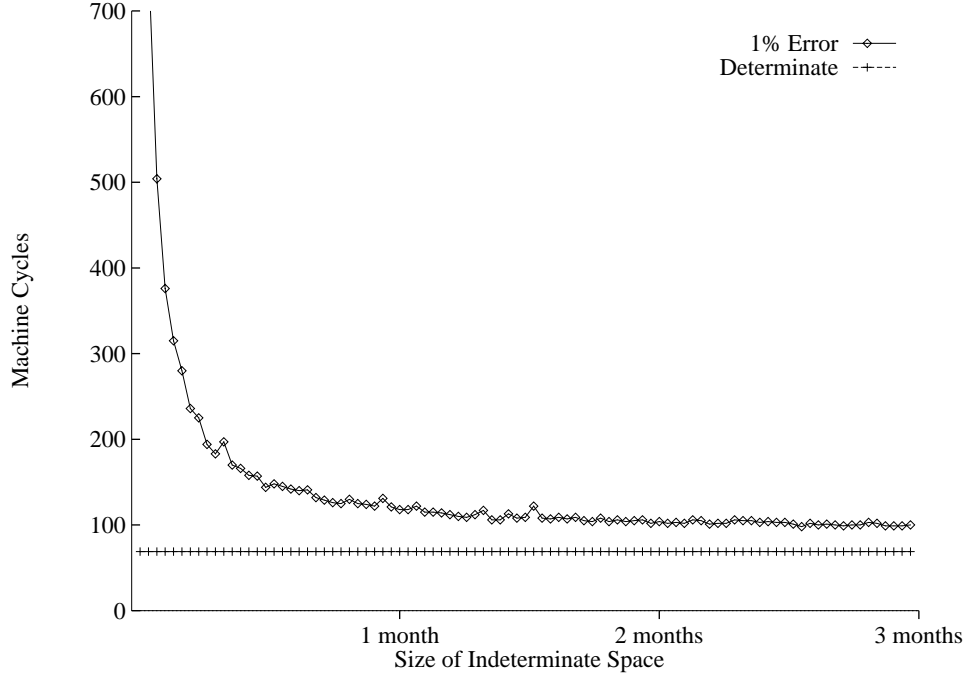
30

Figure 16: The average cost of comparing ten instants randomly placed in a chronon space of varying size

```
{ The determinate sequence }                    { The indeterminate sequence }
    for every combination of r and p                for every combination of r and p
        unpack_instant(r.When);                         unpack_instant(r.When);
        unpack_period(p.During);                        unpack_period(p.During);
        c1 ← Before(r.When, p.Duringₜₒ);                c1 ← Before_I(r.When, p.During_to, 60);
        c2 ← Before(p.During_from, r.When);             c2 ← Before_I(p.During_from, r.When, 60);
        if (c1 and c2) then                             if (c1 and c2) then
            add temp to result                              add temp to result
        end                                             end
    end                                             end
```

Figure 17: MULTICAL calls for example query

To test the query we once again used a variation of "the random placement of instants." We used the tuples shown in Figure 1, but randomly placed the instants in a chronon space of increasing size. We used a period of indeterminacy of one day (86,400 chronons) for the instants in both relations. The results are shown in Figure 18. Except for the rare situations where most of the instants are packed into a relatively tight random space, indeterminacy roughly doubles the CPU cost of the query.

10 Extending Temporal Query Languages

In SQL, valid-time indeterminacy is a form of *value incompleteness* (cf. [Mot90]), where the value of an attribute is not fully known. The *Received* relation in Figure 19 provides examples of value incompleteness
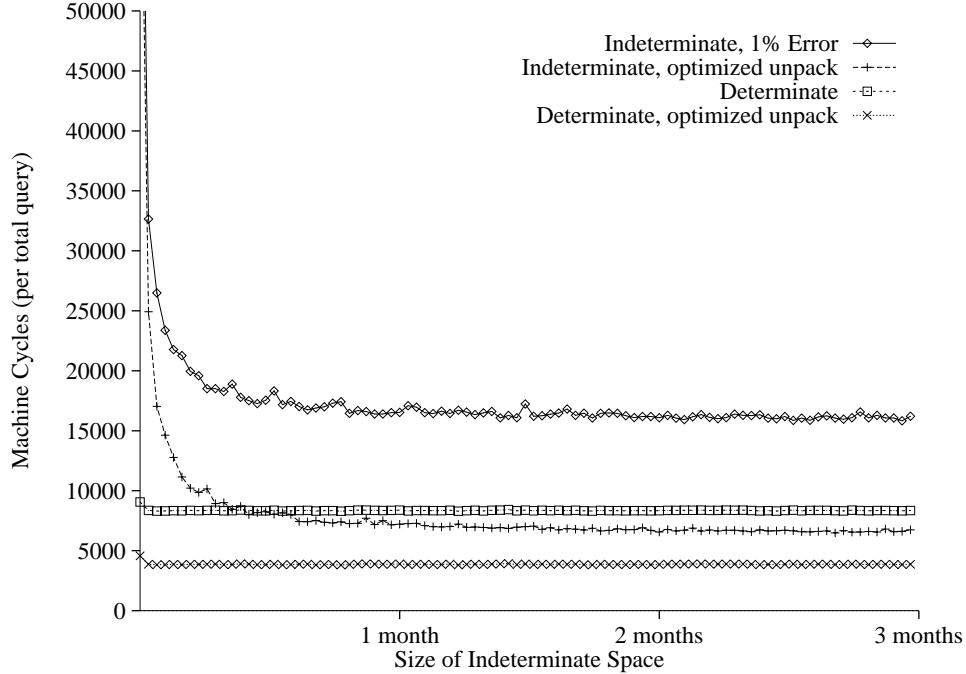
Figure 18: The average cost of the example query per combination of tuples

	Warehouse	Lot_Num	Part	When
s_5	Boeing	40	*unknown*	May 31
s_6	Cessna	70	*some electrical part*	May 31
s_7	Boeing	41	{yoke, throttle}	May 31

Figure 19: Examples of value incompleteness

for the *Part* attribute: a part may exist which we have yet to identify (s_5), 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 showed in Section 3 how to support value incompleteness in instant, period, and interval attributes.

Valid-time indeterminacy can also apply in temporal data models [TCG+93]. In such models, the values themselves vary over time. This behavior is often modeled by associating a timestamp with each tuple. Allowing this implicit timestamp to be indeterminate adds a new source of incompleteness, which we term *tuple valid-time indeterminacy*, which is orthogonal to other sources of incompleteness. In particular, it can peacefully coexist with value incompleteness, of both non-temporal and temporal attributes, and with *tuple incompleteness*, where the membership of a tuple in a relation is not fully determined. We advocate separating the various kinds of indeterminacy, so that users can choose the combination that is most appropriate for their application.

As a concrete example, we now review how tuple valid-time indeterminacy can be added to a specific temporal query language, TSQL2. This language was designed by a committee of eighteen researchers from academia, vendor research labs, and industry [Sno95]. In this section, we consider a variant of TSQL2 termed ATSQL2 that is being proposed for incorporation into the SQL3 standard [SBJS96]. This variant differs from TSQL2's data model in allowing duplicates and timestamping tuples with periods instead of with temporal elements (sets of maximal periods).

TSQL2 supports three temporal dimensions: valid time, transaction time, and user-defined time [Sno95].

The approach to value incompleteness discussed to this point applies to user-defined time. In the remainder of this section, we consider tuple valid-time indeterminacy. Specifically, we allow the period valid timestamp of a tuple to be an indeterminate period. As the timestamp indicates when the fact represented by the tuple was valid in reality, an indeterminate timestamp indicates that it is not known precisely when the fact became true, or no longer was true. Note, however, that we *do* know that the fact was true at some point (orthogonally, tuple incompleteness could be added, to add the uncertainty in whether the fact was ever true).

10.1 Syntactic Extensions

As TSQL2 is an extension of SQL-92, we start with the syntactic extensions introduced in Section 4, specifically, indeterminate temporal attributes in the create table and alter table statements, correlation credibility in the from clause, and ordering plausibility in the where clause. Interestingly, these are all that are necessary to add tuple valid-time indeterminacy to TSQL2. As an example, to define the *Received* relation as an indeterminate valid-time relation, we simply specify an indeterminate period type as the implicit timestamp.

```
CREATE TABLE Received(Warehouse  CHARACTER(30),
                      Lot_Num     INTEGER,
                      Part        CHARACTER(40))
AS VALIDTIME INDETERMINATE PERIOD(DATE) WITH NONSTANDARD DISTRIBUTION;
```

Note that the When attribute is no longer present; instead the timestamp is implicit. The implicit timestamp associated with a correlation name (e.g., r) is accessible within queries via the function VALIDTIME(r); its value is impacted by the correlation credibility and ordering plausibility identically to the explicit values in the tuple.

10.2 Semantic Extensions

The semantics of ATSQL2 is specified in terms of the semantics of SQL-92. Since ATSQL2 with tuple valid-time indeterminacy is an extension of SQL-92 with value valid-time indeterminacy, the major change is to replace $[\![\ ]\!]_{SQL}$ in the ATSQL2 semantics with $[\![\ ]\!]_{ind}$.

ATSQL2 has three modes, specified using new reserved words: *temporal upward compatibility*, in which only the current state is used, *sequenced semantics*, in which the query is applied with SQL semantics at each point in time, and *nonsequenced semantics*, in which the table is treated as a conventional table with an additional period attribute.

The nonsequenced semantics is already dealt with in the rest of the paper, with the proviso that the correlation credibility may replace the implicit period timestamp with a determinate value. In temporal upward compatibility, all tables are snapshot as of now. We use the overlap operator to determine those tuples valid at now; the operator is already defined for indeterminate periods. The specified plausibility, or the default if a plausibility is not specified, is used for the overlap. For the sequenced semantics, a tuple will be considered to be valid at a given time point if the indeterminate period overlaps with the given time point, using the specified plausibility.

The final change required is that to support coalescing. The user can specify in ATSQL2 (via "(PERIOD)" in the from clause) that a relation be *coalesced*: tuples with identical values for the explicit attributes (termed *value-equivalent*) with timestamps that overlap or meet in valid time are merged into a single tuple with a timestamp that is the union of the timestamps of the original tuples. If the periods do not overlap or meet then the tuples are unchanged. The semantic function *Coalesce* repeatedly coalesces overlapping pairs, until there is no more overlap.
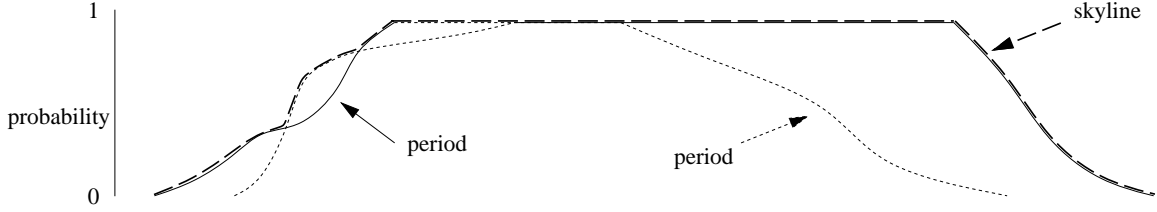
Figure 20: The result of coalescing two indeterminate periods follows the skyline

10.3 Semantics of Coalescing

In the indeterminate semantics, the coalescing operation must be extended to handle periods with indeterminacy[8].

Definition 6 (Indeterminate Coalescing) *For two value-equivalent tuples that have valid-time periods $[\alpha, \beta]$ and $[\eta, \delta]$ which overlap with plausibility 100 (i.e., definitely overlap), the indeterminate coalescing of the periods $[\alpha, \beta]$ and $[\eta, \delta]$ produces $[s, t]$ where*

$$s = (\mathbf{min}(\alpha_*, \eta_*) \sim \mathbf{min}(\alpha^*, \eta^*), P_s),$$

$$t = (\mathbf{max}(\beta_*, \delta_*) \sim \mathbf{max}(\beta^*, \delta^*), P_t),$$

P_s *is the probability mass function such that its cumulative density function F_s behaves as follows.*

$$F_s(x) = \begin{cases} F_\alpha(x) & \text{if at time } x, F_\alpha(x) > F_\eta(x) \\ F_\eta(x) & \text{otherwise} \end{cases}$$

P_t *is the probability mass function such that its complementary cumulative density function F_t' behaves as follows.*

$$F_t'(x) = \begin{cases} F_\beta'(x) & \text{if at time } x, F_\beta'(x) > F_\delta'(x) \\ F_\delta'(x) & \text{otherwise} \end{cases}$$

If the periods do not definitely overlap, then the periods are already coalesced. ∎

While this definition may appear complicated, the underlying idea is simple. Figure 20 shows a probability "profile" for two overlapping periods. Both periods 'ramp up' to definite information during their respective starting instants and 'fall off' to no information during the terminating instants. The union of the two periods is the "skyline" of the two periods. The skyline represents the maximal extent of both the determinate and indeterminate portions of the periods.

10.4 Implementation of Coalescing

The implementation techniques developed in Section 8 apply directly to the indeterminate portions of TSQL2, except for coalescing, which requires a new approach. A straightforward implementation of indeterminate coalescing could be very costly since, in some cases, the operator must dynamically compute a new probability mass function. Not only would dynamically computing such functions quickly exhaust the limited space of probability mass function names (recall from Section 7 that the timestamp formats limit the number

---

[8]This coalesce operator contends with indeterminate attribute values, and so differs from Dey and Sarkar's *coalescence* operations, which handles indeterminate *tuples* [DS96].

of mass functions to 2^{16} possibilities), but the efficient implementation of *Before$_I$* would also suffer since it depends upon the pre-computed approximation of the mass function.

Our solution is to accept some information loss during coalescing and avoid the expense of dynamically computing a mass function by substituting the missing distribution for the new mass function. In other words, for two valid-time periods $[\alpha, \beta]$ and $[\eta, \delta]$ which definitely overlap, the implementation of indeterminate coalescing produces the period $[s, t]$ where

$$
s = \begin{cases}
\alpha & \text{if for every time } t, F_\alpha(t) \geq F_\eta(t) \\
\eta & \text{if for every time } t, F_\eta(t) \geq F_\alpha(t) \\
(\mathbf{min}(\alpha_*, \eta_*) \sim \mathbf{min}(\alpha^*, \eta^*), \text{missing}) & \text{otherwise}
\end{cases}
$$

and

$$
t = \begin{cases}
\beta & \text{if for every time } t, F'_\beta(t) \geq F'_\delta(t) \\
\delta & \text{if for every time } t, F'_\delta(t) \geq F'_\beta(t) \\
(\mathbf{max}(\beta_*, \delta_*) \sim \mathbf{max}(\beta^*, \delta^*), \text{missing}) & \text{otherwise}
\end{cases}
$$

But even this 'weakened' version of coalescing is much too expensive since the cumulative density functions for two instants must be computed at each time in a period of indeterminacy to determine if one function dominates another. However, it is easy to determine dominance in some special, but common cases. For instance, for a pair of starting instants, if both instants have the same probability mass function, and one instant's period of indeterminacy starts before the other instant's period of indeterminacy starts and ends before the other's ends, then the earlier instant is dominant. For example, the starting instant $(1\sim5, \text{uniform})$ dominates the instant $(3\sim6, \text{uniform})$. The implementation of indeterminate coalescing in $Coalesce'$ only checks for dominance in these special cases, and substitutes the missing mass function in all other cases as shown in Figure 21.

10.5   Summary

To support tuple valid-time indeterminacy in the variant of TSQL2 being proposed for SQL3, no syntactic extensions beyond those discussed for conventional SQL, presented in Section 4, are needed. The only significant semantic extension is to support coalescing. We speculate that our approach to valid-time indeterminacy, both the value incompleteness variety and the tuple valid-time incompleteness variety, can be added in a similarly straight-forward manner to any of the many temporal query languages proposed in the literature.

11   Related Work

Despite the wealth of research on adding incomplete information to databases [Dyr93, Par96], 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., [Vas79, Zan84, Cod90, Dat86]). Another primary research thrust has studied the applicability of fuzzy set theory to relational databases (e.g., [DPT88, ZK85, Pra93]). There is also extensive AI literature on integrating various combinations of probabilistic reasoning, temporal reasoning, and planning; Kraus and Subrahmanian provide a nice summary of that literature [KS96].

We first place our work in the context of *value* and *tuple* incompleteness, then examine in detail several papers that concern *temporal* incompleteness.

```
function COALESCE(in [α, β], [η, δ] : period) : set-of-periods;
    var
        s, t : instant;
    begin
        if not overlap(100, [α, β], [η, δ]) then return {[α, β], [η, δ]}

    { Compute starting instant }
        if α* < η* then s ← α
        else if α* > η* then s ← η
        else if Pα is Pη then
            if α* < η* and α* < η* then s ← α
            else if α* > η* and α* > η* then s ← η
            else s ← (min(α*, η*) ∼ min(α*, η*), missing)

    { Compute terminating instant }
        if β* < δ* then t ← β
        else if β* > δ* then t ← δ
        else if Pβ is Pδ then
            if β* > δ* and β* > δ* then t ← β
            else if β* < δ* and β* < δ* then t ← δ
            else t ← (max(β*, δ*) ∼ max(β*, δ*), missing)
        return {[s,t]}
    end; { COALESCE }
```

Figure 21: Indeterminate Coalescing

Information that is valid-time indeterminate is similar to disjunctive information, especially in the context of deductive databases [LS90]. Disjunctive information is a collection of facts, one (or more) of which is true. A set of possible chronons is of the exclusive-or variety of disjunctive information (only one disjunct is true) [Ola92]. Valid-time indeterminacy 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 to model situations where an attribute value could be one of several more or less likely values [BGMP90, BGMP92, FR96, GH86, LLRS96, Lee92, TCW93]. Probabilistic weights have also been appended to tuples, where the weight is the probability that the tuple belongs to the relation [CP87, DS96, FR96, KS93b, KS93a, LLRS96, Lee92, TCW93, Zim92]. Decision support systems, vague queries, information retrieval and data mining have also utilized probabilistic information [Won82, Fuh90, HS93, VCC96]. Our work concerns only probabilities in attribute values and can be seen as an extension of the Probabilistic Data Model (PDM) [BGMP92]. In the 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 the PDM each element in a set of possible values is stored and processed separately. The costs 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).

36

Dey and Sarkar extend this data model, in part to render it in first normal form and to permit a more general join operation [DS96]. More recently, Lakshmanan, Leone, Ross and Subrahmanian have extended the PDM to eliminate several of the assumptions, including independence of elements, in their ProbView system [LLRS96]. In particular, they permit a range of Cartesian product operators to be defined, with the situation dictating the appropriate strategy.

We could not adopt the PDM approach or its successors to support temporal indeterminacy, since there might be several million elements in a set of possible chronons. Representing each alternative with an associated probability is impractical. Due to the unique nature of valid-time indeterminacy, a different approach was required.

We now turn to the literature of temporal incompleteness, which, unlike our approach, does not employ probabilities.

In the earliest work on incomplete temporal information, an indeterminate instant was modeled with a set of possible chronons [Sno82]. *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) [Sch87]. Our current approach allows a probability mass function to be associated with each indeterminate instant, and does not require a multi-valued logic, though we do use such a logic in the operational semantics.

Dutta uses a fuzzy set approach to handle *generalized temporal events* [Dut89]. 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 [DPT88]). 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., [Vit83, DP89]), 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 [KE92, KE94]. Bitemporal elements combine transaction time and valid time in the same temporal element. Since TSQL2 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 period 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). 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.

Another proposed model intertwines support for value and temporal incompleteness [GNP92]. By combining the different kinds of incomplete information, a wide spectrum of attribute values are simultaneously modeled, including values that are completely known, values that are unknown but are known to have occurred, values that are known if they occurred, and values that are unknown even if they occurred. In our approach, value incompleteness, tuple incompleteness, and tuple valid-time indeterminacy are orthogonal. By combining valid-time indeterminacy with other kinds of incomplete information we can support each of the kinds of incomplete information found in Gadia et al., plus others (e.g., fuzzy value incompleteness). Another difference between our approach and theirs is that they make no use of probabilistic information. The

user cannot express his or her credibility in the underlying data nor plausibility in the temporal relationships in the data.

Gadia et al. prove reliability for their model, as we did for ours, in Section 5.9. They also showed that except for certain cases of selection, their operators were maximal. The same holds for our operators. Hence, both of our models (with the exception of certain cases of selection in their language) are theoretically sound.

While the possible and definite limits to incomplete temporal information are well-understood, the *cost* of querying incomplete temporal information can be prohibitive. Satisfaction of general constraints, that is, a well-formed formulae consisting of temporal variables, predicates and boolean connectives (including negation), and allowing substitution of indeterminate instants is, not surprisingly, NP-hard [vB91, DB88].

Koubarakis showed that by restricting the kinds of constraints allowed, polynomial time algorithms can be obtained [Kou93]. Koubarakis proposed a temporal data model with global and local inequality constraints on the occurrence time of an event. 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 ($2\,\mathrm{PM} < \alpha < \beta$), 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 restricts the kinds of constraints allowed (to disjunctions of inequalities) and achieves polynomial time complexity for retrieving information.

Another constraint-based temporal reasoner is LaTeR, which has been successfully implemented [BCTP95]. LaTeR similarly restricts the kinds of constraints allowed (to conjunctions of linear inequalities), but this class of constraints includes many of the important temporal predicates. LaTeR must construct a constraint network during insertion or updating of temporal information, but uses this network very efficiently when retrieving information to achieve $O(n^3)$ performance.

The primary difference between these constraint-based models and ours (other than probabilities) is that SQL (like most relational database models) does not allow inter-tuple constraints. Tuples in SQL relations are "row-independent," that is, no information is shared between tuples. Since the indeterminate data model is based on SQL, it makes no overt provisions for sharing indeterminate information between tuples. Only the constraints given in the query are used to relate tuples. Further, we, like most others, assume that the variables in a query are independent. Our motivation is to avoid the expensive computation of dependent probabilities, but the independence assumption also yields an $O(nr)$ complexity for evaluating an unrestricted query consisting of temporal predicates and boolean connectives (where n is the number of predicates in the query, and r is the number of tuples). However, constraint-based models are able to model temporal information that we cannot.

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

12   Summary and Future Work

This paper has extended the syntax and formal semantics of SQL to support valid-time indeterminacy and has described an efficient implementation for that support. We also showed how the concepts can be applied to temporal query languages, in particular to the variant of TSQL2 being proposed for incorporation into SQL3.

We return to the goals enumerated in Section 1. The syntactic extensions are minimal. In particular, we provide the user with two controls on the retrieval process, correlation credibility and ordering plausibility. We have augmented the create table and alter table statements to specify which attributes incorporate valid-time indeterminacy and to identify which timestamp format to use, extended the from clause to specify

38

correlation credibility using an optional with clause, extended the select statement to specify ordering plausibilities, and added variants to the set statement to specify default plausibilities and credibilities. Correlation credibility changes the information available to query processing; it replaces indeterminate with determinate information. Ordering plausibility controls the construction of an answer to a query using the pool of credible information. A temporal expression is satisfied if there exists a plausible ordering (to the level specified by the user) that satisfies it. The approach is orthogonal to those proposed by others to handle value incompleteness and generalized events (cf. Section 10), has an intuitive semantics (Section 5), retains the first normal form structure of SQL and TSQL2 (Section 3), refines previously proposed techniques to handle multiple granularities of time (Section 7), is temporally upward compatible, in that it reduces to SQL's and ATSQL2's semantics in the absence of indeterminacy (Section 5.3), reduces to SQL's semantics even in the presence of indeterminacy, when the new constructs are not used (Section 5.7), and has been proven to be both reliable and maximal (Section 5.9).

While these languages extensions are highly expressive, this paper demonstrates that they can be efficiently implemented. We showed how indeterminate instants with a uniform probability mass function or mass function that is missing can be represented in only 64 bits in most cases; for user-defined distributions the common representation is only 96 bits (Section 7). The operational semantics was shown to be correct (Section 5.9); the rod-counting algorithm is efficient (Section 8.1.4), and the undercounting of this algorithm is minimal (also Section 8.1.4). We implemented the functions required by the altered semantics (Section 8), and demonstrated that the implementation roughly doubles the CPU cost of a query when indeterminacy is present and plausibilities and credibilities less than 100 are specified, while having little impact on its disk I/O time (Section 9).

This paper only considers the select, create table, and alter table statements of SQL and TSQL2. The modification statements (append, delete, and update) as well as views, integrity constraints, assertions, and cursors, can be extended in an analogous manner. The approach espoused here has been adopted in TSQL2 [DS95a].

One important area of future research is extension of indexing methods to support indeterminate values. Most extant indexing approaches assume a total order on the underlying domain. Only for an ordering plausibility of 100 does such a total order exist for indeterminate instants (cf. Section 5.4.1). Indexing methods need to be extended to support arbitrary ordering plausibilities, to be specified at query time.

Algorithm improvements on the functions discussed in Section 8 are certainly possible. While such improvements might reduce execution time considerably in special cases, we doubt that they will have much impact on the average case.

A useful extension of the current work would be to use periods instead of values to express plausibility. For instance, the user could constrain retrieval to tuples that "overlap March, 1984" to "within a year" (this has been termed a "band join" [DNS91] or a "fuzzy temporal equi-join" [LM91]). This possibility can be seen as an extension of the present paper, specifically, as a refinement of the $Before_I$ operation.

Finally, it is always useful to consider increasing the expressive power of the data model and query language. In particular, one important assumption we make throughout is that tuples are row-independent, with no information shared between indeterminate tuples. Most of the other approaches 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 indeterminate instants are modeled by contiguous sets of possible chronons. We do not support noncontiguous sets which could model indeterminate instants such as "it happened yesterday morning or this morning." We exploit both of these assumptions to achieve efficiency in representation and in query processing. We conjecture that relaxing either of these assumptions will cause the query evaluation complexity to return to cubic (or worse), thereby rendering the performance unacceptable for many temporal database applications.

Acknowledgments

References

[All83] J .F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[BBJS96] J. Bair, M. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Business Informatics (Wirtschaftsinformatik)*, to appear, 1997, 19 pages.

[BCTP95] V. Brusoni, L. Console, P. Terenziani, and B. Pernici. Extending Temporal Relational Databases to Deal with Imprecise and Qualitative Temporal Information. In S. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, pp. 3–22, Zurich, Switzerland, September 1995. Proceedings of the International Workshop on Temporal Databases, Springer Verlag.

[Bel77] N. Belnap. A Useful Four-Valued Logic. In *Modern Uses of Many-valued Logic*, G. Epstein and J. M. Dunn (eds), D. Reidel, pp. 8–37, 1977.

[BGMP90] D. Barbará, H. García-Molina, and D. Porter. A Probabilistic Relational Data Model. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, pages 60–74, Venice, Italy, March 1990.

[BGMP92] D. Barbará, H. García-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, October 1992.

[Cod90] E. F. Codd. Missing Information and Response to Technical Criticisms Regarding Missing Information, chapters 8 and 9 of *The Relational Model for Database Management: Version 2*, pp. 169–206. Addison-Wesley Publishing Company, Inc., 1990.

[CP87] R. Cavallo and M. Pittarelli. The Theory of Probabilistic Databases. In P. Hammersley, editor, *Proceedings of the International Conference on Very Large Databases*, pp. 71–81, Brighton, England, September 1987.

[CR87] J. Clifford and A. Rao. A Simple, General Structure for Temporal Domains. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, pp. 23–30, France, May 1987. AFCET.

[Dat86] C. J. Date. Null Values in Database Management, chapter 15 of *Relational Database: Selected Writings*, pp. 313–334. Addison-Wesley, Reading, MA, 1986.

[DB88] T. Dean and M. Boddy. Reasoning About Partially Ordered Events. *Artificial Intelligence*, 36(3):375–399, October 1988.

[DNS91]   D. DeWitt, J. Naughton, and D. Schneider.   An Evaluation of Non-equijoin Algorithms.   In *Proceedings of the International Conference on Very Large Databases*, pp. 443–452, 1991.

[DP89]   D. Dubois and H. Prade. Processing Fuzzy Temporal Knowledge. *IEEE Transactions of Systems, Man and Cybernetics*, 19(4):729–744, 1989.

[DPT88]   D. Dubois, H. Prade, and C. Testamale.   Handling Incomplete or Uncertain Data and Vague Queries in Database Applications, chapter 6 of *Possibility Theory: An Approach to Computerized Processing of Uncertainty*, pp. 217–257. Plenum Press, New York and London, 1988.

[DS93]   C. E. Dyreson and R. T. Snodgrass. Valid-time Indeterminacy. In *Proceedings of the International Conference on Data Engineering*, pp. 335–343, Vienna, Austria, April 1993.

[DS95a]   C. E. Dyreson and R. T. Snodgrass. Temporal Indeterminacy. Chapter 18 of [Sno95], pp. 327–346.

[DS95b]   C. E. Dyreson and R. T. Snodgrass.   A Timestamp Representation.   Chapter 25 of [Sno95], pp. 475–500.

[DS96]   D. Dey and S. Sarkar. A Probabilistic Relational Model and Algebra. *ACM Transactions on Database Systems*, 21(3):339–369, September 1996.

[DSS95]   C. E. Dyreson, M. Soo, and R. T. Snodgrass. The Data Model for Time. Chapter 6 of [Sno95], pp. 97–101.

[Dut89]   S. Dutta. Generalized Events in Temporal Databases. In *Proceedings of the Fifth International Conference on Data Engineering*, pp. 118–126, Los Angeles, CA, February 1989.

[Dyr93]   C. Dyreson.   A Bibliography on Uncertainty Management in Information Systems.   In *Proceedings of the Workshop on Uncertainty Management in Information Systems: From Needs to Solutions*, pp. 187–220, Avalon, Santa Catalina, California, April 1993.

[FR96]   N. Fuhr and T. Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Transactions on Information Systems*, 1996, to appear.

[Fuh90]   N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. In *Proceedings of the International Conference on Very Large Databases*, Brisbane, Australia, 1990.

[GC66]   S. Goudsmit and R. Claiborne. *Time*. Time Inc., New York, 1966.

[GH86]   E. Gelenbe and G. Hebrail. A Probability Model of Uncertainty in Data Bases. In *Proceedings of the International Conference on Data Engineering*, pp. 328–333, Los Angeles, CA, February 1986.

[GNP92]   S. K. Gadia, S. Nair, and Y.-C. Poon. Incomplete Information in Relational Temporal Databases. In *Proceedings of the International Conference on Very Large Databases*, Vancouver, Canada, August 1992.

[HS93]   M. Henrion and J. Suermondt. Probabilistic and Bayesian Representations of Uncertainty and Information Systems: A Pragmatic Introduction. In *Proceedings of the Workshop on Uncertainty Management in Information Systems: From Needs to Solutions*, pp. 71–90, Avalon, Santa Catalina, California, April 1993.

[JCE+94] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia (eds). A Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52–64, March 1994.

[KE92] V. Kouramajian and R. Elmasri. A Generalized Temporal Model. Tech. Report, University of Texas at Arlington, February 1992.

[KE94] V. Kouramajian and R. Elmasri. A Generalized Temporal Model. In the Uncertainty in Databases and Deductive Systems Workshop, Ithaca, NY, November, 1994.

[Kou93] M. Koubarakis. Representation and Querying in Temporal Databases: The Power of Temporal Constraints. In *Proceedings of the the International Conference on Data Engineering*, pp. 327–334, Vienna, Austria, April 1993.

[KS93a] Y. Kornatzky and S. Shimony. A Probabilistic Object-oriented Data Model. TR FC 93-04, Ben-Gurion University, May 1993.

[KS93b] Y. Kornatzky and S. Shimony. A Probabilistic Spatial Data Model. In *DEXA'93*, Prague, Czech Republic, September 1993.

[KS96] S. Kraus and V. S. Subrahmanian. Multiagent Reasoning with Probability, Time and Beliefs. *International Journal of Intelligent Systems*, 10(5) 459–499, 1994.

[Lee92] S. K. Lee. An Extended Relational Database Model for Uncertain and Imprecise Information. In *Proceedings of the International Conference on Very Large Databases*, Vancouver, Canada, August 1992.

[Lip79] Lipski, W., Jr. On Semantic Issues Connected with Incomplete Information Databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

[LLRS96] V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. ProbView: A Flexible Probabilistic Database System. Submitted for journal publication, 1996.

[LM91] T. Y. Leung and R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. Technical Report CSD-910077, Computer Science Department, UCLA, November 1991.

[LS90] K. C. Liu and R. Sunderraman. Indefinite and Maybe Information in Relational Databases. *ACM Transactions on Database Systems*, 15(1):1–39, March 1990.

[Mel96] J. Melton. SQL/Temporal. ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-0012, July 1996.

[Mot90] A. Motro. Imprecision and Incompleteness in Relational Databases: Survey. *Information and Software Technology*, 32(9):579–588, November 1990.

[MS93] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[Ola92] A. Ola. Relational Databases with Exclusive Disjunctions. In *Proceedings of the Eighth International Conference on Data Engineering*, pp. 328–336, Tempe, AZ, February 1992.

[Par96] S. Parson. Current Approaches to Handling Imperfect Information in Data and Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):353—372, 1996.

[Pet91]   B. W. Petley.   Time and Frequency in Fundamental Metrology. *Proceedings of the IEEE*, 79(9):1070–1077, July 1991.

[Pra93]   H. Prade. Annotated Bibliography on Fuzzy Information Processing. In H. Prade, D. Dubois and R. Yager, editors, *Readings on Fuzzy Sets in Intelligent Systems.* Morgan & Kaufmann, 1993.

[SBJS96]   R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner.   Adding Valid Time to SQL/Temporal.   Change Proposal ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2, November 1996.

[Sch87]   U. Schiel. Representation and Retrieval of Incomplete and Temporal Information. TR DSC-02/87, Universidade Federal Da Paraiba, May 1987.

[Sno82]   R. Snodgrass. *Monitoring Distributed Systems: A Relational Approach*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, December 1982.

[Sno95]   R. T. Snodgrass (editor), I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo and S. M. Sripada. *The Temporal Query Language TSQL2*. Kluwer Academic Pub., 1995.

[SSD$^+$92] M. D. Soo, R. Snodgrass, C. Dyreson, C. S. Jensen, and N. Kline.   Architectural Extensions to Support Multiple Calendars.   TempIS Technical Report 32, Computer Science Department, University of Arizona, Revised May 1992. `ftp://FTP.cs.arizona.edu/tsql/multical`

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

[TCG$^+$93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.). *Temporal Databases: Theory, Design, and Implementation*. Database Systems and Applications Series. Benjamin/Cummings, Redwood City, CA, 1993.

[TCW93]   F. S. C. Tseng, A. L. P. Chen, and W.-P. Wang. Answering Heterogeneous Database Queries with Degrees of Uncertainty. *Distributed and Parallel Databases: An International Journal*, 1(3):281–302, July 1993.

[Vas79]   Y. Vassiliou. Null Values in Database Management–a Denotational Semantics Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 162–169, New York, May 1979.

[vB91]   P. van Beek. Temporal Query Processing with Indefinite Information. *Artificial Intelligence in Medicine*, 3(6):325–339, December 1991.

[VCC96]   S. R. Vasanthakumar, J. P. Collan, and W. B. Croft. Integrating INQUIRY with an RDBMS to Support Text Retrieval. *IEEE Data Engineering*, 19(1):24–33, March 1996.

[Vit83]   M. Vitek. Fuzzy Information and Fuzzy Time. In *Proceedings of the IFAC Symposium Fuzzy Information, Knowledge Representation and Decision Analysis*, pp. 159–162, Marseille, France, 1983.

[Won82]   E. Wong. A Statistical Approach to Incomplete Information in Database Systems. *ACM Transactions on Database Systems*, 7(3):470–488, September 1982.

[Zan84]   C. Zaniolo. Database Relations with Null Values. *Journal of Computer and System Sciences*, 28:142–166, 1984.

[Zim92]   E. Zimányi. *Incomplete and Uncertain Information in Relational Databases*. PhD thesis, Université Libre de Bruxelles, Brussells, Belgium, July 1992.

[ZK85]   M. Zemankova and A. Kandel. Implementing Imprecision in Information Systems. *Information Sciences*, 37:107–141, 1985.

A Proofs of Theorems

Theorem 1 $[\![\,\mathtt{WHERE}\ \langle\text{predicate}\rangle\,]\!]_{ind}(1, r)$ *is reliable.*

Proof: For any where clause W, for all $r' \in \mathbf{C}(r)$, let $c = [\![\,W\,]\!]_{SQL}(r')$. Then $\forall t' \in r'\,(t' \in c \Rightarrow [\![\,\langle predicate\rangle\,]\!]_{SQL}(t'))$. From the definition of $[\![\,W\,]\!]_{ind}$,

$$\forall t' \in r'\,(t' \in c \Rightarrow (\exists t \in r\,(t' \in \mathbf{C}(t) \wedge t \in [\![\,W\,]\!]_{ind}(1, r)))).$$

From the definition of $\mathbf{C}$, it follows that $c \in \mathbf{C}([\![\,W\,]\!]_{ind}(1, r))$. ■

Theorem 2 $[\![\,\mathtt{WHERE}\ \langle\text{predicate}\rangle\,]\!]_{ind}(1, r)$ *is maximal.*

Proof: We need to show $\forall\,\langle\text{where clause}\rangle W, \forall c \in \mathbf{C}([\![\,W\,]\!]_{ind}(1, r)),\ \exists r' \in \mathbf{C}(r)\,(c = [\![\,W\,]\!]_{SQL}(r'))$. For the possible interpretation,

$$\forall\,\langle\text{where clause}\rangle W, \forall c \in \mathbf{C}([\![\,W\,]\!]_{ind}(1, r))\,(\forall t' \in c,\ \exists t \in r\,(\exists t'' \in \mathbf{C}(t) \wedge [\![\,P\,]\!]_{SQL}(t''))).$$

From the definition of $\mathbf{C}$ on relations, it follows that $\exists r' \in \mathbf{C}(r)\,(c = [\![\,W\,]\!]_{SQL}(r'))$. ■

Theorem 3 $[\![\ ]\!]_{ind}$ *and* $[\![\ ]\!]_{op}$ *agree on the possible and definite interpretations.*

Proof: For the definite bound ($\gamma = 100$), we need to show

$$\forall\,\langle\text{predicate}\rangle\ P, \forall r, \forall d\,([\![\,\mathtt{WHERE}\ P\,]\!]_{ind}(100, r) = \{t\,|\,t \in r \wedge T \in [\![\,P\,]\!]_{op}(100, r)\}).$$

From the definition of $[\![\ ]\!]_{ind}$, this is equivalent to $\forall t\,\forall c \in \mathbf{C}(t)\,([\![\,P\,]\!]_{SQL}(c) \Rightarrow T \in [\![\,P\,]\!]_{op}(100, t))$. We show this by induction on the boolean connectives in P, after replacing P with its equivalent in terms of $Before$ (or $Before_I$, for $[\![\ ]\!]_{op}$), $\wedge$ and $\neg$.

Basis: $[\![\,P\,]\!]_{SQL}$ is $Before(a, b)$, where a and b are (instant) values in the tuple t.
From the definition of the completion of a tuple, we must show

$$(\forall c_a \in \mathbf{C}(a),\ \forall c_b \in \mathbf{C}(b)\,(c_a \le c_b)) \Rightarrow T \in Before_I(a, b, 100).$$

The left hand side implies that $\mathbf{Pr}[a \le b] = 1.00$, so $Before_I(a, b, 100) = \{T\}$.

Induction: Assume that $\forall t\,\forall c \in \mathbf{C}(t)\,([\![\,P\,]\!]_{SQL}(c) \Rightarrow T \in [\![\,P\,]\!]_{op}(100, t))$ holds for P containing $i - 1$ boolean connectives. We need to show that this holds for P containing i connectives.

(1) P is $P_1 \wedge P_2$. Applying the definitions of $[\![\ ]\!]_{SQL}$ and $[\![\ ]\!]_{op}$, the following holds from the inductive hypothesis.

$$\forall t\,\forall c \in \mathbf{C}(t)\,([\![\,P_1\,]\!]_{SQL} \wedge [\![\,P_2\,]\!]_{SQL}(c) \Rightarrow T \in ([\![\,P_1\,]\!]_{op}(100, t) \cap [\![\,P_2\,]\!]_{op}(100, t)))$$

It then follows that the inductive hypothesis also holds for P.

(2) P is $\neg P_1$. Again, applying the definitions to the inductive hypothesis,

$$\forall t\, \forall c \in \mathbf{C}(t)\, (\neg [\![\, P_1 \,]\!]_{SQL}) \Rightarrow F \in ([\![\, P_1 \,]\!]_{op}(100, t)).$$

This shows that the the two semantics agree for P containing i predicate connectives.

For the possible bound ($\gamma = 1$), we need to show

$$\forall \langle predicate \rangle\, P, \forall r \forall d([\![\, \mathtt{WHERE}\ P \,]\!]_{ind}(1, r) = \{t \,|\, t \in r \wedge T \in [\![\, P \,]\!]_{op}(1, r)\}.$$

This is equivalent to $\forall t\, \exists c \in \mathbf{C}(t)\, ([\![\, P \,]\!]_{SQL}(c)) \Rightarrow T \in [\![\, P \,]\!]_{op}(1, t))$. Comparing this to the analogous formula in the definite portion of this proof, note that $\forall c$ has been replaced with $\exists c$, and $\gamma = 100$ has been replaced with $\gamma = 1$.

Again, we show this by induction on the number of boolean connectives in P.

Basis: $[\![\, P \,]\!]_{SQL}$ is $Before(a, b)$, where a and b are (instant) values in the tuple t. From the definition of the completion of a tuple, we must show

$$(\exists c_a \in \mathbf{C}(a),\ \exists c_b \in \mathbf{C}(b)\, (c_a \leq c_b)) \Rightarrow T \in Before_I(a, b, 1)$$

Applying the definition of $Before_I$, the right hand side is equivalent to $\mathbf{Pr}[a \leq b] > 0$ (here we use $\mathbf{Pr}'[\ldots] = 0.01$). From the left hand side, we know that $\mathbf{Pr}[a = c_a] > 0$ and $\mathbf{Pr}[b = c_b] > 0$, so $\mathbf{Pr}[a \geq b] > 0$. Hence,

$$\begin{aligned}
(\exists c_a \in \mathbf{C}(a),\ \exists c_b \in \mathbf{C}(b)\, (c_a \leq c_b)) &\Rightarrow \mathbf{Pr}[a = c_a] > 0 \wedge \mathbf{Pr}[b = c_b] > 0 \wedge c_a \leq c_b \\
&\Rightarrow \mathbf{Pr}[a \geq b] > 0 \\
&\Rightarrow T \in Before_I(a, b, 1) \\
&\Rightarrow T \in [\![\, P \,]\!]_{op}(1, t).
\end{aligned}$$

Induction: We utilize a similar induction hypothesis as before, and the inductive steps follow naturally. ∎

Theorem 4 $[\![\, S \,]\!]_{ind}(\delta, \gamma, r)$ *is monotonic in* γ.

Proof: As the semantics of each clause depends only on one parameter, the proof proceeds in two parts: (1) show that $[\![\, \langle where\ clause \rangle \,]\!]_{ind}$ is monotonic in γ, and (2) show that $[\![\, \langle target\ list \rangle \,]\!]_{ind}$ is monotonic in γ.

To show (1), we need to demonstrate that

$$\forall t\, \forall \langle predicate \rangle P\, (\gamma \geq \gamma' \Rightarrow \mathbf{C}([\![\, P \,]\!]_{op}(\gamma, t)) \leq \mathbf{C}([\![\, P \,]\!]_{op}(\gamma', t))).$$

We prove this by induction on the boolean connectives in p, after replacing p with its equivalent, as discussed in Section 5.4.2.
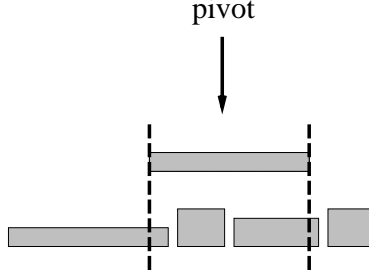
46

Figure 22: The rods within the dotted lines are the undercount for the pivot

Basis: $[\![\,P\,]\!]_{op}$ is *Before$_I$*(a, b, γ). This follows from the definition of *Before$_I$* and the monotonicity of **Pr**.

Induction: Assume that the above holds for P containing $i - 1$ boolean connectives.

   (a) P is P_1 `AND` P_2, which is replaced with $P_1 \cap P_2$, which is monotonic when P_1 and P_2 are monotonic.

   (b) P is `NOT` P_1, which is replaced with $\{x \mid (\neg x) \in [\![\,P\,]\!]_{op}(\gamma, r)\}$. Again, this is monotonic if P_1 is monotonic.

   To show (2), we observe that $[\![\,\langle\textit{target list}\rangle\,]\!]_{ind}$ utilizes constructors based on *Before$_I$*. By Theorem 3, $[\![\,]\!]_{ind}$ is an equivalent semantics, and hence from (1), just shown, (2) follows. ∎

Theorem 5: *The kth pivot will count $P^2/2^{\lfloor \mathbf{log}_2(k)\rfloor + 1}$ 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 $P^2/2^{\lfloor \mathbf{log}_2(k)\rfloor + 1}$ pairs. In general, the kth pivot will count $P^2/2^{\lfloor \mathbf{log}_2(k)\rfloor + 1}$ pairs. In the worst case, $2P$ pivots are required. ∎

Theorem 6: *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 22 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,

47

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%. ∎