# 4

# On the Semantics of “Now” in Databases

J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen,
and R. T. Snodgrass

While “*now*” is expressed in SQL as `CURRENT_TIMESTAMP` within queries, this value cannot be stored in the database. However, this notion of an ever-increasing current-time value has been reflected in some temporal data models by inclusion of database-resident variables, such as “*now*,” “*until-changed*,” “$\infty$,” “@” and “–.” Time variables are very desirable, but their use also leads to a new type of database, consisting of tuples with variables, termed a *variable database*.

This paper proposes a framework for defining the semantics of the variable databases of the relational and temporal relational data models. A framework is presented because several reasonable meanings may be given to databases that use some of the specific temporal variables that have appeared in the literature. Using the framework, the paper defines a useful semantics for such databases. Because situations occur where the existing time variables are inadequate, two new types of modeling entities that address these shortcomings, timestamps which we call *now-relative* and *now-relative indeterminate*, are introduced and defined within the framework. Moreover, the paper provides a foundation, using algebraic *bind* operators, for the querying of variable databases via existing query languages. This transition to variable databases presented here requires minimal change to the query processor. Finally, to underline the practical feasibility of variable databases, we show that database variables can be precisely specified and efficiently implemented in conventional query languages, such as SQL, and in temporal query languages, such as TSQL2.

## 1 Introduction

*Now* is a noun in the English language that means “at the present time” [48]. A variable with this name has also been used extensively in temporal relational data model proposals, primarily as a timestamp value associated with tuples or attribute values in temporal relations. Yet, the precise semantics of databases with this and other current-time variables have never been fully specified. An important goal of this paper is to give a clear semantics for databases with current-time variables.

Time variables such as *now* are of interest and indeed are quite useful in databases, including conventional SQL databases, that record time-varying information, the validity of which often depends on the current-time value. Such databases may be found in many application areas, such as banking, inventory management, and medical and personnel records. For example, in a banking application, it is necessary to record when account balances for customers are valid. Specifically, if a customer opens an account and deposits US$ 200 on January 15 (in some year), the validity of that balance starts when the deposit is made and extends until the current time, assuming no update transactions are committed. Thus, on January 16, the balance is valid from January 15 until January 16; on January 17, the balance is valid from January 15 until January 17, etc. It is impractical to update the database each day (or millisecond) to correctly reflect the valid time of the balance. Rather, it would be quite useful to be able to store a variable, such as *now*, to indicate that the time when a balance is valid depends on the current time. In the example, it would be recorded on January 15 that the customer’s balance of US$ 200 is valid from January 15 through *now*. While SQL–92 [38] has a construct `CURRENT_TIMESTAMP` (as well as `CURRENT_DATE` and `CURRENT_TIME`) for use in queries, one cannot store such a value in a column of an SQL table. All major commercial DBMSs have similar constructs, and impose this same restriction. The user is forced instead to store a specific time, which is cumbersome and inaccurate. This paper shows how database variables such as `CURRENT_TIMESTAMP` can be precisely specified and efficiently implemented in conventional query languages such as SQL-92 and in temporal query languages, while having little impact on the underlying data model.

We knew of no work on storing *now* in conventional databases, so we turned to the literature on temporal databases. In examining the large body of existing temporal data models, it is apparent that two different *types* of models have been proposed. The first type of model essentially accords with the view expressed by Reiter that a relational database can be seen as a set of ground first-order formulæ, for which there is a minimal model [41]. These models have either been presented as logical models directly (e.g., [14], [9]), or have been presented in such a way that their logical model was clear (e.g., [15]).

The second type of model deviates from this tradition. Rather, these models have been presented as a set of formulæ some of which are ground, but others of which have included one or more free, current-time variables. Chief among these current-time variables is “*now*” (e.g., [8, 25, 13]), but a variety of other symbols have been used, including “–” [4], “$\infty$” [44], “@” [37], and “*until-changed*” [52]. As already mentioned and exemplified, the use of such variables is quite convenient and practical. Thus, these approaches have advantages at the implementation level, namely, they are space efficient and avoid the need for updates at every moment in time. However, nowhere have we found a clear exposition of temporal variables, i.e., nowhere has the semantics of this type of database—a database with current-time variables, here termed a *variable database*—been formally specified so that the logical model represented by the database is clear. Rather, the models have relied on the choice of intuitive names for the variables to convey their meaning. This has led many to suppose that they understood their semantics. However, this reliance on intuition and lack of a clear semantics for databases with current-time variables is an unsatisfactory foundation for the development and implementation of variable databases, as it is prone to ambiguities and misinterpretations and, therefore, to errors.

In this paper, we present a framework for the specification of the different semantics that may be given to variable databases, which builds on the approach introduced in [12]. In the framework, the semantics of a variable database is defined by means of an *extensionalization mapping* from a variable database to a fully ground data model. The actual extensionalization mappings for valid-time, transaction-time and bitemporal databases with one or more current-time variables are given in subsequent sections. This illustrates that the framework is general enough to allow for the specification of a wide variety of semantics, an important property of a framework. It also illustrates that the framework can capture the semantics of multidimensional databases in a straightforward manner: the multidimensional extensionalization mapping is obtained by a simple, but coordinated, combination of the mappings for the constituent one-dimensional databases.

We also observe that the modeling capabilities of current-time variables are limited. To overcome these limitations, two new modeling entities, *now-relative* and *now-relative indeterminate* timestamps are introduced and defined within the framework. Next, a mechanism for the querying of variable databases using existing query languages is provided. This mechanism provides added functionality, does not require changes to a query language, and is easily integrated into a query processor. It is observed that the incorporation of the notion of perspective into query languages may provide additional functionality when querying variable databases. Finally, to underline the practicality of a variable database, compact physical representations for timestamps involving current-time variables are provided. These formats can be efficiently manipulated during query processing.

## 2 Motivation

To motivate the need for current-time variables in databases with time-varying data, including a solid, formal foundation for their use, this section introduces the use of such variables and explores some of the perhaps unintuitive, semantic subtleties resulting from their incorporation. Further, this section explores the limitations of current-time variables in some realistic situations.

As the meaning of current-time variables depends on whether the context is valid time or transaction time, current-time variables in valid-time and transaction-time databases are considered in isolation, followed by a short discussion of current-time variables in bitemporal databases.

### 2.1 Storing Valid-time Variables in Databases

The *valid time* of a fact denotes the time(s) when the fact is true in the modeled reality [33, 47, 32]. In the valid-time dimension, a timestamp involving *now* is commonly used to indicate that a fact is currently valid [2, 3, 22, 25, 40, 43, 49, 53].

It is possible to explicitly record when facts are valid in both conventional SQL databases and in truly temporal, e.g., TSQL2 [46], databases. With SQL databases, the semantics of valid time must be implemented in the application programs, while in temporal databases, the semantics are built directly into the data model and query language. The discussion of valid time that follows is phrased in terms of temporal databases, but applies equally well to conventional databases.

As an example, suppose that a database records that Jane was on the faculty of "State University" in some particular year, e.g., 1995; which year is not relevant here. Figure 1(a) shows the relevant tuple from the University's employment database (the **FACULTY** valid-time relation). Jane started working as an Assistant Professor on June 1, as indicated by the "from" attribute. The value *now*, appearing as the "to" time in Jane's employment tuple, represents the (later) time when Jane will stop working for State University as an Assistant Professor. Together, the "to" and "from" attributes encode the valid time associated with the tuple. For simplicity, we assume a timestamp granularity of one day in all examples.

The informal meaning of this tuple is that Jane is a faculty member from June 1 until the current time. Thus, the result of a query that requests the current faculty members will include Jane. As the current time inexorably advances, the value of *now* also changes to reflect the new current time. Some authors have called this concept "*until changed*" [52] or "@" [37] instead of "*now*," but the meaning is the same.

Using the variable *now* in a timestamp is very convenient. To see why, suppose that instead of using the variable as the "to" time, we use a ground time, i.e., a particular date. We start by recording a "to" time of June 1. Then as time advances

**FACULTY**

| | | | VALID TIME | |
|---|---|---|---|---|
| | NAME | RANK | (from) | (to) |
| **(a)** | *Jane* | *Assistant* | *June 1* | *now* |
| **(b)** | *Jane* | *Assistant* | *June 1* | *forever* |
| **(c)** | *Jane* | *Assistant* | *June 1* | *July 6* |
| | *Jane possibly employed as an Assistant* | | *July 6* | *now* |

Figure 1: Describing Jane’s employment

and Jane remains an Assistant Professor, the “to” time on Jane’s tuple must be updated each day to record when she worked. Hence, the “to” time would be updated to June 2, then to June 3, etc. While this representation is faithful to our knowledge at any point in time, having to continuously update the “to” time as time advances is impractical. It is also unclear who should do the updating, as the database has no indication of which tuples have a continuously increasing valid time and which are stable. For these reasons, it is better to use the variable *now*.

## 2.2 Anomalies of Existing Approaches

Here we explore four situations that illustrate shortcomings of a single variable *now* and thus indicate a need for additional current-time modeling entities, which we introduce in Section 4.

### The Pessimistic and Optimistic Assumptions

While using *now* is convenient, using it as the “to” time of a tuple may lead to an overly pessimistic assumption about the modeled reality. The university application introduced in the previous section provides such a situation. Specifically, it is reasonable to expect that if an employee is employed in a certain position today, that employee will also be employed in that position tomorrow (and the next few, following days). However, the **FACULTY** relation given in Figure 1(a) specifically records that Jane will *not* be employed tomorrow. Assume that today is July 9. Then a query asking who will be employed tomorrow (i.e., July 10) will not have Jane in the answer, since the “to” time of Jane’s tuple is *now*, or in this case, July 9. This is overly pessimistic.

Some temporal data models avoid this problem by limiting valid time to the past, that is, “to” times before *now* [25, 49]. For many applications, e.g., the university application, this limitation is much too restrictive. Other data models have advocated using one of the special (non-variable) valid-time values, such as *forever*, $\infty$, or "–" [44, 45, 4, 51]) instead of *now*. These symbols (we will use *forever*) denote the largest representable timestamp value, that is, the one furthest in the future. In SQL and in IBM's DB2, *forever* is about 8,000 years from the present [38, 16]; in our more liberal proposal, it is approximately 18 billion years from the present time [18].

By using a "to" time of *forever*, as in Figure 1(b), we certainly avoid the pessimistic assumption, but we are now being overly optimistic. We have indicated that Jane will be employed as an Assistant Professor not only tomorrow, but *forever*. To assert that Jane will be employed as an Assistant Professor forever is most assuredly incorrect (others have also noted that a "to" time of $\infty$, or *forever*, has erroneous implications for the future [40]). Another indication that *forever* is inappropriate is that when Jane departs the University, *forever* must be replaced by the date of her departure; but the revised date will be a separate and much earlier time that is inconsistent with *forever*. Rather than having the new information *refine* the old information, the new information contradicts the old information. Using, instead of *forever*, some large, application-dependent time value earlier than *forever* (e.g., in the university application, the mandatory retirement date) is better than the generic *forever*, but is still overly optimistic. In Section 4.4, we introduce a new type of timestamp that meets these requirements.

### The Punctuality Assumption

The use of *now* in timestamps implies a strong assumption about the punctuality of updates. For example, the tuple in Figure 1(a) states that Jane will remain an Assistant Professor until the current time. The correctness of this tuple is dependent on the correctness of the assumption that updates are made ahead of time, i.e., predictively. Thus, changes in Jane's employment status and rank are assumed to conform with the punctuality assumption: "changes are recorded in the database no later than the instant they take effect."

This assumption is not often satisfied. Rather, information is often recorded after the time it became valid, but with a well-specified maximum delay [31]. For example, when employees change status, it may be that the database is guaranteed to be updated to reflect this at most three days after the status is changed. If Jane was promoted on July 8, perhaps it is not until July 11 that her tuple is actually updated to reflect her correct status. With this delay, the database is known to correctly describe the mini-world only in the past, up until three days ago. Within the last three days, it can only be concluded that it is likely, or possible, that Jane is employed as an Assistant Professor. In this case, one could interpret the meaning of Jane's tuple in Figure 1(a) as of today (July 9) as shown in Figure 1(c) that intuitively illustrates the “possible” type of information that we would like to be able to record because it more accurately describes our knowledge of the mini-world. This cannot conveniently be recorded using *now*. Sections 4.2 through 4.4 describe a new kind of timestamp that can be used to address these issues.

### The Problem of *Now* in Predictive Updates

Another problem with using the variable *now* as a “to” time in a tuple occurs in predictive updates where the “from” time is after the current time. Thus, the “to” time is before the “from” time, contradicting the intuition that the “from” should always be before the “to” time. To illustrate this use of *now*, assume that the tuple in Figure 1(a) was inserted on May 25, i.e., the fact was recorded prior to when Jane began work. Then, during the remainder of May, the “to” time is before the “from” time.

Some data models do not allow the use of *now* as a “to” time when its value is before the “from” time. Instead a special “to” time value of NULL is used in such situations [22, 40, 53]. This value is replaced by *now* when the value of *now* exceeds the “from” time. Tuples with NULL’s are ignored in queries. However, there is a subtle difficulty with this solution. Suppose that today is May 25 and we record that Jane will be an Assistant Professor from June 1 until *now* (or NULL in this case). We then execute a query that determines who will be employed in June barring any changes to the database between now and June. To evaluate this query, we temporarily “observe” the database from the perspective of a user in June even though today is May 25. The result should include Jane; however, Jane’s tuple is ignored since it has a “to” time of NULL. In Section 4.4 we introduce a new modeling entity that addresses this shortcoming.

### Queries and *Now*

When querying data that involves *now*, the current time must be clearly specified since the value of *now* depends on this time. To illustrate the kind of ambiguity that can result from unclear specification of the current time, assume that today is July 9 and that our database is given as in Figure 1(a). Then, consider the query, “Will we agree on July 13 that Jane was employed on July 11?” Suppose that *now* is interpreted to refer to the time at which the query is asked, in this case July 9. Then Jane will not be employed on July 11 and so we would answer “no.” But *now* could be interpreted as the time mentioned in the query about which we were asked to agree, in this case July 13. Then Jane will be employed on July 11 and so we would answer “yes.”

Another source of ambiguity is that the constant evolution of the current-time variable *now* appears to cause the “same” query to return different results when evaluated at different times, even if no updates have occurred. For instance, consider the query, “Is Jane employed on July 11?” This simple query asked on July 10 will yield one answer (“no”), but if we ask the query on July 12 we will receive a quite different answer (“yes”). Hence, the querying of variable databases introduces new semantic subtleties, not found when querying non-variable databases.

## 2.3 Storing Transaction-time Variables in Databases

The *transaction time* of a database fact denotes the time(s) when the fact is (logically) current in the database [47]. It is an orthogonal concept to valid time, in that it concerns the evolution of the database, as opposed to the enterprise being modeled. The use of current-time relative variables in transaction-time databases introduces a different set of problems.

While a valid-time timestamp is generally supplied by the user, a transaction-time timestamp, an interval from a “start” to a “stop” time, is supplied automatically by the DBMS during updates. Insertions initialize the “start” time to the “current time” and the “stop” time to *now*[1]. Deletions are accommodated by changing “stop” times of *now* to the “current time.” Hence, deletion is logical. The information is not physically removed from the relation; rather, it is tagged as no longer current by having a “stop” time different from *now*. Updates may be considered combinations of deletions and insertions.

As an example, consider the transaction-time relation in Figure 2(a). The distinct semantics of transaction time yields a different interpretation of this relation as compared with the one shown in Figure 1(a). The “start” time of June 1 indicates that this tuple was stored in the database on June 1, i.e., we first became aware that Jane was an Assistant Professor on that date. The value of *now* for the “stop” attribute indicates that the database still records that Jane is an Assistant Professor, i.e., the fact is current. If we learn on July 10 that Jane left State University and thus (logically) delete the fact, this is reflected by changing the “stop” time to July 10.

| | **FACULTY** | | | |
|---|---|---|---|---|
| | | | TRANS TIME | |
| | NAME | RANK | (start) | (stop) |
| **(a)** | *Jane* | *Assistant* | *June 1* | *now* |
| **(b)** | *Jane* | *Assistant* | *June 1* | *forever* |

Figure 2: Describing Jane’s employment in a transaction-time relation

---

[1] The transaction processing system must also obey the requirement that the “start” times of tuples be consistent with a serialization order of their respective transactions.

The problem with using a variable called *now* in transaction time is that the name “now” obscures the use of the variable. Strictly speaking, it implies that every current tuple was deleted by the current transaction! In Figure 2(a), if the current time is July 9, then a strict interpretation of a “stop” time of *now* suggests that the “stop” time is July 9. This is not what was intended.

As with valid time, some data models address this problem by using *forever* (also called $\infty$ or “–”) instead of *now*, as shown in Figure 2(b) [4, 5, 44, 51]. Using this large value, we immediately encounter difficulties. The strict interpretation of this tuple is that some transaction executing a (very) long time in the future will logically delete this tuple from the relation. In the meantime, it will remain in the database. If, on July 10, it becomes known that Jane has left State University, then we logically delete this tuple by changing the “stop” time to July 10. Such a change is inconsistent with the previous “stop” time. Put differently, in this scenario the database first records that we believe that Jane is an Assistant Professor from June 1 until “forever.” The subsequent update then contradicts this belief by saying that it is only from June 1 until July 10 that we believe Jane is an Assistant Professor.

There is a more fundamental problem with *forever* in transaction time. By the semantics of transaction time, storing future transaction times is equivalent to predicting future states of the database, which is a highly problematic proposition. With no crystal ball at hand, it is customary to avoid predictions and require that the right endpoint of every interval be less than or equal to the current time. To distinguish “now” in transaction time from “now” in valid-time, we propose in this paper to adopt the name “*until changed*” for the former and provide a precise semantics for its interpretation.

### 2.4 Variables in Bitemporal Databases

Bitemporal databases support both valid time and transaction time [33]. The confusion that has arisen in a number of bitemporal data models between the use of the same variable in both dimensions was a prime motivation for the semantic framework which we present below. In order to allow for a completely general treatment of the semantics of these variables, we use a different variable in each dimension. In Section 6 we show how the concept of a *reference time* can coordinate the interaction between the current-time variables in both time dimensions.

## 3 Semantic Framework

In order to provide a precise semantics for databases with current-time variables, we propose a semantic framework for defining the meaning of databases with variables in terms of databases of a fully extensional temporal data model. Databases in this latter model are fully ground, i.e., they do not admit variables. While the model is not suitable for the implementation of temporal databases, it is well suited for capturing the semantics of variable databases.

## 3.1 The Temporal Universe

The framework developed below includes three distinct time dimensions, each with its own temporal universe. The framework requires the existence of well-defined mappings between these universes. Although this requirement does not preclude the possibility of different granularities for the universes, we choose to avoid such diversions and instead use a single, underlying granularity. This yields a homogeneous treatment of all time dimensions and their relationships.

Since most database researchers have adopted the view that valid time in a database is best viewed as discrete, and every database transaction model that we are aware of has this property, we adopt a discrete model of time. Let $\mathcal{T}_Z$ to be the totally ordered set $\{\ldots, -2, -1, 0, +1, +2, \ldots\} \cup \{\bot, \top\}$, where $\bot$ (*bottom*) and $\top$ (*top*) are two distinguished elements, which intuitively correspond to $-\infty$ and $\infty$, respectively. The total order $<_{\mathcal{T}_Z}$ on $\mathcal{T}_Z$ is the normal order on integers extended so that $\bot$ and and $\top$ are a bottom and a top element, respectively, i.e.,

1. for any two integers $z$ and $z'$, $z <_{\mathcal{T}_Z} z'$ if $z < z'$ *(as integers);* and
2. $\forall t \in \mathcal{T}_Z \ (\bot \leq_{\mathcal{T}_Z} \ t \ \leq_{\mathcal{T}_Z} \top)$.

The only requirement on our temporal universe, $\mathcal{T}$, is that it has the same order structure as $\mathcal{T}_Z$. That is, $\mathcal{T}$ can be any ordered set of order type $1 + \ {}^*\omega + \omega + 1$ ([24], p. 128). For example, in most of the examples in this article we chose $\mathcal{T}$ to be the ordered set of days, extended infinitely into the past and the future, with added elements $-\infty$ and $\infty$.

In addition to the concepts of valid time and transaction time, we introduce a third time, *reference time,* to represent the relationship between a temporal database and the "real world" time at which it is viewed. Thus, three temporal universes are required in the framework, namely the *reference-time*, the *valid-time*, and the *transaction-time universe*, and it may be desirable or convenient to restrict them to some subset of $\mathcal{T}$. Therefore, let

- $\mathcal{T}_{RT} \subseteq \mathcal{T}$ denote the *reference-time universe* of our database,
- $\mathcal{T}_{VT} \subseteq \mathcal{T}$ denote the *valid-time universe* of our database, and
- $\mathcal{T}_{TT} \subseteq \mathcal{T}$ denote its *transaction-time universe.*

## 3.2 Important Times

Throughout our discussion of variable databases and queries on these databases, five distinct times surface repeatedly. The first of these is called *initiation*. It is relative to a specific relation and denotes the transaction time when that relation was created. To simplify the discussion that follows, we assume that all relations are created at the same time, denoted by $t_0$. Once created, we assume that the database schema never changes (schema versioning [42] is orthogonal to most of the issues discussed in this paper).

The second important time, which is new to most readers, is the *reference time*. The reference time is the time of the database observer’s “frame of reference,” denoted by $rt_*$. Reference time is a term analogous to the *indices* or “points of reference” in intensional logic [39], and discussed more recently in the context of valid-time databases [23]. The reference time facilitates a kind of “time travel” by means of which we may observe the database at times other than the present.

A related time is the *query time*, or *current transaction time*, denoted by $t_{current}$. It is the time at which a query is processed. The reference time, $rt_*$, and current time, $t_{current}$, are related, but distinct. In general, $t_{current}$ is the time at which a query is initiated, while $rt_*$ is the time at which the user “observes” the database. In many queries, the reference time and the query time are the same. But the user may choose to observe the database from a previous perspective; for this kind of query, the reference time is earlier than the query time. For example, if today is July 9 and we wish to observe the database from the perspective of a week ago, then $t_{current} =$ July 9, and $rt_* =$ July 2.

The final two times of special interest are the *valid timeslice time*, $vt_*$, and the *transaction timeslice time*, $tt_*$. These times are important in this paper because, for expository purposes, we focus exclusively on various timeslice queries. The valid and transaction timeslice times could both be an instant, an interval, or a set of instants or intervals. The valid timeslice time(s) specifies the real-world time about which information is wanted, while the transaction timeslice time(s) is the time(s) during which information must be current in the database in order to be of interest for a query. For the example queries given in this paper, it is advantageous to choose instants (as opposed to intervals) as the valid timeslice and transaction timeslice times. Later, we shall see that, while these times are distinct concepts, there are important relationships between the valid timeslice time, the transaction timeslice time, and the reference time.

To illustrate the distinction among these five times, let us consider an example. A temporal database for recording employment information is created on January 11 (again, the particular year is immaterial). Today (which we assume is July 9), the director of the personnel department investigates an apparent discrepancy reported by a co-worker a week earlier, while using the database on July 2. The co-worker discovered that the database had mistakenly recorded on June 27 that an employee had been hired two weeks earlier, on June 13. The five times in this example are as follows.

1. $t_0$ is January 11, the day of the creation of the database;

2. $rt_*$ is July 2, the day when the problem was observed;
3. $t_{current}$ is July 9, the day the personnel department director investigates the database;
4. $vt_*$ is June 13, the real-world day of the problematic information; and
5. $tt_*$ is June 27; this is the day for which we are interested in what was recorded as current information in the database.

By using a reference time of July 2, the director can view the identical database state in existence when the co-worker discovered the discrepancy. If a reference time of June 20 had been used instead, it is possible that no discrepancy would have been found, because that date was well before $tt_*$. Although purposely contrived, this example highlights the differences among the five times. Having made this point, this example will not be used in the remainder of this article.

We have the following constraints on these five times.

- $\bot \leq t_0 \leq tt_* \leq t_{current} \leq \top$
- $\bot \leq rt_* \leq \top$
- $\bot \leq vt_* \leq \top$

Note that $rt_*$ is not bound by $t_{current}$. This provides the ability to ask "hypothetical now" queries, that is, from the perspective of a future valid time (i.e., ten years from now). Such an example is given later in Section 6.2.

### 3.3 Extensional and Variable Database Levels

It is useful to view the semantics of temporal databases with variables within the context of a two-level framework. This section develops such a framework in two steps, by first presenting the levels of a theoretical framework. Then this framework is augmented, motivated by the practical concerns of easily extending existing data models to admit databases with variables, such as *now*, with minimal impact on existing query languages and query processing engines.

A relational database consists of a set of relations, where each relation is a set of tuples. Each tuple in a relation has a number of application-specific attribute values. Temporal databases extend this view by incorporating the temporal aspects of data using special attributes, termed timestamps. These are explored further next.

In our model of time from the previous section, *time instants* (or just *instants* for short) are points in time and *intervals* are sequences of temporally consecutive points. (Indeed, when time is discrete, intervals are merely shorthand for a finite, or countably infinite, set of instants.) Intervals may be uniquely described by two bounding instants, termed the *starting* and *terminating* instants.

Each of the valid and transaction times of data may be recorded by associating a single time interval or a single instant with each tuple. Interval timestamps are very convenient at the conceptual and implementation levels, as they are compact and can represent information about a potentially large number of times in a single tuple. Thus, following a range of temporal data models, we will assume interval timestamps at the variable database level.

We employ specific names for the timestamp attributes that encode the time intervals of tuples. For valid-time intervals, the starting instant is recorded by an attribute “from” and the terminating instant is recorded by an attribute “to”; see Figure 1 for an example. For transaction-time intervals, we use “start” and “stop,” as in Figure 2. In a variable database, the values of the timestamp attributes in any tuple are extended to permit instances of one or more current-time variables, as discussed earlier. Figure 1(a) gives a simple variable database with only one tuple.

Moving to the extensional level, tuples also have timestamp attributes. However, there are three key differences. First, no variables are allowed—the extensional level is fully ground. Second, timestamps are instants rather than intervals. Third, an extensional tuple has one additional temporal attribute, called a *reference time attribute*. Later in this paper we describe the importance of reference time to the meaning of tuples. For now, it may be thought of as representing the time at which a meaning was given to the temporal variables in the original tuple.

Whereas the variable-database level offers a convenient representation that end-users can understand and that is amenable to implementation, the mathematical simplicity of the extensional level supports a rigorous treatment of temporal databases in terms of first order logic. A theoretical framework for providing a logical interpretation or “meaning” for a particular variable database, i.e., a “translation” from variable to extensional level, may be based on a homomorphic mapping from variable-level databases to extensional-level databases [12]. This mapping is termed an extensionalization, and is denoted $[\![\ ]\!]$. In addition to giving the semantics of variable databases, the framework also provides a means for checking the correctness of query languages over variable databases. This is illustrated in Figure 3 and explained using an example.

The top of the figure, labeled the *variable database level*, represents a database model that allows the use of temporal variables in timestamps of tuples. At the top left, we see a particular variable database, $db$. The tuple <*Jane*, *Assistant*, [*June 1*, *now*]> (with *now* being a variable) from Figure 1(a) is an example. A query $q^V$ is applied to this database, resulting in another variable database, $q^V(db)$. Let $q^V$ be “List the faculty on June 15,” and assume this query is evaluated on June 27. The result is then {<*Jane*, *Assistant*>}.

The bottom of the figure, labeled the *extensional database level*, represents our fully extensional temporal data model, whose semantics is well-specified in the standard tradition of a first-order logical framework. Developing a query language in this extensional model is relatively straightforward, due to the model’s simplicity. In contrast, developing a query language for a more complex variable-level data

$$\begin{tikzcd}
\textit{variable database level} & db \arrow[r, "q^V"] \arrow[dd, "[\![\ ]\!]"'] & q^V(db) \arrow[dd, "[\![\ ]\!]"] \\
\textit{extensionalization mapping} & & \\
\textit{extensional database level} & [\![db]\!] \arrow[r, "q^E"'] & q^E([\![db]\!]) \;=\; [\![q^V(db)]\!]
\end{tikzcd}$$

Figure 3: Relationship between variable database and extensional database

model is error prone. The framework can be used for checking the correctness of variable-level query constructs. Specifically, variable-level query constructs must commute with the corresponding extensional-level query constructs, as indicated in the figure: $q^E([\![db]\!]) = [\![q^V(db)]\!]$.

A particular extensionalization mapping from the top level to the bottom level is defined in order to specify the semantics of variable databases. As tuples at the variable database level are independent of each other, an extensionalization mapping may treat each tuple in isolation.

Continuing the example, an extensionalization mapping[2] may map <*Jane*, *Assistant*, [*June 1*, *now*]> of $db$ to {<*Jane*, *Assistant*, *June 1*, *June 27*>, <*Jane*, *Assistant*, *June 2*, *June 27*>, ... , <*Jane*, *Assistant*, *June 27*, *June 27*>} of $[\![db]\!]$. In the extensional database level, valid-time tuples are associated with two (instant) timestamps. The first timestamp, e.g., *June 1*, is the instant when the fact was valid (the interval is deconstructed into its component instants), and the second time, e.g., *June 27*, is the reference time. The extensional-level version of the query then selects the tuple from this set that has a valid time of June 15, giving as result $q^E([\![db]\!]) =$ {<*Jane*, *Assistant*, *June 27*>} (again omitting the valid time). Finally, applying the extensionalization mapping to the variable-level query result, {<*Jane*, *Assistant*>}, yields $[\![q^V(db)]\!] =$ {<*Jane Assistant*, *June 27*>}. The diagram thus commutes for the sample database and query. Section 4 and subsequent sections provide a thorough coverage of extensionalization.

We are concerned in this paper with the practical use of variable databases. In particular, we are interested in how to extend existing data models and query

---

[2] Here, we consider only a reference time of June 27. In the discussion to follow, $[\![\ ]\!]$ takes an optional subscript. We omit these subscripts here to simplify the discussion.

languages with the ability to allow current-time variables, with as little impact as possible on their conceptual model and their associated query processing engines. This is consistent with the philosophy of the designers of the proposed temporal extension to SQL-92, termed TSQL2 [46]. Thus, we next augment the theoretical framework as shown in Figure 4.

$$\begin{tikzcd}
\text{variable database level} & db \arrow[dd, "[\![\ ]\!]"'] \arrow[dr, "bind^V"] & & \\
& & bind^V(db) \arrow[r, "q^V"] \arrow[d, "[\![\ ]\!]"'] & q^V(bind^V(db)) \arrow[d, "[\![\ ]\!]"'] \\
\text{extensional database level} & [\![db]\!] \arrow[r, "bind^E"] & bind^E([\![db]\!]) \arrow[r, "q^E"] & q^E(bind^E([\![db]\!])) = [\![q^V(bind^V(db))]\!] \\
& & = & \\
& & [\![bind^V(db)]\!] &
\end{tikzcd}$$

Figure 4: Preprocessing of variable-level databases

Figure 4 shows that at the variable database level, the database is mapped to an intermediate stage, in which the tuples contain timestamps but no variables, by applying a $bind^V$ operation, which is discussed further below. The various existing temporal data models, including SQL-92 and TSQL2, that do not permit variable timestamps in their databases belong to this stage. By mapping variable-level databases to this intermediate stage, it is possible to reuse existing—or proposed—query engines to query variable databases. This is the motivation for augmenting the framework to permit preprocessing of variable databases before querying them.

The preprocessor substitutes each instance of a variable with a specified time, effectively “binding" the variables in a variable database (as discussed in Section 7, this occurs during query evaluation on a per-tuple basis). The bind operator, $bind^V$, maps a database from the variable to the intermediate level, upon which it is queried with a variable-level query $q^V$. The correctness of this mechanism is ensured by providing extensional-level counterparts to the preprocessor and to the queries, $bind^E$ and $q^E$, respectively, and by demonstrating that the above diagram commutes.

To exemplify, as before let the database $db$ contain the single tuple in Figure 1(a), let the query $q$ be “List the faculty on June 15,” and let the current date be June 27. To evaluate $q$ on $db$, we first bind the variable *now* to the reference time, June 27. The result is $bind^V(db) = \{$<*Jane*, *Assistant*, [*June 1*, *June 27*]>$\}$. The query can then be evaluated on this relation using simple (and already accepted) methods, resulting in $q^V(bind^V(db)) = \{<Jane, Assistant>\}$.

By defining queries $q^V$ at the variable-level in this way, as a composition of a binding operator and a ground query, we conceive a framework where the commutativity of the diagram shown in Figure 3 holds. While the particular binding we exhibit here is simple, Section 2 showed this need not be the case for all temporal databases, particularly not for bitemporal databases. For this reason, we need a precise semantics, provided by the extensionalization mapping and the extensional counterpart to the query operators, and a correctness criterion, that is, the commutativity requirement.

## 4 Valid-Time Databases

Here we present a semantics for variable valid-time databases by specifying mappings from the variable to the extensional level. We initially consider the extensionalization mappings for databases with ground timestamps and timestamps with the variable *now*. In order to address the shortcomings identified in Section 2, we also introduce additional, current-time modeling entities. Specifically, we consider *now-relative* timestamps that allow for positive and negative displacements from *now*. Next, we introduce so-called indeterminate time values which may be used in timestamps to indicate imprecise times. This leads to a further generalization of now-relative instants to *now-relative indeterminate* instants, which are values that are imprecise as well as current-time relative. The section concludes with an illustration of the querying of variable databases.

### 4.1 Extensionalization of Valid-time Tuples with *Now*

We first consider the extensionalization of tuples with ground timestamps. To do this, it is convenient to start by defining the meaning, or denotation, of the ground component in a timestamp. As other timestamp values are introduced, their denotations will also be defined.

**Definition 1 (Denotation of a Time Instant)** The *denotation* of a valid-time instant $t$ at a particular reference time $rt_*$, written $\langle\!\langle t \rangle\!\rangle_{rt_*}$, is defined as follows.

$$\langle\!\langle t \rangle\!\rangle_{rt_*} =_{df} t. \qquad \square$$

In general, to map a *ground* valid-time tuple, i.e., a tuple without variables, to the extensional database level, the tuple is *expanded* into a set of tuples, one for each time instant in its associated timestamp. Let us consider first the extensionalization of a ground tuple at a particular reference time. We use the notation, $[\![T]\!]_{rt_*}$, to denote the extensionalization of tuple $T$ at a reference time of $rt_*$.

**Definition 2 (Extensionalization of a tuple at an Instant)** The *extensionalization* of a ground tuple $T$ of the form $T =< X, [vt_1, vt_2] >$, where $[vt_1, vt_2]$ denotes the set of times $\{vt \mid vt_1 \leq vt \wedge vt \leq vt_2\}$, at reference time $rt_*$ is defined as follows.

$$[\![T]\!]_{rt_*} =_{df} \{(X, vt, rt_*) \mid vt \in [\langle\!\langle vt_1 \rangle\!\rangle_{rt_*}, \langle\!\langle vt_2 \rangle\!\rangle_{rt_*}]\}. \qquad \square$$

Note that each tuple at the extensional level is tagged with the reference time.

To exemplify, assume that the academic career of Jane at State University is given by the tuple $T$ =<*Jane, Assistant,* [*June 3, June 9*]>. The extensionalization mapping of this tuple at time June 6, i.e., $[\![T]\!]_{June\ 6}$, consists of seven tuples: {<*Jane, Assistant, June 3, June 6*>, <*Jane, Assistant, June 4, June 6*>, . . . , <*Jane, Assistant, June 9, June 6*>}. Recall also the sample mapping given in Section 3.3.

**Definition 3 (Extensionalization of a tuple at an Interval)** In the extensionalization mapping, a reference time interval may be used rather than a single reference time. The extensionalization of the tuple $T$ over the reference time interval $[rt_1, rt_2]$ is defined as follows.

$$[\![T]\!]_{[rt_1,rt_2]} =_{df} \bigcup_{rt_* \in [rt_1,rt_2]} [\![T]\!]_{rt_*}. \qquad \square$$

**Definition 4 (Extensionalization (Complete))** The complete meaning or extensionalization of a tuple $T$, denoted $[\![T]\!]$, is simply the extensionalization of $T$ over all reference times, i.e., $\bigcup_{rt_* \in \mathcal{T}_{RT}} [\![T]\!]_{rt_*}$. Equivalently, the general meaning or extensionalization of a tuple $T$ is:

$$[\![T]\!] =_{df} [\![T]\!]_{[\bot,\top]} . \qquad \square$$

We have found that a two-dimensional graphical notation makes valid-time concepts easier to grasp. In the visualization, reference time corresponds to the X-axis and valid time corresponds to the Y-axis. The graphical representation is a plot of the tuple at the extensional database level. Each cell in the plot stands for a particular reference time, $RT$, and valid time, $VT$, combination. The cells corresponding to the temporal coordinates of tuples in the extensional set of tuples are shaded, indicating when a tuple is valid relative to the reference time of an observer. Even though our underlying model of time is discrete, we treat each cell as a region rather than a point since this results in a better visualization. Several tuples may be plotted in the same graph by using different cell colors or patterns. The key, shown below the graph, indicates the explicit attribute values of the corresponding tuples. Variations of these graphs have been independently explored [34, 30, 12].

As an example, Figure 5(a) shows the extensionalization of Jane’s employment tuple from before for a sequence of reference times, June 1 through June 11, that is, $[\![T]\!]_{[\textit{June 1,June 11}]}$ . The figure illustrates that the valid time of this tuple is *reference-time invariant*, that is, it is independent of the reference time. So for a tuple with a valid-time interval but without variables, it does not matter at what time the tuple is observed—it is always valid over exactly the same interval.

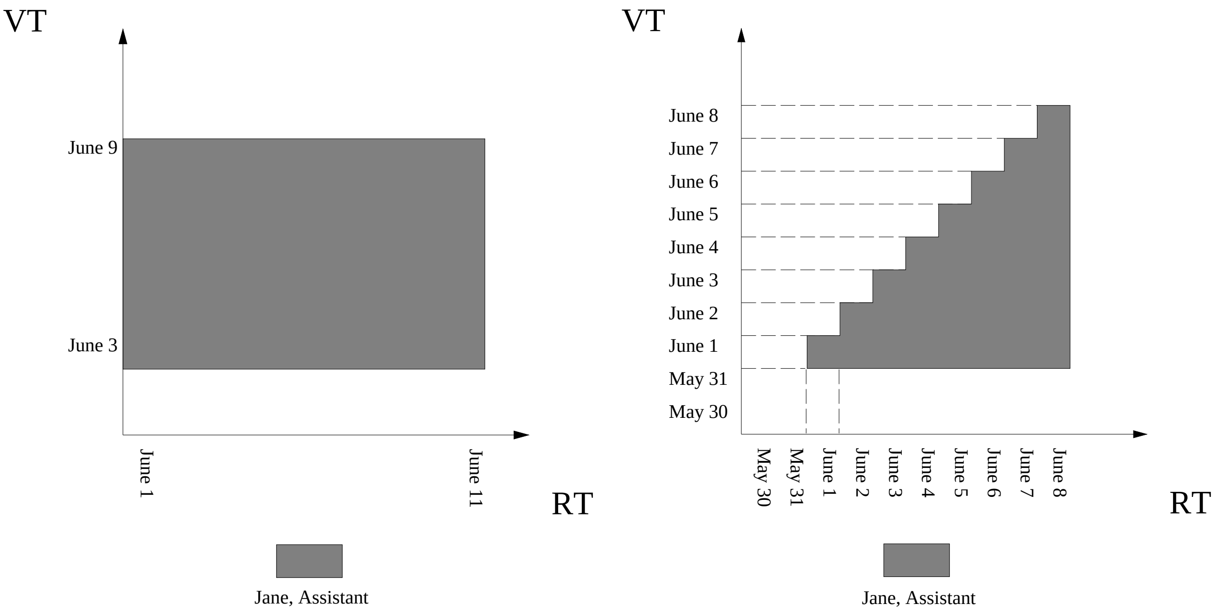
The meaning of a tuple with the variable *now*, however, is not reference-time invariant. The denotation of *now* makes this dependence explicit.

**Definition 5 (Denotation of** *now***)** The *denotation* of the current-time variable *now* at a particular reference time $rt_*$ is defined as follows.

$$\langle\!\langle now \rangle\!\rangle_{rt_*} =_{df} rt_* \qquad \Box$$

This is precisely how reference time enables us to ‘materialize’ variables in the extensional level. While variables *per se* are not permitted at the extensional level, a valid-time tuple does vary with reference time. With this additional timestamp value, the extensionalization of a tuple with *now* as the “to” or “from” time is still given by Definition 2.

As an example, assume that the academic career of Jane at State University is given by the tuple $T =< Jane, Assistant, [June\ 1, now] >$. Figure 5(b) visualizes the extensionalization of this tuple for every reference time between May 30 and June 8. Note that before June 1 the *empty interval* is depicted in the figure. This is because a timestamp with a “to” time that is before the “from” time denotes the empty interval. This situation occurs prior to June 1. The valid-time region in the figure is “stair-shaped” since the extensionalization of a tuple with variables is dependent on the time at which we observe the tuple. The stair-shape is a result of the constraint that the “to” time in the valid-time interval is bound to the reference time.



(a) A ground valid-time tuple (b) A valid-time tuple with a variable

Figure 5: A graphical representation of the extensionalization of a valid-time tuple

It is our contention that all other valid-time current-time variables currently in use (e.g., “@” [37] and *until-changed* [52]) have the same meaning as *now*. Thus having covered existing variables, we now proceed by proposing new timestamps that address the shortcomings of *now* discussed in Section 2.

### 4.2 Now-relative Instants

In this section we introduce a new type of timestamp, called a *now-relative instant*, that adds flexibility to the variable *now*. A now-relative instant generalizes the variable *now* by allowing an offset from this variable to be specified. Now-relative times were first introduced in transaction time for vacuuming [28].

With now-relative instants, we have a means of more accurately recording our knowledge of Jane’s employment with State University. For example, it may be that changes in hirings at State University are recorded in the database only three days after they take effect. Assuming that Jane was hired on June 1, we can accurately record our definite knowledge of her employment in the tuple <*Jane*, *Assistant*, [*June 1*, *now* − 3 *days*]>. This tuple states that Jane was an Assistant Professor from June 1 and until three days ago, but it contains no information about her employment as of, e.g., yesterday.

A now-relative instant thus includes a displacement, which is a (signed) span, from *now*. In the example given above, the displacement is minus three days. The extensionalization of tuples with now-relative instants is formalized as follows.

**Definition 6 (Denotation of a Now-relative Instant)** The *denotation* of a now-relative instant, *now* OP $n$ days, where OP $\in$ $\{+, -\}$, at a particular reference time $rt_*$ is defined as follows.

$$\langle\!\langle \mathit{now}\ \mathrm{OP}\ n\ \text{days}\ \rangle\!\rangle_{rt_*} =_{df} \langle\!\langle \mathit{now} \rangle\!\rangle_{rt_*}\ \mathrm{OP}\ n \qquad \square$$

Even with this additional timestamp value, the extensionalization of a valid-time tuple is still given by Definition 2.

Although now-relative instants allow us to relax the otherwise close coupling between valid and transaction time found in the punctuality assumption, now-relative instants still suffer from making a pessimistic assumption. The use of *now* − 3 days in the first example is an ultra-pessimistic view of the future. Jane would not even be employed *now* since her employment terminates three days prior to *now*. To address this potential shortcoming, we next introduce the notion of indeterminate timestamp values.

### 4.3 Indeterminate Timestamp Values

It turns out that support for valid-time indeterminacy [6, 17, 27, 35] can also alleviate the shortcomings of *now* and now-relative instants. This section introduces indeterminate timestamp values for ground timestamps. The next section extends this treatment to indeterminate timestamps with variables.

Sometimes, the time when an event occurred is known only imprecisely. For instance, we may know that an event happened "sometime in June 1993," which is an imprecise period of 30 days. An *indeterminate instant* is the time of an event, which is known to have occurred, but exactly when is unknown [19, 21].

The times when the event might have occurred is called the *period of indeterminacy* and is delimited by a lower and an upper bound (e.g., the event occurred sometime between June 1 and June 30). An indeterminate instant could have an associated probability distribution that gives the probability that the event occurred for each time in the period of indeterminacy. For the purposes of this paper, we ignore the probability information: every indeterminate instant is treated as though it has a distribution that is *missing* [21]. A *determinate* instant may be thought of as an indeterminate instant, with identical lower and upper bounds. An *indeterminate interval* is an interval bounded by indeterminate instants.

By using indeterminate instants, we can more accurately record our knowledge of Jane's employment with State University. Instead of using *now* as the "to" time in Jane's tuple, we can use an indeterminate instant. Which indeterminate instant to use depends on our knowledge of the situation. If Jane was hired to work *at least* two months, we could record this information as shown in Figure 6(a). Here two time bounds, July 31 and *forever*, delimit the "to" indeterminate instant. If State University has a mandatory retirement policy, we could decrease the indeterminacy considerably, as shown in Figure 6(b).

| | **FACULTY** | | | |
|---|---|---|---|---|
| | | | VALID TIME | |
| | NAME | RANK | (from) | (to) |
| **(a)** | *Jane* | *Assistant* | *June 1* | *July 31 ∼ forever* |
| **(b)** | *Jane* | *Assistant* | *June 1* | *July 31 ∼ January 1, 2028* |

Figure 6: Using indeterminate timestamps for recording Jane's appointment

Indeterminate instants address the pessimistic update assumption, providing evidence that Jane might still be employed in the future. They also remove the problem of incompleteness in the non-timestamp attributes (e.g., *possibly* employed, as shown in Figure 1(c)), and ensure that new knowledge acquired later, such as the information that Jane left the company on August 10, is not inconsistent with currently stored information, but rather is a refinement of that information. They also address the problem of *now* in predictive updates; an indeterminate interval is a valid interval no matter when it was stored in the database.

There are two bounds on the information represented by an indeterminate interval [36]. The first bound is the *definite* information. The definite information represents all that is definitely known about the interval and is the intersection of all of the possible intervals. The second bound is the *possible* information. The possible information represents the maximum possible extent of an interval and is the union of all of the possible intervals. The two bounds have different extensionalizations. The definite information is given by the *definite extensionalization*, presented next.

**Definition 7 (Indeterminate Ground Tuple)** An *indeterminate ground tuple* is a ground tuple of the form $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$, where $vt_1 \leq vt_2$ and $vt_3 \leq vt_4$. Here, $vt_1$ and $vt_2$ are the lower and upper bound, respectively, of the starting instant and $vt_3$ and $vt_4$, are the lower and upper bound, respectively, of the terminating instant. □

**Definition 8 (Definite Extensionalization of an Indeterminate Tuple)** The definite extensionalization of an indeterminate ground tuple of the form $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$, at the reference time $rt_*$ is defined as follows.

$$[\![T]\!]^D_{rt_*} =_{df} \{(X, vt, rt_*) \mid vt \in [\langle\!\langle vt_2 \rangle\!\rangle_{rt_*}, \langle\!\langle vt_3 \rangle\!\rangle_{rt_*}]\} \qquad \square$$

The possible information is given by the *possible extensionalization*.

**Definition 9 (Possible Extensionalization of an Indeterminate Tuple)** The possible extensionalization of a ground indeterminate tuple of the form $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$ at the reference time $rt_*$ is defined as follows.

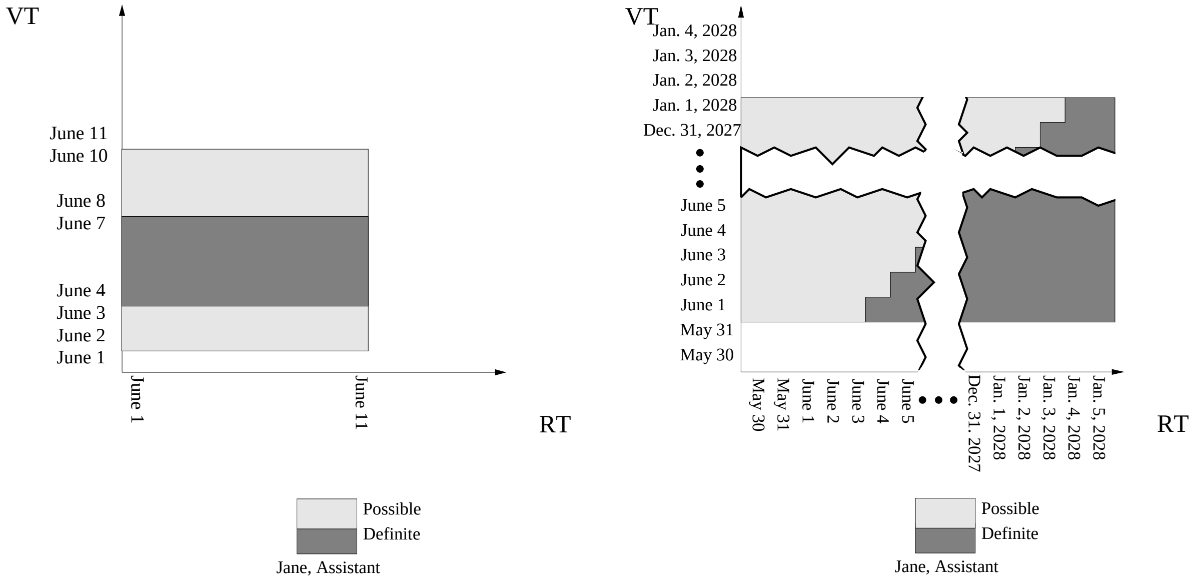
$$[\![T]\!]^P_{rt_*} =_{df} \{(X, vt, rt_*) \mid vt \in [\langle\!\langle vt_1 \rangle\!\rangle_{rt_*}, \langle\!\langle vt_4 \rangle\!\rangle_{rt_*}]\} \qquad \square$$

It is always the case that the definite information is a subset of the possible information. Note that if the bounding instants are determinate, that is, if the lower and upper bounds are the same, then the possible and definite extensionalizations yield exactly the same set of tuples. Consequently, for the extensionalization of determinate intervals, we omit the possible or definite superscript and use $[\![\ ]\!]_{rt_*}$ instead of either $[\![\ ]\!]^P_{rt_*}$ or $[\![\ ]\!]^D_{rt_*}$.

Valid-time tuples timestamped with indeterminate intervals have a graphical representation similar to the one described above. Both the possible and definite extensionalizations are represented. We use different shadings to distinguish the regions in the two extensionalizations. As an example, assume that the academic career of Jane at State University is given by the tuple

$$< \mathit{Jane}, \mathit{Assistant}, [\mathit{June\ 1} \sim \mathit{June\ 3}, \mathit{June\ 7} \sim \mathit{June\ 10}] > .$$

Jane's academic career, for the reference times [*June 1*, *June 11*], is graphically represented in Figure 7(a). Note that the region of possible information is never smaller than the region of definite information and that the valid time is reference-time invariant (just as it is for determinate intervals) when the tuple has no variables.



(a) A ground indeterminate valid-time tuple

(b) Jane's possible and definite employment

Figure 7: A graphical representation of the extensionalization of an indeterminate valid-time tuple

## 4.4 Now-relative Indeterminate Instants

To achieve the full benefit of indeterminate timestamp values, we proceed by introducing now-relative indeterminate instants, which may be understood as generalizations of the ground, indeterminate timestamps presented above and of the now-relative instants presented earlier.

To exemplify and motivate the utility of this new type of instant, assume that today is July 9, that Jane is still employed, and that there is at most a three-day lag in recording a fact in the database. Jane's tuple in the database should not be that of Figure 6(b), but rather that shown in Figure 8(a) which is more accurate. The state on July 10 is shown in Figure 8(b). Note how the indeterminacy in the "to" instant has decreased ever so slightly—on July 10 we know that Jane was employed on July 7.

| | **FACULTY** | | | |
|---|---|---|---|---|
| | | | VALID TIME | |
| | NAME | RANK | (from) | (to) |
| **(a)** | *Jane* | *Assistant* | *June 1* | *July 6* $\sim$ *January 1, 2028* |
| **(b)** | *Jane* | *Assistant* | *June 1* | *July 7* $\sim$ *January 1, 2028* |
| **(c)** | *Jane* | *Assistant* | *June 1* | *now* – 3 *days* $\sim$ *January 1, 2028* |

Figure 8: Using indeterminate and now-relative indeterminate timestamps

To accurately represent our continuously changing knowledge about Jane’s employment, we need to combine now-relative instants and ground indeterminate values into a new kind of instant, which we call a *now-relative indeterminate instant*. An example is shown in Figure 8(c) where the “to” timestamp is such an instant. Note that a tuple with a now-relative indeterminate instant may yield no definite information or may have the same possible and definite information content; it all depends upon when we observe that tuple.

The visualization of a tuple at the extensional database level with a now-relative indeterminate time is similar to the visualization of a tuple with an indeterminate interval. Both the definite and possible regions are plotted on the same graph but using different colors or patterns. Figure 7(b) shows a graph of both the possible and definite extensionalizations of the tuple in Figure 8(c) for every reference time between May 30 and January 5, 2028. Note that for all reference times before June 4 the tuple does not contain any definite information, only possible information. The definite information gradually increases as the reference time advances. On January 4, 2028, and for all reference times thereafter, the possible and definite information for the tuple are the same.

Now-relative indeterminate instants provide a flexible means of precisely capturing our imprecise, but current-time dependent, knowledge of when a fact is valid. For instance, in the tuple given in Figure 8(c), we are certain that Jane was an Assistant Professor starting on June 1, but our knowledge of when she ceases to be an Assistant Professor is imprecise; all we know is that she was definitely an Assistant Professor until three days ago and that it is possible that she will remain an Assistant Professor until retirement on January 1, 2028. The “to” timestamp allows us to capture this precisely. Using a now-relative indeterminate instant ensures that continual updates are not required, while capturing all of our knowledge of exactly when Jane is employed by State University.

A now-relative indeterminate instant consists of a variable lower bound and a ground upper bound. The lower bound cannot exceed the instant's upper bound, consequently the upper bound represents a limit on the possible or definite information in the instant. So, for instance, the possible or definite information represented by Jane's employment tuple shown in Figure 8(c) cannot extend beyond January 1, 2028, even if today is after January 1, 2028. If today is May 9, then the lower bound is May 6 and the tuple indicates that we *expect* Jane to be (possibly) employed from June 1 to January 1, 2028. If today is January 1, 2050, then the upper bound is January 1, 2028 and the tuple indicates that Jane was *actually* employed from June 1 to January 1, 2028. In short, now-relative indeterminate instants capture the semantics of predictive updates. They are also able to model the evolutionary character of temporal databases since values in the *possible* extensionalization of a tuple evolve into *definite* values as the reference time increases.

**Definition 10 (Possible Ext. of a Now-relative Indeterminate Tuple)** The possible extensionalization at reference time $rt_*$ of the tuple $T =< X, \ [e_1 \sim vt_2, e_3 \sim vt_4], >$ where $e_1$ and $e_3$ stand for "expressions using variables," and $vt_2$ and $vt_4$ are ground values, is defined as follows.

$$[\![T]\!]^P_{rt_*} =_{df} \{(X, vt, rt_*) | \ vt \in [\min(\langle\!\langle e_1\rangle\!\rangle_{rt_*}, \langle\!\langle vt_2\rangle\!\rangle_{rt_*}), \langle\!\langle vt_4\rangle\!\rangle_{rt_*}]\}. \qquad \square$$

**Definition 11 (Definite Ext. of a Now-relative Indeterminate Tuple)** The definite extensionalization of the tuple $T =< X, \ [e_1 \sim vt_2, e_3 \sim vt_4], >$ at reference time $rt_*$ is defined as follows.

$$[\![T]\!]^D_{rt_*} =_{df} \{(X, vt, rt_*) | \ vt \in [\langle\!\langle vt_2\rangle\!\rangle_{rt_*}, \min(\langle\!\langle e_3\rangle\!\rangle_{rt_*}, \langle\!\langle vt_4\rangle\!\rangle_{rt_*})]\}. \qquad \square$$

### 4.5 Summary of Extensionalizations

Table 1 summarizes some of the valid-time extensionalizations (the most representative cases). Case **v1** (the **v** stands for "valid-time" database) specifies the extensionalization of tuple timestamped with a determinate interval, case **v2** a now-relative interval, case **v3** an indeterminate interval, and case **v4** a now-relative indeterminate interval. Note that the possible and definite extensionalizations in cases **v1** and **v2** are the same since the intervals are determinate.

### 4.6 Querying Variable Valid-time Databases

In this section we enhance the query facilities of existing (non-variable) data models to support queries on timestamps containing variables. The essential problem is what to do when encountering a variable during query evaluation. Below, we describe a solution to that problem. Further, we show how the framework may be utilized in defining algebraic operators on variable databases that are consistent with the semantics of variable databases. Specifically, we consider the valid-time timeslice operation.

| | **Variable Database** | **Extensional Database** |
|---|---|---|
| **v1** | $T =< X, [vt_1, vt_2] >$ | $[\![T]\!]_{rt_*} = \{(X, vt, rt_*) \mid vt_1 \le vt \le vt_2\}$ |
| **v2** | $T =< X, [vt_1, now \pm n \text{ days }] >$ | $[\![T]\!]_{rt_*} = \{(X, vt, rt_*) \mid vt_1 \le vt \le rt_* \pm n\}$ |
| **v3$^D$** | $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$ | $[\![T]\!]^{D}_{rt_*} = \{(X, vt, rt_*) \mid vt_2 \le vt \le vt_3\}$ |
| **v3$^P$** | $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$ | $[\![T]\!]^{P}_{rt_*} = \{(X, vt, rt_*) \mid vt_1 \le vt \le vt_4\}$ |
| **v4$^D$** | $T =< X, [vt_1, now \pm n \text{ days } \sim vt_2] >$ | $[\![T]\!]^{D}_{rt_*} = \{(X, vt, rt_*) \mid vt_1 \le vt \le \min(rt_* \pm n, vt_2)\}$ |
| **v4$^P$** | $T =< X, [vt_1, now \pm n \text{ days } \sim vt_2] >$ | $[\![T]\!]^{P}_{rt_*} = \{(X, vt, rt_*) \mid vt_1 \le vt \le vt_2\}$ |

Table 1: Extensionalization of valid-time databases

When evaluating a user-level query, e.g., written in some dialect of SQL, it is common to transform it into an internal algebraic form that is suitable for subsequent rule or cost-based query optimization. As the query processor and optimizer are among the most complex components of a database management system, it is important that the added functionality of current-time-related timestamps necessitates only minimal changes to these components.

While many solutions may be envisioned, a solution that meets this requirement and is natural in our semantic framework is to eliminate variables before they are seen. More specifically, when a timestamp that contains a variable is used during query processing (e.g., in a test for overlap with another timestamp), a ground version of that timestamp is created and is used instead. Thus, only minimal, incremental changes to the query processor are needed. Existing components remain unchanged. Only a new component that substitutes variable timestamps with ground timestamps has to be added.

More specifically, we define a "bind" operator that is added to the set of operators already present. When user-level queries are mapped to the internal representation, this operator is utilized. The operator accepts any valid-time tuple with variables as defined earlier in the paper. It substitutes a ground value for each variable and thus returns a ground (but still variable-level) tuple.

To exemplify, assume that we on June 20 are interested in Jane's employment status at State University as of June 15 and that we have available the database with the single tuple in Figure 1(a), but that our query processor is unable to contend with variables in timestamps. To answer the query, we first eliminate the variable *now* by applying the bind operator (defined below) to the tuple, resulting in $\{<$*Jane*, *Assistant*, [*June 1, June 20*] $>\}$. Second, this tuple is passed to the query processor, where it is then used to compute that Jane is an Assistant Professor on June 15.

**Definition 12 (Variable-level Valid-time Bind)** Given an arbitrary valid-time tuple $T =< X, [e_1 \sim vt_2, e_3 \sim vt_4] >$ and a reference time $rt_*$, the variable-level valid-time bind operation eliminates all variables and is defined as follows.

$$bind^{V,VT}_{rt_*}(T) =_{df} < X, [\langle\!\langle e_1 \rangle\!\rangle_{rt_*} \sim \langle\!\langle vt_2 \rangle\!\rangle_{rt_*}, \langle\!\langle e_3 \rangle\!\rangle_{rt_*} \sim \langle\!\langle vt_4 \rangle\!\rangle_{rt_*}] > \qquad \square$$

This operation can be extended in the obvious way to an operator on sets of tuples, i.e., relations. The superscript, "$^{V,VT}$," indicates that this is a variable-level, valid-time operator. Note that two tuples that have timestamps "$[vt_1 \sim vt_1, vt_2 \sim vt_2]$" and "$[vt_1, vt_2]$," but are otherwise identical, have the same extensionalizations. Thus the timestamps are equivalent, and therefore the definition above also covers determinate timestamps.

The outcome of a query on a variable database generally depends on the specific reference-time argument given to the *bind* operator. To provide a foundation for understanding how to use the *bind* operator when mapping user-level queries to algebraic equivalents, we must explore its meaning.

The *bind* operator with reference-time argument $rt_*$ replaces each variable by its denotation or value at time $rt_*$. Put differently, the operator replaces each variable timestamp with a ground timestamp that has the special property of having the same denotation, or value, as the variable timestamp at the reference time $rt_*$. At other reference times, the original and the ground timestamps will generally not have the same denotation. This semantics may be expressed at the extensional level as follows.

**Definition 13 (Extensional-Level Valid-time Bind)** Given an arbitrary set $S$ of extensional-level valid-time tuples of the form $(X, vt, rt)$ and a reference time $rt_*$, the extensional-level valid-time bind operation is defined as follows.

$$bind_{rt_*}^{E,VT}(S) =_{df} \{(X, vt, rt) \mid (X, vt, rt_*) \in S \ \wedge \ rt \in \mathcal{T}_{RT}\} \qquad \Box$$

The “$^E$” in the operator’s superscript indicates that this is an extensional-level operator. At the extensional level, the bind operator chooses the meaning of a tuple at the indicated reference time and propagates that meaning over every possible reference time, resulting in a reference-time invariant meaning. To prove that this definition is correct *vis-á-vis* the required commutativity of the left side of the diagram in Figure 4, we need to show that given a tuple $T$, and a reference time $rt_*$, $[\![bind_{rt_*}^{V,VT}(T)]\!] = bind_{rt_*}^{E,VT}([\![T]\!])$. This follows directly from the definitions. For brevity, we omit the proof.

Intuitively, the *bind* operator sets the perspective of the observer, i.e., it sets the reference time as described in Section 3.2. Existing query languages generally assume that the perspective of a user observing the database is the same as what we termed the query time or current time and denoted $t_{current}$ in that section. However, as we shall see, a bind operator provides a basis for added functionality.

Recall that the definition of query operators at the variable level is complex and that current temporal data models have not satisfactorily resolved the complex problems involved. In our approach, we first preprocess the variable-level database by binding timestamps to $rt_*$, effectively removing the variables. We can then apply any algebraic operators from an existing temporal query language. It should be clear from the discussion above that the composition of bind with any of these algebraic operators is well-defined, and the timestamps have the appropriate meaning.

To show how operators are defined within the semantic framework, we now define several timeslice operators. *Valid-time timeslice* is a fairly standard operation; some variant of timeslice is a component of virtually all temporal algebras. Standard definitions of determinate and indeterminate timeslice operators are given below. Note that these do not have to contend with variables; because of the use of the bind operator, they can be defined solely on ground tuples.

**Definition 14 (Variable-level Definite Valid-time Timeslice)** Let $S$ be a set of tuples at the variable database level, i.e., a set of tuples of the form $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$, where the $vt_i$ are ground values. The definite valid-time timeslice of $S$ at valid time $vt_*$ is defined as follows.

$$\begin{aligned}\Pi^{D,V,VT}_{vt_*}(S) =_{df} \{ &< X, [vt_*, vt_*] > \mid \\ &\exists\, T = < X, [vt_1 \sim vt_2, vt_3 \sim vt_4] > \in S \ (vt_* \in [vt_2, vt_3])\} \quad \Box\end{aligned}$$

**Definition 15 (Variable-level Possible Valid-time Timeslice)** Let $S$ be a set of tuples at the variable database level, i.e., a set of tuples of the form $T =< X, [vt_1 \sim vt_2, vt_3 \sim vt_4] >$, where the $vt_i$ are ground values. The possible valid-time timeslice of $S$ at valid time $vt_*$ is defined as follows.

$$\begin{aligned}\Pi^{P,V,VT}_{vt_*}(S) =_{df} \{ &< X, [vt_*, vt_*] > \mid \\ &\exists\, T = < X, [vt_1 \sim vt_2, vt_3 \sim vt_4] > \in S \ (vt_* \in [vt_1, vt_4])\} \quad \Box\end{aligned}$$

The superscript "$^{D,V,VT}$" of the first operator indicates that it considers only the definite information contents, that it belongs at the variable level, and that it is a valid-time timeslice. Also, recall that definite timestamps are special cases of indeterminate timestamps, which are then also covered by the definition. The straightforward extensions of the operator to slice on valid-time intervals and to take as input a set of tuples (i.e., a relation), are omitted for brevity. A timeslice operator at the extensional level that satisfies the correctness criterion of the framework, as illustrated in Figure 3, specifically, $\Pi^{E,VT}_{vt_*}([\![T]\!]^D) = [\![\Pi^{D,V,VT}_{vt_*}(T)]\!]$, is given next. The proof of this statement is omitted for space considerations.

**Definition 16 (Extensional-level Valid-time Timeslice)** Since there is no indeterminacy at the extensional database level, there is no need for two timeslice operators; one suffices. At the extensional database level, the valid-time timeslice of a set $S$ consisting of tuples of the form $(X, vt, rt)$ is defined as follows.

$$\Pi^{E,VT}_{vt_*}(S) =_{df} \{(X, vt_*, rt) \mid (X, vt_*, rt) \in S\} \qquad \Box$$

We are now in a position to explore the interaction of the important times (Section 3.2) by using the new *bind* operator and existing (variable-level) timeslice operators. We generally consider only the definite version of the timeslice operator.

The *bind* operator sets the perspective and is combined with timeslice to formulate queries. In the first two example queries given below, we assume that the database is to be observed from the perspective of June 5, i.e. $rt_* =$ June 5. For all examples, the query time is assumed to also be June 5, i.e. $t_{current}$ = June 5.

- Who is employed on June 5?

$$\Pi^{D,V,VT}_{June\ 5}(bind^{V,VT}_{June\ 5}(Faculty))$$

- Who will actually be employed on June 7?

$$\Pi^{D,V,VT}_{June\ 7}(bind^{V,VT}_{June\ 5}(Faculty))$$

Tuples with a “to” time of *now* will *not* be in the result.

- Discounting future (and as yet unknown) employee hirings or firings, who do we expect to be employed on June 7?

  $$\Pi^{D,V,VT}_{June\ 7}(bind^{V,VT}_{June\ 7}(Faculty))$$

  Our “expectation” is that current employees (i.e., those employed *now*) will remain employed through June 7. We make this expectation concrete by adopting a June 7 perspective of the database. Then all tuples with a “to” time of *now* will contribute to the result.

- Making no assumptions about the future evolution of the database, who will possibly be employed on June 7?

  $$\Pi^{P,V,VT}_{June\ 7}(bind^{V,VT}_{June\ 5}(Faculty))$$

  We limit the future evolution of the database by adopting a June 5 perspective, and query about a possible future from that perspective. Tuples with intervals with a “to” time of *now* $\sim$ *June 7* (or a later upper bound) will be in the result, although those with a “to” time of *now* will not be in the result.

We have seen that the binding of *now* impacts the meaning of query results and that query results must be interpreted with respect to a particular perspective. Existing query languages, e.g., TSQL2 [46], generally assume that the perspective and the query time coincide. This assumption leads to a restriction in functionality, but it also simplifies the interpretation of answers.

The bind operation removes all variables from an answer to a query. Some users, however, might wish to see *now* in query results. One way to support *now* in query results is to redefine every temporal operation, making each sensitive to user perspective. For example, consider the temporal constructor, *First*. Currently, *First* is a reference-time independent operation that determines if one interval is earlier than another, and returns the tuple with the earlier interval. We could redefine *First* to use the meaning of a tuple with respect to a given reference time to determine which interval is earliest. By choosing a single reference time at which to evaluate *First* we are “temporarily” binding *now* to a particular reference time, but only during evaluation of the operation (and the binding is not made manifest in the result). There is one chief drawback to this method of supporting *now* in query results: the semantics of every existing temporal operation must be redefined to add a “temporary” bind of *now*. An important virtue of the “permanent” bind operation that we have described in this paper is that it is a minimal extension of the semantics and implementation of non-variable databases to support current-time variables. In terms of operations, only a single new operation, bind, need be added and implemented; the non-variable database semantics and implementation remain intact.

## 4.7 Summary

*Now* appears in many temporal database models, although it is sometimes disguised under a different name. *Now* is commonly used as the “to” time in a valid-time tuple. It has one principal advantage: it efficiently represents that a tuple will continue to be valid, barring further updates. But it also suffers from several anomalies, as discussed in Section 2.2. In the following, we show how each of these four anomalies are addressed in our approach.

The use of *now* as a “to” time makes a *pessimistic* assumption about a tuple’s continuing validity since it indicates that a tuple’s validity ends immediately, whereas we expect such tuples to remain valid in future. To address this problem we propose a semantics that allows users to bind *now* to any desired “perspective,” i.e., any reference time, in a query. The user can adopt a pessimistic perspective, by binding *now* to the current time, or an optimistic perspective, by binding *now* to some future time, e.g., *forever*. The proposed semantics is backwards-compatible with existing, non-variable semantics.

*Now* also imposes an unrealistic assumption about the *punctuality* of updates to a tuple because it presupposes that the current database state accurately models the current real-world state. To address this anomaly we introduce *now-relative* instants that include a displacement from *now*. Now-relative instants can relax the strict punctuality assumption by using the displacement from *now* to model the real-world delay in updating tuples.

Further, the use of *now* as a timestamp value necessitates special-case processing to correctly support *predictive* updates. A predictive update inserts into a database a fact that is valid sometime in the future. If such a fact has a “to” time of *now*, its valid time ends before it starts. This not only violates a common assumption about interval timestamps (that the timestamp is a valid interval), it can also lead to an incorrect result for a query about information valid in the future. To support predictive update we propose *now-relative indeterminate* instants that combine indeterminacy with now-relativity. An interval with a now-relative indeterminate instant as the “to” time is a valid interval no matter when it is inserted into the database. Furthermore, the indeterminacy in a now-relative indeterminate instant can be used to model the uncertainty of future information, while the now-relative portion of the instant relaxes the punctuality assumption and allows the user to adopt both optimistic and pessimistic query perspectives.

Finally, when querying data that involves *now*, the current time must be clearly specified since the value of *now* depends on this time. An unclear specification can result in ambiguous query results. In our proposed framework, the current time in fixed by the bind operation. This allows the perspective of the observer to be set, thereby ensuring that the same answer is always returned for a particular reference time.

## 5 Transaction-time Databases

The use of a current-time variable in the transaction-time dimension is not as fraught with problems as its use in the valid-time dimension. The reason for this lies in the different meaning of transaction time in a database. The valid time of a tuple indicates when it is considered valid, and, as such, valid timestamps of tuples are generally provided by the users. In contrast, transaction timestamps are supplied by the database management system itself. This is a consequence of the meaning of transaction time: the transaction timestamp indicates when the tuple is current in the database.

Although several timestamp values, e.g., *forever* and *now*, have been used, it is our contention that they all have the same meaning. Specifically, they are all employed as a “stop” timestamp that indicates that the tuple stamped is current (from the “start” time) until the database is updated to indicate otherwise. However, the various names used do not convey the intuitive semantics of the variable in this dimension. A term more precise than *now* or *forever* for this meaning of “not yet logically deleted or updated” is *until changed*—a fact is current in the database until changed. It has no counterpart in valid time. Using *until changed* instead of *now* avoids also potential confusion with *now* in valid time, although some authors have used *until changed* in valid time [52]. Unlike the (valid-time variable) *now*, *until changed* can only be used as the “stop” time; it is undefined to use it as the “start” time.

### 5.1 Extensionalization of a Ground Transaction-time Tuple

We first examine the meaning of a tuple without variables in transaction time. The extensionalization of such a tuple differs from its valid-time counterpart, because the semantics of transaction time does not allow future transaction times to be recorded in the database. Hence, the extensionalization of such tuples must be restricted to ensure that no matter when we look at the database, we can never see a “future” transaction time. Since the future depends on when we observe the database, the reference time is used to constrain the transaction-time in the expanded set of tuples.

In Definition 1, the denotation at any reference time of a ground valid-time instant was given to be the instant itself. The same applies to ground transaction-time instants.

**Definition 17 (Transaction-time Extensionalization of a Ground Tuple)** The transaction-time extensionalization of a tuple of the form $T =< X, [tt_1, tt_2] >$, where $X$ is some set of attribute values and $tt_1$ and $tt_2$ are transaction-time instants, at the reference time $rt_*$, where $t_0 \leq rt_* \leq t_{current}$, is defined as follows.

$$[\![T]\!]^{TT}_{rt_*} =_{df} \{(X, tt, rt_*) \mid tt \in [\langle\langle tt_1\rangle\rangle_{rt_*}, \min(\langle\langle tt_2\rangle\rangle_{rt_*}, rt_*)]\} \qquad \Box$$

We use a ${}^{TT}$ superscript to differentiate this mapping from a valid-time extensionalization.

The visualization of a transaction-time tuple is similar to that of a valid-time tuple. Again, a two-dimensional graph is used. The X-axis of the graph is the reference time, while the Y-axis is the transaction time. However, unlike a valid-time tuple without variables, the transaction-time interval for a tuple is not independent of the time at which we observe the tuple. Figure 9 depicts the extensionalization of the transaction-time tuple $<$ *Jane*, *Assistant*, [*June 5*, *June 8*] $>$ for a sequence of reference times, June 1 through June 11. Note that the depicted region has a "stair shaped" feature which is a result of the constraint that the transaction time cannot exceed the reference time.

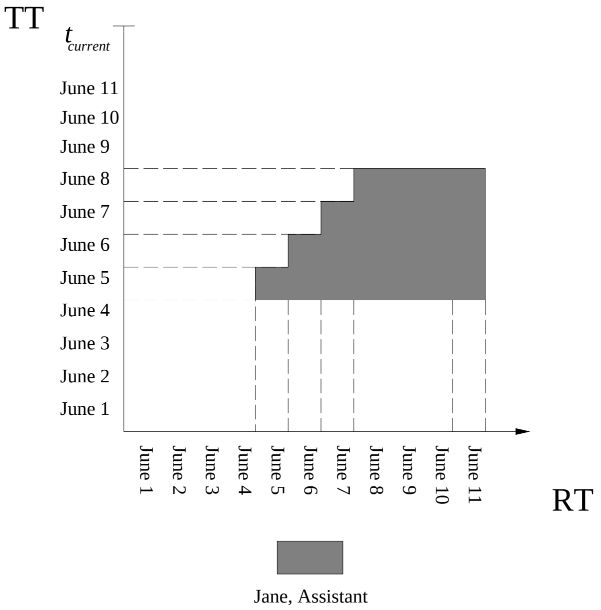


Figure 9: Graphical representation of a transaction-time tuple

## 5.2 Semantics of "Until Changed"

The current-time variable in transaction time indicates that the associated fact is current in the database until the fact is changed by a subsequent update. Substituting transaction time for valid time in our running example yields the relation shown in Figure 10.

**Definition 18 (Denotation of** *Until Changed***)** The *denotation* of the transaction-time variable *until changed* at a particular reference time $rt_*$, where $t_0 \leq rt_* \leq t_{current}$, is defined as follows.

$$\langle\langle \textit{until changed}\rangle\rangle_{rt_*} =_{df} rt_* \qquad \Box$$

| **FACULTY** | | | |
|---|---|---|---|
| | | TRANS TIME | |
| NAME | RANK | (start) | (stop) |
| *Jane* | *Assistant* | *June 1* | *until changed* |

Figure 10: Using until changed in a transaction-time relation

The extensionalization of a transaction-time tuple with the variable *until changed* as the value of its “stop” time is obtained by generating tuples for each instant in the ground interval that results from substituting *until changed* by $rt_*$. Thus, Definition 17 also applies when *until changed* is allowed as a “stop” time.

## 5.3 Summary of Extensionalizations

Table 2 summarizes the extensionalizations presented for transaction time. Case **t1** (the **t** stands for “transaction-time” database) applies to tuples with fully ground timestamp values only, whereas Case **t2** covers the case where *until changed* is the “stop” time.

| | **Variable Database** | **Extensional Database** |
|---|---|---|
| **t1** | $T =< X, [tt_1, tt_2] >$ | $[\![T]\!]_{rt_*} = \{(X, tt, rt_*) \mid tt_1 \leq tt \leq \min(tt_2, rt_*)\}$ |
| **t2** | $T =< X, [tt_1, \mathit{until\ changed}] >$ | $[\![T]\!]_{rt_*} = \{(X, tt, rt_*) \mid tt_1 \leq tt \leq rt_*\}$ |

Table 2: Extensionalization of transaction-time databases

## 5.4 Querying Variable Transaction-time Databases

The bind operator for transaction time eliminates occurrences of *until changed* in the “stop” component of timestamps.

**Definition 19 (Variable-level Transaction-time Bind)** Given a tuple $T = < X, [tt_1, e_2] >$, where $tt_1$ is a ground transaction time and $e_2$ is *until changed* or a ground transaction time, the variable-level transaction-time bind operation is defined as follows.

$$\mathit{bind}^{V,TT}(T) =_{df} < X, [\langle\!\langle tt_1 \rangle\!\rangle_{t_{current}}, \langle\!\langle e_2 \rangle\!\rangle_{t_{current}}] > \qquad \square$$

Again, this operation can be extended in the obvious way to relations. Transaction-time *bind* is very similar to valid-time bind, but differs in one important respect. The $\mathit{bind}^{V,TT}$ operator does not accept any time argument, but *always* binds *until changed* to the query time or current transaction time, $t_{current}$.

Since the $\mathit{bind}^{V,TT}$ operator lacks a time parameter and is always applied before any other operator, it is feasible to omit the operator and instead build it into the transaction timeslice operator, as has been done in some variable-level transaction-time algebras [29]. However, it would also need to be built into any additional operators, so to preserve the parallel with Section 4.6, we choose not to do this. The definition of the extensional-level bind for transaction time is omitted because it is very similar to Definition 13.

**Definition 20 (Variable-level Transaction-time Timeslice)** Let $S$ be a set of tuples at the variable database level, i.e., a set of tuples of the form $T \; =< \; X, [tt_1, tt_2] \; >$, where $tt_1$ and $tt_2$ are ground transaction times. The transaction-time timeslice of $S$ at transaction-time $tt_*$ is defined as follows.

$$\begin{aligned}\Pi^{V,TT}_{tt_*}(S) \; =_{df} \; \{< X, [tt_*, tt_*] > \mid \\ \exists\, T = \; < X, [tt_1, tt_2] > \in S \; (tt_* \in [tt_1, tt_2])\} \quad \Box\end{aligned}$$

**Definition 21 (Extensional-level Transaction-time Timeslice)** At the extensional database level, the transaction-time timeslice of a set $S$ consisting of tuples of the form $(X, tt, rt)$ is defined as follows.

$$\Pi^{E,TT}_{tt_*}(S) \; =_{df} \; \{(X, tt_*, rt) \mid (X, tt_*, rt) \; \in \; S\} \qquad \Box$$

The definitions of transaction-time binding and slicing conform to the framework we set up in Section 3.3, specifically to Figure 4, i.e., $\Pi^E, TT_t t * (bind^E_r t * ([[S]]_r t*) = [[\Pi^V, TT_t t * (bind^V_r t * (S))]]_r t*$. The proof, which follows from the definitions, is omitted for brevity.

As with valid-time queries, a combination of bind and timeslice supports transaction-time queries. When asking queries about a transaction-time database, there are two important times to consider: (i) the transaction-time timeslice time, $tt_*$, indicating that information is sought that was current in the database at time $tt_*$, and (ii) the query time, $t_{current}$, the time at which the query is asked.

As an example, we consider several timeslice operations on the tuple, $T$, depicted in Figure 2(a). For the following queries, it is assume that $t_{current}$ is June 11.

- $\Pi^{V,TT}_{June\ 11}(bind^{V,TT}(T))$ yields an empty result because the interval associated with $T$ is before the timeslice time—tuple $T$ ceased to be current starting on June 9.
- $\Pi^{V,TT}_{June\ 7}(bind^{V,TT}(T))$ yields tuple $T$, but with "start" and "stop" times of June 7. This is so because the information recorded by $T$ was current on June 7.

## 6 Bitemporal Databases

A *bitemporal* relation supports both transaction and valid time [33, 47]. The combination of these two temporal dimensions empowers the database to record time-dependent information as well as earlier database states. Bitemporal databases thus combine the advantages of valid-time and transaction-time databases. Yet, this greater flexibility comes at a cost: increased complexity derives from the interactions between the two temporal dimensions which must be carefully considered. The logical framework we have presented for current-time variables has been designed to make it relatively straightforward to obtain the semantics of bitemporal databases. The interaction between the current-time variable for valid time, *now*, and transaction time, *until changed*, is coordinated through the reference time. We demonstrate *one* possible (and, we think, reasonable) semantics for this combination, but we emphasize that the framework is general enough to allow the definition of other, alternative semantics for the interaction of these variables.

## 6.1 Extensionalization of Bitemporal Databases

The timestamp of a bitemporal tuple contains both a valid-time and a transaction-time component. Since the valid-time component may be indeterminate, it is necessary to distinguish between a definite and a possible extensionalization, $[\![\ ]\!]_{rt_*}^{BT,D}$ and $[\![\ ]\!]_{rt_*}^{BT,P}$, respectively.

**Definition 22 (Definite Extensionalization of a Bitemporal Tuple)** The definite extensionalization of a bitemporal tuple $T$ of the form $T = < X, [vt_1, vt_2], [tt_1, tt_2] >$, where $X$ is some set of attribute values and the timestamp $[vt_1, vt_2], [tt_1, tt_2]$ may contain any of the variables introduced earlier, at the reference time $rt_*$ is defined as follows.

$$\begin{aligned}[\![T]\!]_{rt_*}^{BT,D} =_{df} \{(X, vt, tt, rt_*) \mid {} & (X, vt, rt_*) \in [\![ < X, [vt_1, vt_2] > ]\!]_{rt_*}^{VT,D} \wedge \\ & (X, tt, rt_*) \in [\![ < X, [tt_1, tt_2] > ]\!]_{rt_*}^{TT}\} \quad \Box\end{aligned}$$

**Definition 23 (Possible Extensionalization of a Bitemporal Tuple)** The possible extensionalization of a bitemporal tuple $T$ of the form $T = < X, [vt_1, vt_2], [tt_1, tt_2] >$ at the reference time $rt_*$ is defined as follows.

$$\begin{aligned}[\![T]\!]_{rt_*}^{BT,P} =_{df} \{(X, vt, tt, rt_*) \mid {} & (X, vt, rt_*) \in [\![ < X, [vt_1, vt_2] > ]\!]_{rt_*}^{VT,P} \wedge \\ & (X, tt, rt_*) \in [\![ < X, [tt_1, tt_2] > ]\!]_{rt_*}^{TT}\} \quad \Box\end{aligned}$$

The definitions show that the framework has been constructed so that the extensionalization of bitemporal tuples is the combination of the extensionalizations for valid and transaction time. It also shows how the reference time $rt_*$ serves as an essential coordination mechanism between the valid and transaction time components of the timestamp: the same reference time appears in the valid-time and in the transaction-time denotations. Although, it is possible and may be interesting to consider situations where the two reference times differ, we have found that for all practical purposes this coordination is desirable. Nevertheless, other kinds of coordination through the reference time are possible. For example, instead of the standard Cartesian product used here, a coordination mechanism that utilizes a step-wise cross product of the two temporal dimensions is possible [12].

Another feature of the framework is that the uniform and component-wise treatment of time dimensions makes it easy to include additional dimensions. To specify the semantics of a variable database with additional dimensions, it is necessary to first specify the semantics of the variables and tuples in that new dimension, e.g., as is done for the transaction-time dimension in Section 5. Subsequently, the new dimension can be easily integrated with the other dimensions in a definition similar to the one above. Thus, our framework can be extended to encompass multidimensional temporal databases (also termed *indexical* [7] and *parametric* [26] temporal databases), for example *temporally generalized* [31] and *spatio-temporal* [1] databases.

Tables 1 and 2 may be combined to cover the bitemporal extensionalizations. The combination of Case **v1** from Table 1 and Case **t1** from Table 2 gives the bitemporal extensionalization for a tuple timestamped with a determinate valid time interval, $[vt_1, vt_2]$, and a transaction time interval, $[tt_1, tt_2]$, both without variables. Note that the transaction time in this case is restricted to the “past” relative to the reference time, just as in transaction-time tuples. For example, the extensionalization at reference time June 2 of the tuple

$$< \mathit{Jane}, \mathit{Assistant}, [\mathit{June\ 3}, \mathit{June\ 10}], [\mathit{June\ 1}, \mathit{June\ 3}] >$$

is

$$\{(\mathit{Jane}, \mathit{Assistant}, vt, tt, \mathit{June\ 2}) \mid \\ vt \in [\mathit{June\ 3}, \mathit{June\ 10}] \wedge tt \in [\mathit{June\ 1}, \min(\mathit{June\ 2}, \mathit{June\ 3})]\}.$$

In this example, the terminating transaction time, June 3, is constrained by the reference time, June 2.

The graphical representation of bitemporal tuples is three-dimensional; transaction time is the X-axis, valid time is the Y-axis, and reference time is the Z-axis. To this point, the reference time has been the X-axis, but making the reference time the Z-axis in the three-dimensional visualization results in a better picture. The graph is displayed so that the Z-axis goes “into” the page. The three-dimensional picture of a bitemporal tuple allows us to represent the passage of time as a spatial displacement, and provides a visual representation for interesting phenomena such as history changes and predictions about the future, as well as incorporating the viewpoint of an observer into these phenomena. As we will see below, the graphical representation shows the subtle interaction between *now*, *until changed*, and the reference time.

Examples of the combinations of the extensionalizations presented in Tables 1 and 2 are graphically depicted in Figures 11 and 12. The dotted line vectors in the graph represent directions of growth as either the reference time, valid time, or transaction time extends to $\top$. Only one generic example tuple is depicted in each case. The evolutionary nature of temporal databases, a key concept, comes through very clearly in the figures. Notice how the shaded areas grow as reference time increases, most prominently for tuples containing variables, indicating an accumulation of knowledge stored in the database. Note also how information in later reference times is always consistent with that in earlier reference times.

Figure 11 illustrates the determinate cases. For example, the lower right corner of Figure 11 depicting the **v2**$\times$ **t2** case, shows how *now* and *until changed* are bound to an increasing reference time, resulting in a three-dimensional stair-shaped pattern. The tuple’s extensionalization grows as time passes encompassing more points. In contrast, case **v1** $\times$ **t1** depicts constrained growth, as the tuple ceases to exist beyond transaction time $tt_2$. Note that, unless a tuple is known to have been deleted from the database, its “transaction-stop time” is *until changed*, and hence it has unlimited growth in the transaction-time dimension. This is true for the determinate cases shown in Figure 11 as well as for the indeterminate cases of Figure 12. Notice for example, how the possible and definite extensionalizations in cases $\mathbf{v3}^D \times \mathbf{t2}$ and $\mathbf{v3}^P \times \mathbf{t2}$, the upper right-hand corner of Figure 12, remain constant in the valid-time dimension while growing in transaction-time. In contrast, case $\mathbf{v4}^D \times \mathbf{t2}$ illustrates constrained growth, i.e., constant evolution up through time $vt_2$.

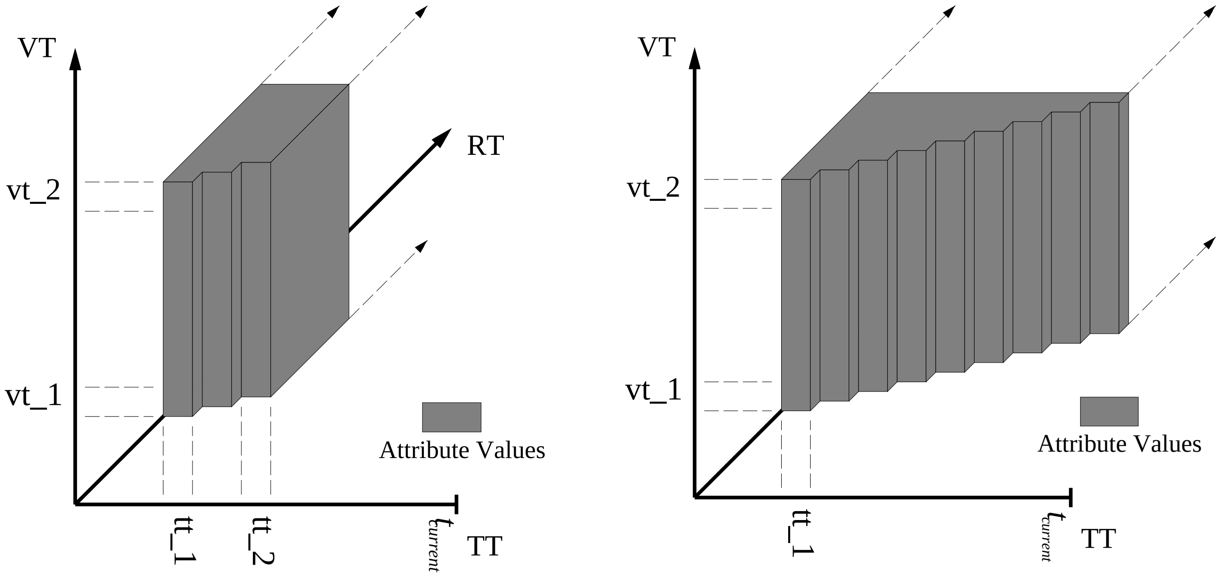
## 6.2 Querying Variable Bitemporal Databases

The existing bind and timeslice operators, developed for valid-time and transaction-time databases, are easily generalized to apply to bitemporal databases. A bitemporal tuple differs from a valid-time tuple by having a transaction time interval in its timestamp. The valid-time operators are generalized to corresponding bitemporal operators by simply ignoring this extra timestamp. For example, the definite bitemporal valid-time timeslice is defined by generalizing Definition 14 as follows.

**Definition 24 (Variable-level Definite Bitemporal Valid-time Timeslice)** Let $S$ be a set of tuples at the variable database level, i.e., a set of tuples of the form $T = < X, [vt_1 \sim vt_2, vt_3 \sim vt_4], [tt_1, tt_2] >$, where $T$ is ground. The definite bitemporal valid-time timeslice of $S$ at valid time $vt_*$ is defined as follows.

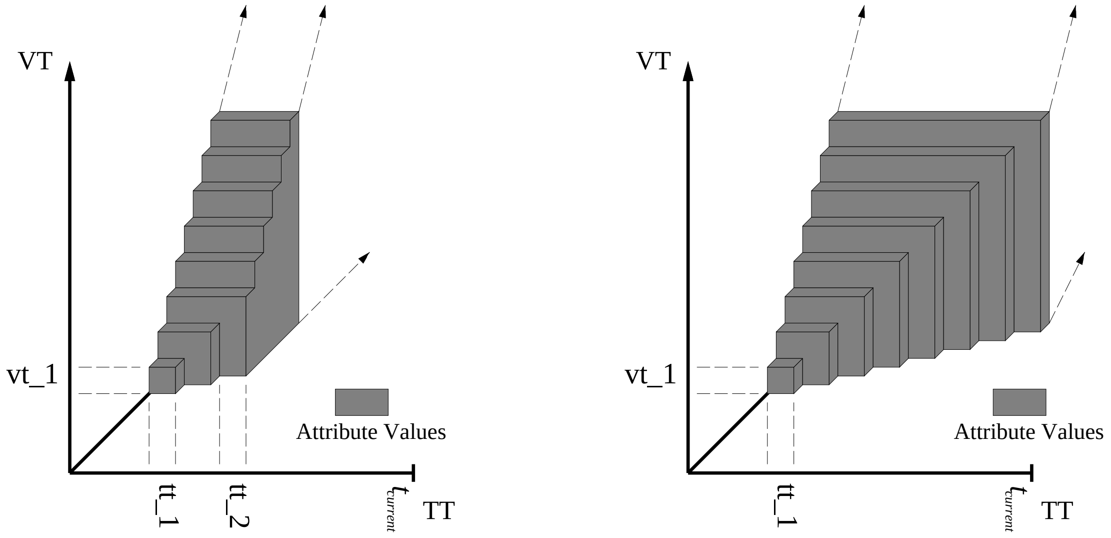
$$\begin{aligned}\Pi^{D,V,VT,BT}_{vt_*}(S) =_{df} \{ &< X, [vt_*, vt_*], [tt_1, tt_2] > \mid \\ &\exists\, T = < X, [vt_1 \sim vt_2, vt_3 \sim vt_4], [tt_1, tt_2] > \in S\ (vt_* \in [vt_2, vt_3])\} \quad \Box\end{aligned}$$

The superscript “$^{D,V,VT,BT}$” indicates that the operator considers only the definite information in the tuple, belongs at the variable level, performs a timeslice in the valid-time dimension, and is applicable to bitemporal tuples. In addition to this operator, the subsequent discussion uses the operators $\Pi^{P,V,VT,BT}$, $\Pi^{V,TT,BT}$,



Case **v1** × **t1**

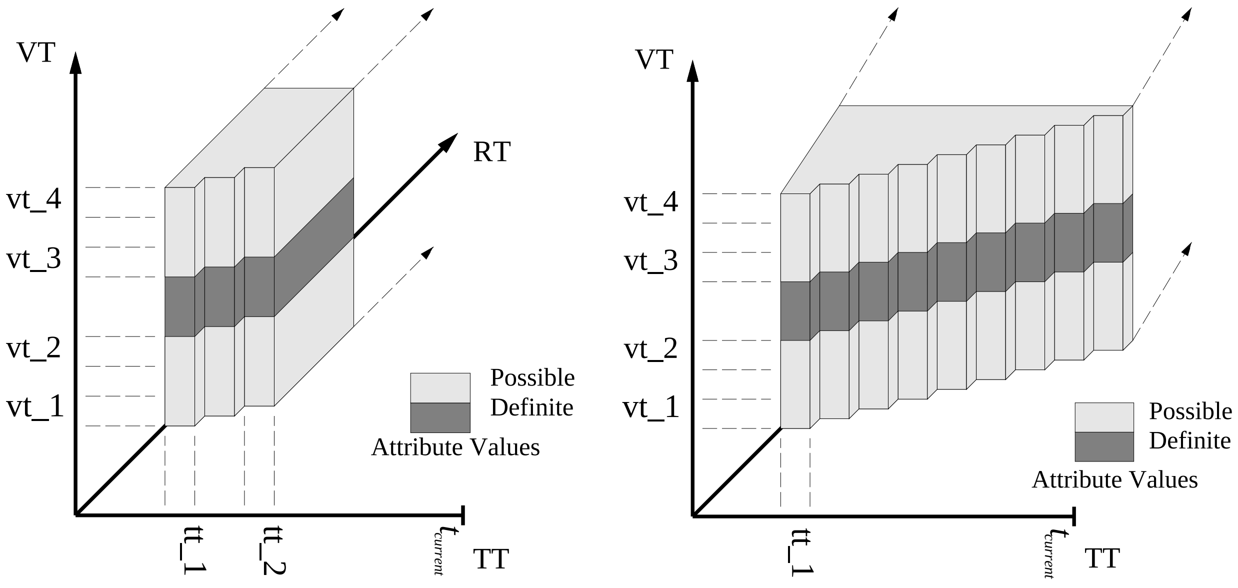
Case **v1** × **t2**



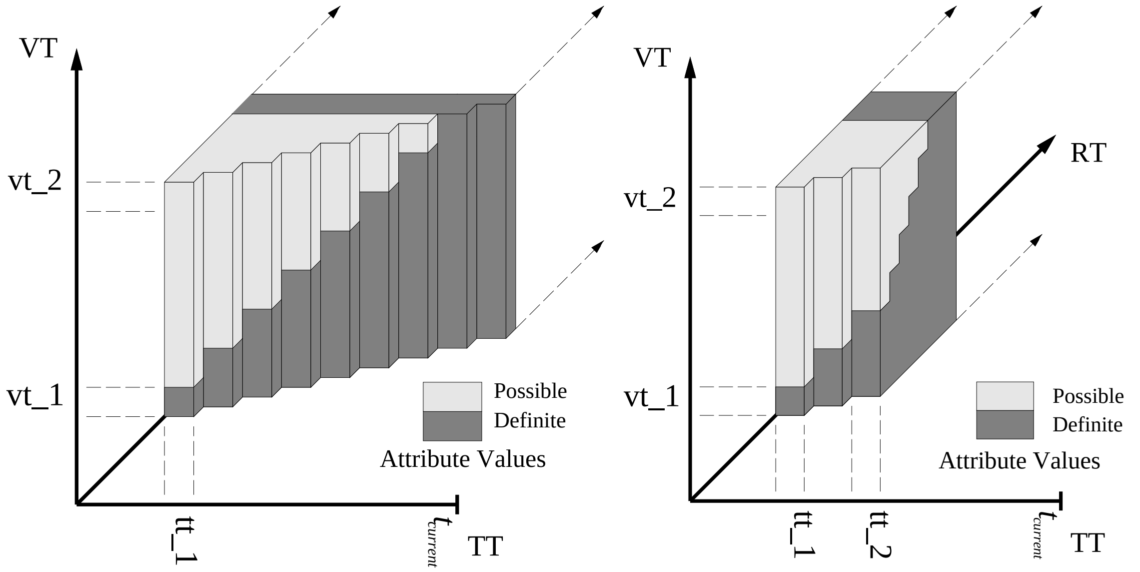
Case **v2** × **t1**

Case **v2** × **t2**

Figure 11: Examples of the bitemporal determinate cases



Cases $\mathbf{v3}^D \times \mathbf{t1}$ and $\mathbf{v3}^P \times \mathbf{t1}$ Cases $\mathbf{v3}^D \times \mathbf{t2}$ and $\mathbf{v3}^P \times \mathbf{t2}$



Cases $\mathbf{v4}^D \times \mathbf{t1}$ and $\mathbf{v4}^P \times \mathbf{t1}$ Cases $\mathbf{v4}^D \times \mathbf{t2}$ and $\mathbf{v4}^P \times \mathbf{t2}$

Figure 12: Examples of the bitemporal indeterminate cases

$bind^{V,VT,BT}$, and $bind^{V,TT,BT}$ which are all similar generalizations of previous definitions.

As with valid-time and transaction-time databases, queries are evaluated by combining bitemporal timeslice and bind operations. Also as before, valid and transaction times must be bound before the bitemporal valid-time timeslice or bitemporal transaction-time timeslice, respectively, can be applied.

To explore the interaction of times in queries on bitemporal databases, we consider a number of queries on the simple database depicted in Figure 13 which shows that Jane's employment tuple was added to the database on June 2. Note that it contains an now-relative indeterminate "to" time and "*until changed*" as the "stop" time. For the purpose of the example, we assume that today is July 9. Thus, the transaction-time bind operator binds *until changed* to July 9 in all queries. The first four queries all include Jane in the result.

- Using the current database state, who was possibly a faculty member on July 7?

  $$\Pi^{P,V,VT,BT}_{\mathit{July\ 7}}(bind^{V,VT,BT}_{\mathit{July\ 9}}(\Pi^{V,TT,BT}_{\mathit{July\ 9}}(bind^{V,TT,BT}(\mathit{Faculty}))))$$

  In this query, we transaction timeslice to get only the most current information. The valid-time bind ensures a perspective of today, and the valid timeslice retrieves those tuples that were possibly valid two days ago (on July 7).

- Using the current database state, who was definitely a faculty member on July 1?

  $$\Pi^{D,V,VT,BT}_{\mathit{July\ 1}}(bind^{V,VT,BT}_{\mathit{July\ 9}}(\Pi^{V,TT,BT}_{\mathit{July\ 9}}(bind^{V,TT,BT}(\mathit{Faculty}))))$$

  As before, the lower bound of the "to" time is ground to July 6 (July 9 $-$ 3 days). The difference is solely in the valid timeslice; we require definite information, and so we use a definite timeslice. Since July 1 is before July 6, Jane is in the result.

- Using the database state on July 1, who was definitely a faculty member on June 15?

  $$\Pi^{D,V,VT,BT}_{\mathit{June\ 15}}(bind^{V,VT,BT}_{\mathit{July\ 1}}(\Pi^{V,TT,BT}_{\mathit{July\ 1}}(bind^{V,TT,BT}(\mathit{Faculty}))))$$

  The transaction timeslice retrieves the information current on July 1. The valid-time bind adopts this day as the perspective of the subsequent valid timeslice which retrieves information about June 15. Since Jane's tuple was current on July 1 and June 15 is more than three days before July 1, Jane will be in the result.

- Using the current database state, who will we say on July 12 is possibly a faculty member on September 1?

  $$\Pi^{P,V,VT,BT}_{\mathit{September\ 1}}(\mathit{bind}^{V,VT,BT}_{\mathit{July\ 12}}(\Pi^{V,TT,BT}_{\mathit{July\ 9}}(\mathit{bind}^{V,TT,BT}(\mathit{Faculty}))))$$

  We first consider only current information. Then we adopt a valid-time perspective of July 12 to examine the database as it will appear on July 12 if no updates are made, i.e., our best guess as to what will be current information on July 12. Finally, using that perspective, we ask about possible information on September 1. Jane will thus be in the result.

In contrast to the queries above, the following three queries do *not* include Jane in the result.

- Using the current database state, who was definitely on the faculty of State University on July 7?

  $$\Pi^{D,V,VT,BT}_{\mathit{July\ 7}}(\mathit{bind}^{V,VT,BT}_{\mathit{July\ 9}}(\Pi^{V,TT,BT}_{\mathit{July\ 9}}(\mathit{bind}^{V,TT,BT}(\mathit{Faculty}))))$$

  Here, the valid-time bind operation yields a ground “to” time of July 6 $\sim$ January 1, 2028. Since July 7 is after July 6, Jane is possibly, but not definitely, on the faculty.

- Using the database as of July 1, who was definitely a faculty member on July 1?

  $$\Pi^{D,V,VT,BT}_{\mathit{July\ 1}}(\mathit{bind}^{V,VT,BT}_{\mathit{July\ 1}}(\Pi^{V,TT,BT}_{\mathit{July\ 1}}(\mathit{bind}^{V,TT,BT}(\mathit{Faculty}))))$$

- Using the current database state, who will we say on July 12 is definitely a faculty member on September 1?

  $$\Pi^{D,V,VT,BT}_{\mathit{September\ 1}}(\mathit{bind}^{V,VT,BT}_{\mathit{July\ 12}}(\Pi^{V,TT,BT}_{\mathit{July\ 9}}(\mathit{bind}^{V,TT,BT}(\mathit{Faculty}))))$$

In most of the examples above, the transaction timeslice time and the valid-time bind time, or reference time, are the same. Indeed, this is the typical and most useful scenario, as the following example makes clear. Suppose that today, July 9, we execute a transaction-time timeslice with time argument February 1 (that is, the *preceding* February 1). This operation chooses the most up-to-date information as of February 1, and disregards information that was not up-to-date on February 1 or was recorded at a later time. The user’s perspective for subsequent operations using this information should naturally switch to the frame of reference of the chosen information. Hence, for this example, it would be natural to also bind *now* to February 1.

Yet, two of the queries given above illustrate that this in not a necessary restriction. Lifting it leads to increased functionality, but also to queries that are conceptually more involved. Existing query languages generally enforce this restriction.

| **FACULTY** | | | | | |
|---|---|---|---|---|---|
| | | VALID TIME | | TRANS TIME | |
| NAME | RANK | (from) | (to) | (start) | (stop) |
| *Jane* | *Assistant* | *June 1* | *(now − 3 days) ∼ January 1, 2028* | *June 2* | *until changed* |

Figure 13: A bitemporal relation

## 7 Timestamp Implementation

This paper has proposed four new current-time-related timestamps; namely, *until changed*, *now*, now-relative instants, and now-relative indeterminate instants. Elsewhere we show how these timestamps may be efficiently represented [18, 20, 10, 21]. For example, a now-relative timestamp can be encoded as a datetime value coupled with a one-bit flag differentiating it from a ground timestamp. Consequently, the timestamps proposed in this paper impose little space overhead.

We also proposed adding bind operations for valid time, transaction time, and bitemporal databases; no other operations are needed to support current-time-related modeling entities. The bind operations have no significant impact on the run-time efficiency of a temporal database. The transaction-time bind is very efficient. It simply replaces *until changed* with the current transaction time. The valid and bitemporal bind operations are only slightly less efficient. For now-relative instants (and now-relative indeterminate instants) these operations replace *now* with the reference time and then displace that reference time by a span. The displacement costs one integer addition operation.

Now-relative instants also add an extra comparison to interval constructors. As we observed in Section 2.2, predictive updates could insert into the database intervals that end before they start. For a tuple without variables, such intervals can be detected and eliminated when the tuple is first inserted into the database. But a tuple with a variable might initially end before it starts, and only later evolve into a valid interval. Consequently, during run-time each interval involving a variable must be tested to ensure that the starting instant is before the terminating instant. This test needs to be performed only once per interval per query.

## 8 Summary and Research Directions

The overall conclusion of this paper is a recommendation that timestamps involving current-time variables–that is, *now*, *until changed*, now-relative, and now-relative indeterminate timestamps–be allowed to be stored as values of columns, for conventional and temporal databases, as well as implicit valid and transaction timestamps, for temporal databases.

This paper makes a number of contributions. First, it provides a formal basis for defining the semantics of databases with variables. The use and generality of the framework was demonstrated by giving a semantics for conventional, valid-time, transaction-time and bitemporal databases with all existing variables. Apart from specifying a reasonable semantics for such databases, this exercise demonstrates two important properties of the framework. The first property is that it is capable of capturing the semantics of a wide range of variables. We provide the semantics of variables of all kinds of general temporal aspects of database facts currently identified [33]: user-defined time, valid time, and transaction time. The second is that the semantics of a multidimensional database may be specified as a coordinated combination of the semantics of the constituent one-dimensional databases. The reference-time dimension in the framework provides the coordination mechanism. For example, the semantics of variable bitemporal databases was specified very easily by using the already specified semantics for valid-time and transaction-time databases. This property makes it relatively easy to specify the semantics of multidimensional databases. It also makes it easy to add further kinds of time that may emerge in the future, as well as other dimensions, such as space, again, in all their various combinations.

Second, without current-time variables, temporal databases provide inadequate support for their applications. The paper demonstrates that existing variables, such as *now* and *until changed*, are indispensable in databases. It also identifies situations were even these variables are inadequate, and introduces new *now-relative* and *now-relative indeterminate* instants that provide the desired support. The semantics of databases with variables are also defined within the framework.

Third, a foundation for the querying of variable databases from existing temporal query languages was presented. The paper provides algebraic "bind" operators for valid-time, transaction-time, and bitemporal databases, and it shows how these can be used to permit existing query languages to access variable databases. As a first step during query processing, the bind operation is applied to variable databases, thus replacing all variables with ground values appropriate for the processing of the query at hand. The framework also clarifies which values are the appropriate ground values. This approach encapsulates the handling of variables in a single operator per temporal dimension. It also requires only minimal changes to the query processor: support for one new operator has to be added, but all other components remain unchanged.

These three observations provide the rationale for the conclusion that variable databases are viable. A number of secondary, but noteworthy, contributions also deserve mention. The paper resolves the meaning of the use of variables in existing temporal data models. A graphical notation with two or three dimensions used throughout the paper proved to be helpful when describing the semantics of variable databases. The complex interactions of current time, reference time, transaction time, and valid time within queries and variable databases were investigated in detail. These interactions were not thoroughly understood or explicated in existing bitemporal data models. The concept of "perspective" within queries was illustrated. Perspective adds the ability to bind the valid-time variable *now* "to" times other than the current time. Supporting this notion within a query language enhances its functionality when querying variable databases.

This framework has implications for database query language design. The user-defined time types available in SQL-92 can be easily extended to store now-relative and indeterminate non-relative variables as values in columns. The TSQL2 language [46] does so, and also supports those variables for valid and transaction time. In TSQL2 the “bind” operation is implicit; `NOBIND` is provided to store variables in the database.

There are several directions for future research. The precise semantics of several temporal models proposed in the literature could profitably be examined in light of the framework presented here. In defining the semantics for bitemporal databases, we have chosen but one possible way of combining the semantics of valid-time and transaction-time databases; other possible combinations of these two temporal dimensions might also prove useful. In addition, the use of the graphical representation of temporal relations at the user interface—for displaying the results of queries and, e.g., for the assertion of temporal integrity constraints—seems to us a promising one for further research. The impact of stored variables on database storage structures and access methods is an open problem. It also presents an opportunity, e.g., if the optimizer knows (through attribute statistics) that a large proportion of a tuples have a “to” time of now. It may then decide that a sort-merge temporal join will be less effective. Finally, new kinds of variables, such as *here* for spatial and spatio-temporal databases, should be investigated, as an extension of the framework introduced in this paper.

## 9 Acknowledgments

Partial support for Curtis Dyreson and Richard Snodgrass was provided by the National Science Foundation through grants IRI–8902707 and IRI–9302244, the IBM Corporation through Contract #1124, and the AT&T Foundation. Partial support for Christian S. Jensen was provided by the Danish Natural Science Research Council through grants 11–9675–1 SE and 11–0061. Won Kim, Nick Kline and the anonymous referees provided helpful comments on a previous draft.

## References

[1] K. K. Al-Taha, R. T. Snodgrass, and M. D. Soo. Bibliography on Spatiotemporal Databases. *International Journal of Geographical Information Systems*, 8(1):95–103, January-February 1994.

[2] G. Ariav, A. Beller, and H. L. Morgan. A Temporal Data Model. Technical Report DS-WP 82-12-05, Decision Sciences Department, University of Pennsylvania, December 1984.

[3] M. A. Bassiouni and M. J. Llewellyn. A Relational-calculus Query Language for Historical Databases. *Computer Languages*, 17(3):185–197, 1992.

[4] J. Ben-Zvi. *The Time Relational Model*. Ph.D. thesis, University of California at Los Angeles, 1982.

[5] G. Bhargava and S. Gadia. Achieving Zero Information Loss in a Classical Database Environment. In *Proceedings of the International Conference on Very Large Databases*, pages 217–224, Amsterdam, August 1989.

[6] V. Brusoni, L. Console, P. Terenziani, and B. Pernici. Extending Temporal Relational Databases to Deal with Imprecise and Qualitative Temporal Information. In *Proceedings of the VLDB International Workshop on Temporal Databases*, J. Clifford and A. Tuzhilin (editors), Workshops in Computing Series, Springer Verlag, Zurich, Switzerland, pages 3–22, September, 1995.

[7] J. Clifford. Indexical Databases. In *Advanced Database Systems*, Lecture Notes in Computer Science 759, Springer-Verlag, 1993.

[8] J. Clifford and A. Croker. The Historical Relational Data Model HRDM and Algebra Based on Lifespans. In *Proceedings of the IEEE International Conference on Data Engineering*, pp. 528–537, Los Angeles, February 1987.

[9] J. Clifford, A. Croker, and A. Tuzhilin. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, 19(2):64–116, March 1994.

[10] J. Clifford, C. E. Dyreson, T. Isakowitz, C. C. Jensen, and R. T. Snodgrass, “On the Semantics of ‘Now’ in Temporal Databases.” Technical Report R-94-2047, Aalborg University, Department of Mathematics and Computer Science, Frederik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark, November, 1994.

[11] J. Clifford and T. Isakowitz, On The Semantics of Transaction Time and Valid Time in Bitemporal Databases. In *Proceedings of the ARPA/NSF International Workshop on an Infrastructure for Temporal Databases*, R. T. Snodgrass, editor, pp. I.1–I.17, Arlington, TX, June 1993.

[12] J. Clifford and T. Isakowitz, On The Semantics of (Bi)Temporal Variable Databases. In *Proceedings of the Fourth International Conference on Extending Database Technology*, pp. 215–230, Cambridge, England, March 1994.

[13] J. Clifford and A. U. Tansel. On an algebra for historical relational databases: Two views. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, S. Navathe, editor, pp. 247–265, Austin, TX, May 1985.

[14] J. Clifford and D. S. Warren. Formal Semantics for Time in Databases. *ACM Transaction On Database Systems*, 8(2):214–254, 1983.

[15] E. F. Codd, A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[16] C. J. Date and C. J. White. *A Guide to DB2*, Volume 1, 3rd edition. Addison-Wesley, Reading, MA, September 1990.

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

[18] C. E. Dyreson and R. T. Snodgrass. Timestamp Semantics and Representation. *Information Systems*, 18(3):143–166, 1993.

[19] 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.

[20] C. E. Dyreson and R. T. Snodgrass. A Timestamp Representation. Chapter 25 of [46], pp. 475–499.

[21] C. E. Dyreson. Valid-time Indeterminacy. Ph.D. thesis, Computer Science Department, University of Arizona. October 1994, 187 pages.

[22] R. Elmasri, G. Wuu, and Y. Kim. The Time Index — an Access Structure for Temporal Data. In *Proceedings of the International Conference on Very Large Databases*, Brisbane, Australia, August 1990.

[23] M. Finger. Handling Database Updates in Two-dimensional Temporal Logic. *Journal of Applied Non-Classical Logics*, 2(2), 1992.

[24] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*. North-Holland Publishing Company, 1973.

[25] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transaction On Database Systems*, 13(4):418–448, 1988.

[26] S. Gadia and S. Nair. Temporal Databases: A Prelude to Parametric Data. Chapter 2 of [50], pp. 28–66.

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

[28] C. S. Jensen and L. Mark. A Framework for Vacuuming Temporal Databases. Technical Report CS-TR-2516/UMIACS-TR-90-105, University of Maryland, Department of Computer Science, College Park, MD, August 1990.

[29] C. S. Jensen and L. Mark. Queries on Change in an Extended Relational Model. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):192–200, April 1992.

[30] C. S. Jensen and R. T. Snodgrass. Temporal Specialization. In F. Golshani, editor, *Proceedings of the IEEE International Conference on Data Engineering*, pp. 594–603, Tempe, AZ, February 1992.

[31] C. S. Jensen and R. T. Snodgrass. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–974, December 1994.

[32] C. S. Jensen and R. T. Snodgrass "Semantics of Time-Varying Information," *Information Systems*, to appear.

[33] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia (editors). A Consensus Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52–65, March 1994.

[34] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7):513–547, December 1994.

[35] W. Kurutach and J. Franklin. On Temporal-fuzziness in Temporal Fuzzy Databases. In *DEXA'93*, pages 154–165, Prague, Czech Republic, September 1993.

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

[37] N.A. Lorentzos and R.G. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, 13(3):286-296, 1988.

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

[39] R. Montague. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, 1974.

[40] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Information Sciences*, 49:147–175, 1989.

[41] R. Reiter. Towards a Logical Reconstruction of Relational Database Theory. In *On Conceptual Modelling*, pp. 191–233. Springer Verlag, 1984.

[42] J. F. Roddick. Schema Evolution in Database Systems — An Annotated Bibliography. *SIGMOD Record*, 21(4):35–40, December 1992.

[43] N. L. Sarda. Algebra and Query Language for a Historical Data Model. *The Computer Journal*, 33(1):11–18, February 1990.

[44] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[45] R. T. Snodgrass. An Overview of TQuel. Chapter 6 of [50], pp. 141–182.

[46] R. T. Snodgrass (editor). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995, 674+xxiv pages.

[47] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In S. Navathe, editor, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 236–246, Austin, TX, May 1985.

[48] J. B. Sykes, editor. *The Concise Oxford Dictionary*. Oxford University Press, Oxford, England, 1964.

[49] A. U. Tansel. Modelling Temporal Data. *Information and Software Technology*, 32(8):514–520, October 1990.

[50] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[51] S. Thirumalai and S. Krishna. Data Organization for Temporal Databases. Technical report, Raman Research Institute, Bangalore, India, 1988.

[52] G. Wiederhold, S. Jajodia, and W. Litwin. Integrating Temporal Data in a Heterogeneous Environment. Chapter 22 of [50], pp. 563–579.

[53] C. Yau and G. S. W. Chat. TempSQL — a Language Interface to a Temporal Relational Model. *Information Sc. & Tech.*, pp. 44–60, October 1991.