

Temporal Data Model and Query Language Concepts

Michael H. Böhlen and Christian S. Jensen

Aalborg University

- I. INTRODUCTION
- II. TEMPORAL DATA SEMANTICS
- III. TEMPORAL DATA MODELS

- IV. TEMPORAL QUERY LANGUAGES
- V. SUMMARY
- VI. OUTLOOK

GLOSSARY

bitemporal relation A relation with exactly one system-supported valid time and exactly one system-supported transaction time.

calendar Provides a human interpretation of time. As such, calendars ascribe meaning to temporal values where the particular meaning or interpretation is relevant to the user. In particular, calendars determine the mapping between human-meaningful time values and an underlying time-line.

chronon, time instant, time point A (one-dimensional) chronon is a nondecomposable time interval of some fixed, minimal duration. An n -dimensional chronon is a nondecomposable region in n -dimensional time. Important special types of chronons include valid-time, transaction-time, and bitemporal chronons. This article uses *time instant* and *time point* as synonyms to chronon, and, in keeping with the literature on which the article is based, we generally prefer the latter term.

snapshot equivalent Two tuples are snapshot equivalent if the snapshots of the tuples at all times are identical. Similarly, two relations are snapshot equivalent if at every instant their snapshots are equal.

snapshot relation Relations of a conventional relational database system incorporating neither valid-time nor transaction-time timestamps.

span A directed duration of time. A duration is an amount of time with known length, but no specific starting or ending instants. For example, the duration "one week" is known to have a length of seven

days, but can refer to any block of seven consecutive days. A span is either positive, denoting forward motion of time, or negative, denoting backward motion in time.

temporal element A finite union of n -dimensional time intervals. Special cases of temporal elements include valid-time elements, transaction-time elements, and bitemporal elements. They are finite unions of valid-time intervals, transaction-time intervals, and bitemporal intervals, respectively.

temporal expression A syntactic construct used, e.g., in a query that evaluates to a temporal value, i.e., an instant, a span, a time interval, or a temporal element.

time interval Two different definitions are used in the literature. In the first definition, it is the same as a time period. In the second definition, which is used in SQL standards, it is the same as a span. This article generally uses the first definition and resorts to the second in only a few situations where predefined query syntax requires this.

time period The time between two instants. In a system that models a time domain using chronons, a time period may be represented by a set of contiguous chronons.

timestamp A time value associated with some object, e.g., an attribute value or a tuple. The concept may be specialized to valid timestamp, transaction timestamp, instant timestamp, span timestamp, interval timestamp, bitemporal-element timestamp, etc.

transaction time A database fact is stored in a database at some time instant, and after it is stored, it is current until the instant it is logically deleted. The transaction time of a database fact is the time

when the fact is current in the database and may be retrieved. Transaction times are consistent with the serialization order of the transactions. They cannot extent into the future. Also, as it is impossible to change the past, (past) transaction times cannot be changed. Transaction times are system-generated and -supplied. While valid times may only be associated with "facts," statements that can be true or false, transaction times may be associated with any database object.

transaction-time relation A relation with exactly one system-supported transaction time. As for valid-time relations, there are no restrictions as to how transaction times may be incorporated into the tuples.

valid time The time when the fact is true in the modeled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user.

valid-time relation A relation with exactly one system-supported valid time. There are no restrictions on how valid times may be incorporated into the tuples; e.g., the valid times may be incorporated by including one or more additional valid-time attributes in the relation schema, or by including the valid times as a component of the values of the application-specific attributes.

I. INTRODUCTION

A wide range of database applications manage time-varying data. The area of temporal databases characterizes the semantics of temporal data and provides expressive and efficient ways to model, store, and query temporal data. This article introduces the reader to temporal data models and query languages. It concisely introduces fundamental temporal database concepts, surveys state-of-the-art solutions to challenging aspects of temporal data management, and offers a look into the future of temporal databases.

Indeed, most applications of database technology are temporal in nature. Examples include financial applications such as portfolio management, accounting, and banking; record-keeping applications such as personnel, medical-record, and inventory management; scheduling applications such as airline, train, and hotel reservations and project management; and scientific applications such as weather monitoring. Applications such as these rely on *temporal databases*, which record time-referenced data.

In contrast, the support for temporal data management offered by commercially available software

tools and systems is quite limited and leaves much to be desired. As a result, temporal database management is a vibrant field of research, with an active community of several hundred researchers who have produced some 2000 papers over the last two decades. Many of these papers are listed in a series of seven cumulative bibliographies. The newest, listed in the bibliography, provides pointers to its predecessors. The field has produced a comprehensive glossary of terminology (the glossary at the outset of this article borrows from this glossary), an edited volume which captures state-of-the-art circa 1993, and three workshop proceedings. These are all listed in the bibliography. The topic of temporal databases is now also covered in many textbooks, and Snodgrass has written the first book dedicated entirely to temporal databases (again, listed in the bibliography).

This article provides an introduction, readable by the nonexpert, to central concepts and issues in temporal data models and query languages. More specifically, the article examines in turn the semantics of temporal data, temporal data models, and temporal query languages, which are central areas of temporal database management. Each area is first motivated, and then sample contributions are surveyed, to give the reader a feel for the challenges and issues that are faced in each particular area. The article concludes with a summary and a look into the possible futures of temporal databases.

The presentation is by no means complete in its coverage of areas, let alone contributions within areas. The article does not cover the issues related to the representation of the time domain such as whether time is assumed to be continuous or discrete. Further, the article does not cover the many issues related to the support for multiple time granularities (e.g., day, week, month, and year), calendars (e.g., national, solar, lunar, financial, and business calendars), indeterminacy (e.g., from between 14.00 and 15.00 until between 21.00 and 22.00), and the unique temporal notion *now*. Finally, the article focuses on the recording of phenomena that have a duration in time. It largely avoids issues specific to the recording of instantaneous events and also time series.

The bibliography contains works that either survey the area or are compilations of research contributions.

II. TEMPORAL DATA SEMANTICS

Before we proceed to consider temporal data models and query languages, we examine, in data model-independent terms, the association of times and facts, which is at the core of temporal data management.

Initially, a brief description of terminology is in order. A database models and records information about a part of reality, termed either the *modeled reality* or the *mini-world*. Aspects of the mini-world are represented in the database by a variety of structures that we will simply term *database entities*. In general, times are associated with database entities. We will employ the term “fact” for any (logical) statement that can meaningfully be assigned a truth value, i.e., that is either true or false.

The facts recorded by the database entities are of fundamental interest. Several different temporal aspects may be associated with these. Most importantly, the *valid time* of a fact is the collected times—possibly spanning the past, present, and future—when the fact is true in the mini-world. Valid time thus captures the time-varying states of the mini-world. All facts have a valid time by definition. However, the valid time of a fact may not necessarily be recorded in the database, for any of a number of reasons. For example, the valid time may not be known, or recording it may not be relevant for the applications supported by the database. If a database models different possible worlds, the database facts may have several valid times, one for each such world.

Next, the *transaction time* of a database entity is the time when the entity is current in the database. Like valid time, this is an important temporal aspect. Transaction time is the basis for supporting accountability and “traceability” requirements, which exist in many applications, e.g., financial and medical applications.

Note that transaction time, unlike valid time, may be associated with any database entity, not only with facts. For example, transaction time may be associated with objects and values that are not facts because they cannot be true or false in isolation. To be more concrete, the value “5” may be stored in a database, but does not denote a logical statement. It is meaningful to associate transaction time with 5, but not valid time. Thus, all database entities have a transaction-time aspect. This aspect may or may not, at the database designer’s discretion, be captured in the database. This is dependent on the requirements posed by the applications to be supported.

The transaction-time aspect of a database entity has a duration: from insertion to deletion, with multiple insertions and deletions being possible for the same entity. As a consequence of the semantics of transaction time, capturing this aspect of database entities renders deletions purely logical. Deleting an entity does not physically remove the entity from the database; rather, the entity remains in the database, but ceases to be part of the database’s current state.

Transaction time captures the time-varying states of the database, and, as mentioned, applications that demand accountability or “traceability” may use databases that record transaction time.

Observe that the transaction time of a database fact, say F , is the valid time of the related fact, F is *current in the database*. This would indicate that supporting transaction time as a separate aspect is redundant. However, both valid and transaction time are aspects of the content of all databases, and recording both of these is essential in a wide range of applications. In addition, transaction time, due to its special semantics, is particularly well behaved and may be supplied automatically by the database management system (DBMS). Specifically, the transaction times of facts stored in the database are bounded by the time the database was created at one end of the time line and by the current time at the other end. This provides the rationale for the focus of most temporal database research on providing improved support for valid time and transaction time as separate aspects.

In addition, some other times have been considered, e.g., decision time. But the desirability of building decision time support into temporal database technologies is less clear, because the number and meaning of “the decision time(s)” of a fact varies from application to application and because decision times, unlike transaction time, generally do not exhibit specialized properties.

The valid and transaction time values of database entities are drawn from some appropriate time domain. There is no single answer as to how to perceive time in reality and how to represent time in a database, and different time domains may be distinguished with respect to several orthogonal characteristics. First, the time domain may or may not stretch infinitely into the past and future. Second, time may be perceived as discrete, dense, or continuous. Some feel that time is really continuous; others contend that time is discrete and that continuity is just a convenient abstraction that makes it easier to reason mathematically about certain discrete phenomena. In databases, a finite and discrete time domain is typically assumed, e.g., in the standard query language (SQL) standards. Third, a variety of different structures have been imposed on time. Most often, time is assumed to be totally ordered, but various partial orders have also been suggested, as has cyclic time.

An aspect of time that has been intriguing philosophers for centuries and that is difficult to describe fully is the concept of the current time, which we term *now*. This concept is unique to time; indeed, there really does not exist any other notion quite like it.

Among its properties, the current time is ever-increasing, all activity is trapped at the current time, and the current time separates the past from the future. The spatial equivalent, *here*, simply fails to enjoy the properties of *now*. The uniqueness of *now* is one of the reasons why techniques from other research areas are not readily, or not at all, applicable to temporal data.

Much research has been conducted on the semantics and representation of time, from quite theoretical topics, such as temporal logic and infinite periodic time sequences, to more applied questions such as how to represent time values in minimal space. Substantial research has been conducted that concerns the use of different time granularities and calendars in general, as well as the issues surrounding the support for indeterminate time values. Also, there is a significant body of research on time data types, e.g., time instants, time intervals (or "periods"), and temporal elements.

Temporal data management can be very difficult using conventional (nontemporal) data models and query languages. Accommodating the time-varying nature of the enterprise is largely left to the developers of database applications, leading to ineffective and inefficient ad-hoc solutions that must be reinvented each time a new application is developed. The result is that data management is currently an excessively involved and error-prone activity. Temporal database research has produced several proposals for temporal data models and query languages to remedy this situation. Section III considers temporal data models, and Section IV then covers query languages based on these models.

III. TEMPORAL DATA MODELS

A first step in providing support for temporal data management is to extend the database structures of the data models supported by a conventional DBMS. Assuming a relational data model, mechanisms must be provided for capturing the times of the facts recorded by the relations, leading to temporal relations. Throughout, we focus on the valid-time aspect of the facts.

More than two dozen temporally extended relational data models have been proposed. We consider the main types of models that have been proposed. In order to exemplify these, consider a video store where customers, identified by a *CustID* attribute, rent video tapes, identified by a *TapeNum* attribute. We consider a few rentals during May 1997. On the 3rd of May, customer C101 rents tape T1234 for three days. On the

5th, customer C102 rents tape T1245. The tape is returned on the 7th. From the 9th to the 12th, customer C102 rents tape T1234. Customer C102 rents tape T1245 from the 19th to the 20th and again from the 21st to the 22nd. The video store keeps a record of these rentals in a relation termed *CheckOut*.

A. Points

The most basic type of temporal data model uses tuple timestamps that are instants, or *points*. Under this view, each fact is associated with a time point that indicates when the fact is valid. Multiple tuples are used if a fact is valid at several points in time. The resulting *CheckOut* instance is illustrated in Fig. 1.

A distinguishing feature of this approach is that (syntactically) different relations have different information content. Next, timestamps are atomic values that can be easily compared. Assuming a totally ordered time domain, the standard set of comparison predicates, =, ≠, <, >, ≤, and ≥, is sufficient to conveniently compare timestamps. As we will see in the next section, this simplicity is lost when moving to more compact representations of time, which has the effect of making some conceptually simple queries cumbersome to formulate. The conceptual simplicity of time points comes at a cost, though. In general, it is impossible to reconstruct the original intervals from the points. Additional attributes, such as the *SeqNo* attribute in *CheckOut* in Fig. 1, must be used for this purpose. The reader can easily verify that it becomes impossible to represent the 2- and 4-day rentals described in the running example if the *SeqNo* attribute is omitted.

SeqNo	CustID	TapeNum	T
1	C101	T1234	3
1	C101	T1234	4
1	C101	T1234	5
2	C102	T1245	5
2	C102	T1245	6
2	C102	T1245	7
3	C102	T1234	9
3	C102	T1234	10
3	C102	T1234	11
3	C102	T1234	12
4	C102	T1245	19
4	C102	T1245	20
5	C102	T1245	21
5	C102	T1245	22

Figure 1 Snapshots of the *CheckOut* instance at times 4(a), 5(b), and 8(c).

It is important to note that the point model is an *abstract* view of a temporal database and is not meant for physical representation. For all but the most trivial time domains, the space needed when using the point model is prohibitive.

The combination of conceptual simplicity and computational complexity has made the point model popular for theoretical studies while there is a lack of implementations. Only recently have some implementation issues been considered.

B. Intervals

From an implementation perspective, the most popular type of data model is the one that uses time *intervals*. This type of model associates each fact with an interval that captures the valid time of the fact. Multiple tuples are needed if a fact is valid over disjoint intervals. Figure 2 illustrates the approach.

When employing intervals rather than time points as timestamps, two timestamps satisfy precisely one of the following thirteen relationships, first enumerated by James Allen: before, meets, overlaps, during, starts, finishes, and equal, in addition to the inverses of the first six of these relationships. Allen’s pioneering work in this area has inspired designs of many of the collections of interval predicates available in temporal query languages. While well-chosen interval predicates are more convenient to use than relationships over interval start and end points, such predicates alone turn out to not be sufficient to provide comprehensive and easy-to-use support for temporal data management in general and the point-based view in particular (see Section IV.A).

The notion of *snapshot equivalence*, which reflects a point-based view of data, establishes a correspondence between the point based and interval based models. Consider the relations in Fig. 3. The relations are different, but snapshot equivalent. Specifically, the relation in Fig. 3b is a coalesced version of the relation in Fig. 3a. (In *coalescing*, tuples with the same non-timestamp values, termed value-equivalent tuples, and adjacent or overlapping time intervals are merged.)

SeqNo	CustID	TapeNum	T
1	C101	T1234	[3,5]
2	C102	T1245	[5,7]
3	C102	T1234	[9,12]
4	C102	T1245	[19,20]
5	C102	T1245	[21,22]

Figure 2 An interval model of the CheckOut instance.

a			b		
CustID	TapeNum	T	CustID	TapeNum	T
C101	T1234	[3,5]	C101	T1234	[3,5]
C102	T1245	[5,7]	C102	T1245	[5,7]
C102	T1234	[9,12]	C102	T1234	[9,12]
C102	T1245	[19,20]	C102	T1245	[19,22]
C102	T1245	[21,22]			

Figure 3 Different but snapshot equivalent CheckOut instances.

Clearly, the two relations are different. However, in some temporal data models, the relations are taken to contain the exact same information because they contain the same snapshots, some of which are illustrated in Fig. 4. These models adopt a point-based view and are only interval-based in the weak sense that they use time intervals as convenient representations of (contiguous) sets of time points.

It then also makes sense for such models to require that their relation instances be coalesced. This requirement ensures that relation instances that are syntactically different are also semantically different, and vice versa.

In our example, the requirement rules out Instance 1 in Fig. 3. This implies that it is not possible to distinguish two consecutive 2-day check-outs from one four-day check-out, unless an additional attribute is used to uniquely identify check-outs. For such weakly interval-based models, the CheckOut instance in Fig. 2 is thus the appropriate representation of the rental scenario.

In an inherently interval-based model, intervals are not merely representational devices—they carry meaning beyond denoting a set of points. Returning to the example, it is easy to imagine a scenario where the average rental price per day decreases as the check-out span increases. It then makes a difference whether a customer checks out a tape for two short, consecutive intervals (see Fig. 3a) or for one longer interval (see Fig. 3b). Interval-based models do not enforce coalescing and capture this distinction naturally and are in this sense more expressive than point models. They require no additional attribute to identify the check-outs, and the CheckOut instance in Fig. 3a is the appropriate representation of the rental scenario.

C. Temporal Elements

A frequently mentioned shortcoming of intervals is that they are not closed under all set operations. For example, when subtracting the interval [5,7] from the

Figure 4 Snapshots of the CheckOut instance.

interval [1,9], we do not get an interval, but a set of intervals: {[1,4], [8,9]}. This has led to the proposal that *temporal elements* be used as timestamps instead. These are finite unions of intervals.

In the definition, the unions are restricted to be finite in order to ensure that temporal elements can be represented conveniently in a database. It should also be noted that the definition is meaningful for both a discrete and a continuous time domain. In this article, we generally assume a discrete and bounded time domain, as does probably most data model research.

With temporal elements, the same two semantics as for intervals are possible, although models that use temporal elements seem to prefer the point-based semantics. Figure 5a uses temporal elements to capture the rental scenario assuming the interval-based semantics. Under this view, intervals are perceived as atomic units. Under the point-based view, an interval is identical to a set of points. This implies that the instance in Fig. 5a is equivalent to the one shown in Fig. 5b, as the tuples of the two relations then have timestamps that denote the same time points. Under this assumption (and recalling our assumption about the time domain), “a (finite) set of time points” is an equivalent definition of a temporal element.

We consider next the bitemporal conceptual data model (BCDM), which has been used to TSQ_{L2}, to be discussed in Section IV.F. This model adopts the point-based view. So the SeqNo attribute is introduced in Fig. 6, which shows the BCDM relation instance that captures the rental scenario.

The BCDM timestamps facts with values that are sets of time points. (We used the valid-time variant of the BCDM.) This is equivalent to temporal elements because the BCDM adopts a discrete and bounded time domain.

Because value-equivalent tuples are not allowed (this corresponds to the enforcement of coalesced re-

lations as discussed in the previous section), the full history of a fact is contained in exactly one tuple, and one tuple contains the full history of exactly one fact. In addition, relation instances that are syntactically different have different information content, and vice versa. This design decision reflects the point-based underpinnings of the BCDM.

Note that the BCDM is intended for the conceptual level and as the basis for a query language. When it comes to the physical representation and the display to users of temporal information, the BCDM is impractical. The varying length and voluminous timestamps of tuples are impractical to manage directly, and the timestamp values are hard to comprehend.

D. Attribute Value Timestamping

With temporal elements, the full history of a fact is contained in a single tuple. The information in a relation that pertains to some real-world object may still be spread across several tuples, though. To capture all information about a real-world object in a single tuple, attribute value timestamping has been introduced. We use the parametric model of Gadia and Nair for illustration. This is a typical non-first-normal-form data model. The name of the model derives from the view that a temporal database is a time-varying nontemporal database. In other words, a temporal database is a nontemporal database that is time parameterized.

The relation in Fig. 7 is thought of as recording information about some types of objects. It records information about customers and thus holds one tuple for each customer in the example, with a tuple containing all information about a customer. An obvious consequence is that the information about a single tape cannot be contained in a single tuple. Another observation is that a single tuple may record multiple

a

CustID	TapeNum	T
C101	T1234	[3, 5]
C102	T1245	[5, 7] ∪ [19, 20] ∪ [21, 22]
C102	T1234	[9, 12]

b

CustID	TapeNum	T
C101	T1234	[3, 5]
C102	T1245	[5, 7] ∪ [19, 22]
C102	T1234	[9, 12]

Figure 5 Temporal element representation of the CheckOut instance.

SeqNo	CustID	TapeNum	T
1	C101	T1234	{3, 4, 5}
2	C102	T1245	{5, 6, 7}
3	C102	T1234	{9, 10, 11, 12}
4	C102	T1245	{19, 20}
5	C102	T1245	{21, 22}

Figure 6 Using the BCDM to model the CheckOut instance.

facts. In the example, the second tuple records multiple facts: rental information for customer C102 for the two tapes, T1245 and T1234, and four different check-outs.

It should also be noted that different groupings of the information into tuples are possible for this attribute-value timestamping model. For example, Fig. 8 illustrates a relation instance that is snapshot equivalent to the one in Fig. 7. This new instance is “grouped on” the TapeNum attribute, indicating that it is now the tapes, not the customers, that are the objects in focus. (Section IV.H elaborates on this.)

IV. TEMPORAL QUERY LANGUAGES

This section illustrates and discusses central query language concepts that have emerged after years of temporal database research and are representative for the field. To be specific, these concepts are discussed in the contexts of specific query languages that are all extensions to variants of the SQL query language. Because the aim is to communicate the essential principles, we gloss over semantic variations among temporal constants and predicates (e.g., overlaps), and introduce additional functions (e.g., duration) as needed.

We first motivate the need for temporal query languages, then extend the video rental example for use in this section. This is followed by discussions of six different approaches to temporal query languages.

A. Motivation

Given the prevalence of applications that currently manage time-varying data, one might question the need for a temporal query language. Is the existence of all this SQL code of ostensibly temporal applications not proof that SQL is sufficient for writing such applications? The reality is that in conventional query languages like SQL, temporal queries *can* be expressed, but often only with great difficulty.

To illustrate the issue, we use the relations CheckOut in Fig. 2, which records check-out intervals for video rentals, and CheckOutSnap in Fig. 4b, which records the check-outs at time 5. Using SQL, it is straightforward to express the number of check-outs from CheckOutSnap. For example, this can be expressed as follows.

```
SELECT COUNT (TapeNum) AS Cnt FROM
CheckOutSnap
```

The temporal generalization of this query, asking now for the time-varying count of tapes checked out, as recorded in relation CheckOut, is nontrivial. The reader may verify that the result in Fig. 9 is correct. Although possible, expressing this query in SQL is quite difficult. The authors are aware of one stepwise and systematic solution that consists of six steps and takes up 35 lines of complex SQL code.

As another example, specifying a key constraint on relation CheckOutSnap is trivial in SQL.

```
ALTER TABLE CheckOutSnap ADD PRIMARY
KEY (TapeNum)
```

This key constraint may be generalized to apply to a valid-time relation, now meaning that TapeNum is a key at each point in time or, equivalently, is a key in each snapshot that may be produced from the valid-time relation. Specifying this constraint on relation CheckOut in SQL is again difficult. Expressed as an SQL assertion, it may easily take more than 10 lines.

The lesson learned is that quite reasonable queries can be difficult to express on temporal relations using a query language such as SQL. Even SQL experts

SeqNo	CustID	TapeNum
[3, 5] 1	[3, 5] C101	[3, 5] T1234
[5, 7] 2	[5, 7] $\cup$ [9, 12] $\cup$ [19, 22] C102	[5, 7] $\cup$ [19, 22] T1245
[9, 12] 3		[9, 12] T1234
[19, 20] 4		
[21, 22] 5		

Figure 7 Parametric representation of the CheckOut instance.

SeqNo		CustID	TapeNum
[3, 5]	1	[3, 5] C101	[3, 5] $\cup$ [9, 12] T1234
[9, 12]	3	[9, 12] C102	
[5, 7]	2	[5, 7] $\cup$ [19, 22] C102	[5, 7] $\cup$ [19, 22] T1245
[19, 20]	4		
[21, 22]	5		

Figure 8 Parametric representation of the CheckOut instance.

would be hard pressed to express the examples above in SQL. Given also the ubiquitous nature of temporal information, this indicates a strong need for temporal support beyond what SQL offers today.

B. Video Rental Example

The following queries together with their intended results build on the video rental example and will serve for illustration in the subsequent sections.

Query *Q1* asks for all available information about check-outs that overlap the interval [7,9]. Returning the nontemporal information of a check-out is straightforward. Returning the temporal information is more subtle, as we will see Section IV.E.

Q1 All check-outs that overlap the time interval [7,9].

C102	T1245	[5,7]
C102	T1234	[9,12]

Query *Q2* constrains the number of time points included in a time interval. This query teases out the difference between time points and intervals. Only if a model gives an interval a meaning beyond a set of time points or if auxiliary attributes are used will it be possible to distinguish a four-day check-out from two consecutive two-day check-outs.

Q2 All 2-day check-outs.

T1245	[19,20]
T1245	[21,22]

Cnt	T
0	$[-\infty, 2]$
1	[3,4]
2	[5,5]
1	[6,7]
0	[8,8]
1	[9,12]
0	[13,18]
1	[19,20]
1	[21,22]
0	[23, ∞]

Figure 9 A time-varying count.

Query *Q3* is an ordinary query that, conceptually, must be applied to each state of a temporal database. In this case the nontemporal query is a (self-)Cartesian product.

Q3 Which tapes were checked out together with tape T1234?

T1245	[5,5]
-------	-------

Query *Q4* is another example of an ordinary query that must be applied to each state of a temporal database. The only difference with respect to query *Q3* is the actual nontemporal query. In this case the nontemporal query is an aggregation.

Q4 What is the number of tapes that have been check out?

See Fig. 9.

Like Query *Q4*, Query *Q5* is an aggregation. However, the aggregation is to be applied independently of any temporal information. Instead of a time-varying count of the check-outs, we want to determine the total number of check-outs.

Q5 How many check-outs were made in total?

5

The final query refers to the (constantly moving) current time. (We assume that the current time is 5.)

Q6 List all (current) check-outs.

C101	T1234
C102	T1245

This type of query can come from two sources. First, applications that are time-aware might ask for the current status of the check-outs. Second, time-unaware applications might ask it as a legacy query.

C. Approach I: Abstract Data Types

The earliest and, from a language design perspective, simplest approaches to improving the temporal data management capabilities of query languages have simply introduced time data types and associated predicates and functions. This approach is attractive because it has limited impact on the language, e.g., SQL, that it is applied to and because the addition of new data types, predicates, and functions is fairly well understood. To illustrate this approach, we assume that the video rental data is represented by the interval-timestamped CheckOut instance in Fig. 3a.

Formulations of predicates on time-interval data types have been influenced by Allen's 13 interval relationships: any two intervals are related by precisely

one of these. With reference to these, different sets of practical proposals for predicates have been proposed. For example, the overlaps predicate can be used to formulate query $Q1$.

```
 $Q1^{SQL}$ : select * from CheckOut
      where T overlaps [7,9]
```

The result of this query depends on how the rental information is represented—see Fig. 3, which illustrates that the same information (from a point-based view) may be represented in more than one way.

A selection that limits the duration of an interval follows the same approach:

```
 $Q2^{SQL}$ : select TapeNum, T
      from CheckOut where duration(T) = 2
```

The third query can be expressed using selection, projection, and join. Its formulation is simplified if predicates and functions on intervals are available:

```
 $Q3^{SQL}$ : select b.TapeNum,
      intersect(a.T, b.T) as T
      from CheckOut as a,
           CheckOut as b
      where a.TapeNum = T1234
            and b.TapeNum <> T1234
            and a.T overlaps b.T
```

As mentioned in Section IV.A, expressing a time-varying aggregation is possible, but exceedingly complicated. There exists no reasonable SQL solution for counting the number of check-outs at any point in time ($Q4$).

In contrast, counting the check-outs independent of the temporal references is easily possible:

```
 $Q5^{SQL}$ : select count (*)
      from CheckOut
```

Retrieving the current check-outs is stated as follows:

```
 $Q6^{SQL}$ : select CustID, TapeNum
      from CheckOut
      where T overlaps [now]
```

In summary, the availability of appropriate time data types aids in the formulation of several of the example queries considered, but falls short when it comes to formulate temporal aggregation queries such as $Q4$.

Let us consider this point in a bit more detail, as it indicates the limits of the abstract data type approach and motivates the consideration of other approaches. Queries $Q3$ and $Q4$ are related in the sense that, conceptually, both generalize a simple snapshot query to becoming time-varying, i.e., to be applied to all states of the database. The only difference is in the snapshot query, a join ($Q3$) versus an aggregation ($Q4$). In spite of their close relationship, the temporally extended

language offers no systematic way of formulating the two queries when starting from their snapshot formulations. The techniques that are applied to formulate $Q3$ may not be reused when formulating $Q4$ and vice versa. Rather, new solutions must be thought out for each new kind of query. So although the availability of abstract data types for time do aid in formulating temporal queries, they fall short in offering means of conveniently formulating a wide range of queries on interval-timestamped temporal relations such as the time-varying aggregation and the primary key constraint mentioned in Section IV.A. Indeed, more than 20 years of temporal database research has not produced an abstract data type that allows one to conveniently express temporal queries.

D. Approach II: Folding and Unfolding

Being of fixed size, interval timestamps are very convenient when representing the temporal aspects of information. In some respects, the most straightforward and simplest means of capturing temporal aspects is by means of an extra interval timestamped attribute in each relation. On the other hand, one might suspect that the difficulty in formulating temporal queries in the previous section is caused by the intervals. SQL comes unprepared to support something (intervals) that represent something (sets of consecutive time points) that they are not.

In response to this, it has been proposed to equip SQL with the ability to *normalize* timestamps. The idea is to split or merge interval timestamps so that they are *aligned* (identical or disjoint) and can be treated as atomic entities.

Advanced most prominently by Lorentzos, the earliest and most radical approach is to introduce the two function *unfold* and *fold*. The unfold function decomposes an interval timestamped tuple into a set of point timestamped tuples, one for each point in the original interval. The fold function “collapses” a set of point timestamped tuples into value-equivalent tuples timestamped with maximum intervals. Thus, unfolding the relations in Fig. 3 yields the relation in Fig. 1 (modulo the sequence number attribute), which may be folded to yield the relation in Fig. 3b. The idea is to use the interval-based representation of temporal information while being able to manipulate it as if the point-based representation was used, thus obtaining the representational benefits of intervals while avoiding the problems they seem to pose in query formulation.

The fold and unfold functions have been integrated into IXSQL, which we use for illustration. IXSQL inherits and extends the semantics of SQL. Thus, each

SQL query statement is also an IXSQL query statement. For instance the IXSQL formulations of the first three queries are essentially those given for SQL in the previous section. (There are minor differences in the syntax of temporal predicates and functions, which are not relevant for our discussion. For example, *cp* is an IXSQL predicate that is true if the argument intervals have common (time) points. It corresponds to the overlaps predicate used in Section IV.C.) We assume the CheckOut relation in Fig. 3a.

```

Q1IXSQL: select * from CheckOut
  where T cp [7,9]
Q2IXSQL: select TapeNum, T
  from CheckOut
  where dur(T) = 2
Q3IXSQL: select b. TapeNum,
  intervsect (a.T, b.T) as T
  from CheckOut as a,
  CheckOut as b
  where a.TapeNum = T1234
  and b. TapeNum <> T1234
  and a.T cp b.T

```

The fold and unfold functions become useful when we consider temporal aggregation. Here, unfold is used to explicitly convert the interval-based representation into its point-based representation; then the point-based representation is used in query formulation; and, the final interval-based result is produced by applying fold. Query *Q4* illustrates this:

```

Q4IXSQL: select count (*), T
  from (select *
  from CheckOut
  reformat as unfold T)
  as a
  group by T
  reformat as fold T

```

Thus, the general pattern for queries using unfold and fold is to (1) explicitly construct the point-based representation by unfolding the argument relation(s), (2) compute the query on interval-free representation, and (3) fold the result to end up with an interval-based representation.

The standard SQL solution can be used to count the check-outs independently of time, and to retrieve the current state of the CheckOut relation.

```

Q5IXSQL: select count (*)
  from CheckOut
Q6IXSQL: select CustID, TapeNum
  from CheckOut
  where T overlaps [now]

```

In summary, a language enriched with folding and

unfolding may offer a systematic approach to formulating temporal queries, in particular temporal queries that generalize nontemporal queries.

The efficient evaluation of queries formulated using fold and unfold has yet to be resolved. Unfolding has a worst case exponential space complexity; and for the time domains available in current systems, unfolded relations are so large that storing them is impractical.

A more subtle observation is that IXSQL adopts a view on relations instances that is neither purely point-based nor interval-based. It is not purely point-based because it is sensitive to the specific interval representation chosen for the data. Thus when different, but snapshot-equivalent, relations are used, the same query generally returns different results. In contrast, the fold and unfold functions only preserve the information content in a relation up to that captured by the point-based view. For example, unfolding and then folding the relation instance in Fig. 3a yields the instance in Fig. 3b.

E. Approach III: Point Timestamps

A more radical approach to designing a temporal query language is to simply assume that temporal relations use point timestamps—fold and unfold are then not needed. The temporal query language SQL/TP advanced by Toman takes this approach to generalizing queries on nontemporal relations to apply to temporal relations. The semantics of SQL/TP is defined with respect to the point-based representation, and we thus assume the CheckOut relation instance in Fig. 1 in the following. The restriction to point timestamps yields a simple and unambiguous semantics that avoids many of the pitfalls that can be attributed to interval timestamps.

Q1 calls for a comparison of neighbor database states. The point view, which separates the database states, does not easily support this type of query. The straightforward approach is to use a disjunction of equality predicates: *select CustID, TapeNum from CheckOut where T = 7 or T = 8 or T = 9*. This query does not yield the intended result. The query restricts the time of the original check-out interval to the time points 7, 8, and 9, and it fails to report the original check-out intervals as intended. A more complicated query that involves a join is needed.

```

Q1SQL/TP: select a.*
  from CheckOut a,
  CheckOut b
  where a.SeqNo = b.SeqNo

```

```

and a.CustID = b.CustID
and a.TapeNum = b.TapeNum
and (b.T = 7 or b.T = 8
    or b.T = 9)

```

In SQL/TP, duration queries are formulated as aggregate queries. (Remember that the semantics of queries is defined with respect to the point representation.)

```

Q2SQL/TP: select SeqNo, TapeNum, T
  from CheckOut
 group by SeqNo
 having count(T) = 2

```

As explained before this query can only be answered if an attribute is available to distinguish individual check-outs. Note that if this additional attribute is omitted from the final projection, the two-day intervals will generally not follow from the results. The result might consist of four tuples timestamped with time points, one tuple timestamped with an interval spanning four time points, or two tuples with intervals spanning two time points.

The strength of SQL/TP is in its generalization of queries on snapshot relations to corresponding queries on corresponding temporal relations. The general principle is to extend the snapshot query with equality constraints on the timestamp attribute of the temporal relation, to separate different database snapshots during query evaluation.

```

Q3SQL/TP: select b.TapeNum, b.T
  from CheckOut as a,
       CheckOut as b
 where a.TapeNum = T1234
       and b.TapeNum <> T1234
       and a.T = b.T

```

```

Q4SQL/TP: select count(*), T
  from CheckOut
 group by T

```

In the first query we use a self-join to determine the time points when tape T1234 was checked out together with another tape. In the second query, the grouping takes care of isolating the database states from each other. The second query is restricted to bounded time domains, to avoid infinite relations.

In a sense, SQL/TP and SQL are opposites when it comes to the handling of temporal information. In SQL, intervals have no special meaning—they are treated as atomic entities. On the other hand, SQL/TP effectively decomposes intervals into sets of points. This difference becomes clear when considering aggregate queries. In SQL, time-varying aggregation (Q4) is poorly supported, while SQL/TP needs an ad-

ditional attribute in the argument relation to support time-invariant aggregation (Q5). Note that the SQL/TP query `select count(*) from CheckOut` counts the number of time points (possibly multiple times) when tapes were checked out and thus does not give the intended result. However, if an attribute is available that identifies the real-world facts, counting the facts is possible:

```

Q5SQL/TP: select count(distinct SeqNo)
  from CheckOut

```

The current state of the CheckOut relation is retrieved as follows:

```

Q6SQL/TP: select CustID, TapeNum
  from CheckOut
 where T = now

```

In several of the examples, we have used relations with sequence numbers in order to be able to capture the intended information and express the desired queries. While the introduction of sequence numbers appears to be a minor issue, it is worth noting that these do not offer a systematic approach to obtaining point-based semantics *and* a semantics that preserves the intervals of the argument relations. The query *When was tape T1245, but not tape T1234, checked out?* may serve to illustrate this point. The expected answer is the intervals {[6,7], [9,12], [19,20], [21]22]}. The following formulation, using a temporal difference, seems plausible.

```

select T from CheckOut
 where TapeNum = T1245
 except
 select T from CheckOut
 where TapeNum = T1234

```

This query does not guarantee that fragments of individual check-out intervals are returned because the sequence numbers that are used to identify check-outs are not included. On the other hand, if we add the sequence number attribute to the select lists, the temporal difference will effectively be disabled, and the returned result would simply be the intervals of the original check-outs of tape T1245. The problem here is that set operations are sensitive to any additional attributes and essentially do not permit such attributes. This issue is not germane to SQL/TP, but applies equally to all approaches that use a point-based data model.

In summary, the strength of SQL/TP is the restriction to time points to ensure a simple, unambiguous, and well-defined semantics. As intervals are still to be used in the physical representation of the temporal information as well as when presenting the results of

queries to the users, one may think of SQL/TP as a variant of IXSQL where, conceptually, queries must always apply unfold as the first operation and fold as the last. A compilation technique has been supplied for SQL/TP that avoids this unfolding, thus offering hope that SQL/TP queries can be evaluated efficiently in practice.

F. Approach IV: Syntactic Defaults

Along with the introduction of temporal abstract data types, what may be termed *syntactic defaults* have been introduced that make the formulation of common temporal queries more convenient. The most common defaults concern access to the current state of a temporal database and for handling temporal generalizations of nontemporal queries, e.g., joins. The most comprehensive approach based on syntactic defaults is the TSQL2 language, which we use for exemplification. We assume the CheckOut instance in Fig. 2.

In TSQL2, a default valid clause, placed after the select clause, computes the intersection of the valid times of the argument relations mentioned in the from clause, which is then returned in the result. With only one relation in the from clause this yields the original timestamps.

```
Q1TSQL2: select * from CheckOut
        where valid(T) overlaps
              period '7-9'
```

The cast function is available to map between intervals and time spans. (Note the terminology: in TSQL2, a period (e.g., [7-9]) is an anchored interval, and an interval (e.g., 4 days) is an unanchored time span.)

```
Q2TSQL2: select SeqNo, TapeNum, T
        from CheckOut
        where cast(valid(T) as
              interval) = 2
```

Note that the returned result must include the SeqNo attribute in its projection list to properly show the two-day check-outs (see Section IV.E).

The default for the default behavior of the valid clause was designed with snapshot reducibility in mind. This shows nicely if we consider the snapshot reducible join.

```
Q3TSQL2: select b.TapeNum
        from CheckOut as a,
             CheckOut as b
        where a.TapeNum = T1234
              and b.TapeNum <> T1234
```

The implicit default valid clause is here `valid intersect(valid(a), valid(b))`. If two tuples have valid times that do not overlap, they do not contribute to the result. Tuples with empty timestamps are not allowed in TSQL2's data model.

```
Q4TSQL2: select COUNT(*)
        from CheckOut
Q5TSQL2: select snapshot COUNT(*)
        from CheckOut
```

The default behavior of TSQL2 is to return temporal relations. The snapshot keyword is used to retrieve nontemporal relations. Thus, to retrieve the current state of the CheckOut instance, the following query is issued:

```
Q6TSQL2: select snapshot *
        from CheckOut
        where valid(CheckOut)
              overlaps [now]
```

TSQL2 is a large language with many parts and an informally specified semantics. It provides syntactic defaults that simplify the formulation of temporal queries over point-based temporal databases. The problem with syntactic defaults relates to lack of the "scalability" over language constructs. When defining a language that uses syntactic defaults, one must explicitly specify a large number of defaults. It becomes challenging to be comprehensive and systematic in the specification of such defaults, and to ensure that the defaults do not interact in unanticipated ways. This approach tends to yield a language where, although it may be possible to formulate common queries concisely, the language itself is complex and therefore difficult to understand and use.

G. Approach V: Semantic Defaults

ATSQL introduces temporal statement modifiers to add temporal support to SQL. In contrast to syntactic defaults, statement modifiers are *semantic defaults* that indicate the intended semantics without specifying how to compute it.

The basic idea in statement modifiers is to offer a systematic means of constructing temporal queries from nontemporal queries, the motivation being that queries that are easily formulated in SQL on nontemporal relations are very difficult to formulate on temporal relations, as discussed in Section IV.A. With statement modifiers, one thus formulates a temporal query by first formulating the corresponding nontemporal query (i.e., assuming that there are no time-

stamp attributes on the argument relations) and then applies a statement modifier to this query.

For example, to formulate a temporal join the first step is to formulate a nontemporal join. Next, a modifier is prepended to tell the database system to use temporal semantics. The modifier ensures that the argument timestamps overlap and that the resulting timestamp is the intersection of the argument intervals. If the enclosed query is simply a selection, the timestamps do not have to be transformed, and the only task of the modifier is to ensure that the original timestamps are returned as the timestamps of the result. If the enclosed statement is a difference, the modifier ensures that the intervals are appropriately subtracted.

While ATSQL supports both valid time and transaction time, we consider only valid time. Unlike the languages that consider intervals as compact representations of sets of points, ATSQL gives more meaning to intervals. Reflecting this, relations in ATSQL consist of interval timestamped tuples, and value-equivalent tuples with adjacent or overlapping intervals are permitted. Thus, *CheckOut* as given in Fig. 3a is assumed in the following.

The first and second queries can be formulated almost as in SQL. The *seq vt* modifier ensures that the original timestamps are returned as intended. The *seq vt* is short for “sequenced valid time” and indicates that the semantics is consistent with evaluating the nontemporal query on a sequence of nontemporal relations, see Section IV.E. Note that the *seq vt* modifier does not limit the possibility to, e.g., use Allen’s interval predicates, to constrain the timestamps, which is otherwise typical for point-based languages. For example, it is perfectly natural (and easy to understand) to have a temporal join between relations *p* and *q* and to restrict the temporal join to tuples in *p* with a timestamp that spans at least two time points.

```
Q1ATSQL: seq vt
  select * from CheckOut
  where T overlaps [7,9]
Q2ATSQL: seq vt
  select TapeNum
  from CheckOut
  where duration(T) = 2
```

The next two statements are temporal generalizations of nontemporal queries. Thus, they can be formulated by prepending the nontemporal SQL queries by the *seq vt* modifier.

```
Q3ATSQL: seq vt
  select b.TapeNum
```

```
from CheckOut as a,
  CheckOut as b
where a.TapeNum = T1234
  and b.TapeNum <> T1234
Q4ATSQL: seq vt
  select count(*)
  from CheckOut
```

Query *Q5* is to be evaluated independently of the time attribute values of the tuples. This is achieved by using the *nseq vt* modifier (short for “nonsequenced valid time”), which indicates that what follows should be treated as a regular SQL query.

```
Q5ATSQL: nseq vt
  select count(*)
  from CheckOut
```

The design decision to make a query without any modifiers considers only the current state of the argument relations aids in ensuring that legacy queries on nontemporal relations are unaffected if the nontemporal relations are made temporal by the addition of a timestamp attribute. The last query may then be formulated as follows.

```
Q6ATSQL: select * from CheckOut
```

In summary, semantic defaults offer systematic support for writing temporal queries that can be evaluated on all sets of concurrent states of the argument relations in isolation. This language mechanism is independent of the syntactic complexity of the queries that the modifiers are applied to, which is very desirable. While statement modifiers offer attractive means of formulating the example queries, it should be noted that a language based on statement modifiers represents a much more fundamental change to the language than, e.g., a language based on temporal abstract data types.

H. Approach VI: Temporal Expressions

The notion of temporal expression was originally advocated by Gadia and is supported in his TempSQL language, which is based on the data model described in Section III.D. Relations in TempSQL thus consist of tuples with attribute values that are functions from a subset of the time domain to some value domain (specified as a pair of a temporal element and a value). The functions in the same tuple must have the same domain. Figure 7 exemplifies these relations.

In addition, relations are keyed. A set of attributes being a key means here that no two tuples are allowed to exist in the relation that have the same range values

for those attributes. In our example, we are not interested in customers, but in check-outs. Therefore, the key should be SeqNo, not CustID as in Fig. 7. This can be achieved by applying a special restructuring operation to the CheckOut instance on SeqNo, thus obtaining the result in Fig. 10.

The first two queries can be formulated using temporal expressions. If X is an expression that returns a function from time to some value domain then $\llbracket X \rrbracket$ is a temporal expression. This expression returns the domain of X , i.e., returns the time when X is true.

```
Q1TempSQL: select *
      from CheckOut:SeqNo
      where  $\llbracket \text{TapeNum} \rrbracket \cap [7,9] \neq \emptyset$ 
Q2TempSQL: select TapeNum
      from CheckOut:SeqNo
      where duration( $\llbracket \text{TapeNum} \rrbracket$ ) = 2
```

The result of the second query is the relation $\langle [19,22] \text{ T1235} \rangle$.

For the third query, the while clause can be used to specify a temporal element to which the time domain of the result tuples is restricted.

```
Q3TempSQL: select b.TapeNum
      while  $\llbracket \text{a.TapeNum} \rrbracket \cap$ 
             $\llbracket \text{b.TapeNum} \rrbracket$ 
      from CheckOut a,
            CheckOut b
      where a.TapeNum = T1234
            and b. TapeNum <> T1245
```

The fifth query may be formulated as a simple aggregate, where the argument relation is keyed on the sequence number. (The authors have been unable to formulate the fourth query based on the available documentation.)

```
Q5TempSQL: select count(*)
      from CheckOut:SeqNo
```

TempSQL offers built-in support for accessing the current state of a database. Users that wish to query

only the current states of temporal relations are specified as so-called *current users*, upon which the argument relations are always assumed to be the ordinary snapshot relations that contain the current states of the temporal relations.

```
Q6TempSQL: select * from CheckOut
```

In summary, TempSQL timestamps attribute values with temporal elements and permits the use of so-called temporal expressions in this context. This combination simplifies some queries.

V. SUMMARY

This article has introduced the reader to temporal data management with an emphasis on temporal-data semantics and temporal data model and query language concepts.

Based mainly on the three central notions of *fact*, *valid time*, and *transaction time*, we offered a conceptual understanding of a temporal database that records either when facts are true in the modeled reality or when database entities are or were part of the current database state, or records both. With this understanding as the basis and restricting our attention to the valid-time aspect of data, we then surveyed, discussed, and compared different proposals for the temporal relations that serve as the starting points for temporal query languages. This led the foundation for an exploration of important approaches to the querying of temporal data. With the purpose of illustrating diverse query language concepts, we considered the formulation of six queries in concrete language contexts that embody the concepts.

Although temporal relations capture the temporal aspects of data using time points, time intervals, and so-called temporal elements, this article shows that this is not the most important distinction. Rather, some relations are designed to capture collections of time-indexed snapshots, which may be done using all three types of timestamps. In this case, time intervals are merely compact representation of sets of time points. In contrast, other models give more semantics to time intervals and may thus be termed interval based. These models require intervals or temporal elements for timestamping. In addition, they allow uncoalesced relations. We have seen that the interval semantics may to some extent be simulated in point-based models by using an additional “sequence number” attribute.

The different query language concepts explored assume different relation constructs, although they

SeqNo	Cust ID	TapeNum
[3,5]	1	[3, 5] C101 [3, 5] T1234
[5,7]	2	[5, 7] C102 [5, 7] T1234
[9, 12]	3	[9, 12] C102 [9, 12] T1245
[19, 20]	4	[19, 20] C102 [19, 20] T1245
[21, 22]	5	[21, 22] C102 [21, 22] T1245

Figure 10 Relation CheckOut : SeqNo.

may make sense for other relation constructs than the ones used for illustrating them in this article. The abstract data type approach to querying temporal data is simple, but also limited in the support offered. The fold and unfold functions convert between corresponding relations with time point versus time interval timestamps. Point timestamps make the formulation of some queries easier, while interval timestamps are convenient for other queries, as well as for physical representation of relations and user display of query results. Fold and unfold imply a point-based view. The approach that solely uses point timestamps assumes that representation and display are beyond the scope of the query language. This leads to a clean, point-based query language. Next, with syntactic defaults, typical queries may be given very short formulations. However, it is challenging to design a query language with comprehensive support for syntactic defaults that is also easy to understand. The notion of statement modifiers offers what may be termed semantic defaults: modifiers are introduced that control the semantics of any query language statements. The strong point is the support for intervals and the ability to easily construct temporal queries that generalize snapshot queries. Finally, the concept of temporal expressions was explored. These constructs extract the time when a logical expression is true and are quite convenient in formulating a range of queries.

VI. OUTLOOK

Although many important insights and results have resulted from the research in temporal database management, many challenges remain. Some of these are considered here.

The lack of consideration of some of these challenges has reduced the potential of earlier results, including those described in this article. In many cases, core concepts have been established, but it remains to be shown how they may be combined and applied, to simplify and automate the management of time-referenced data in practice.

There is a need for increased *legacy-awareness* in a number of areas within temporal databases. Research is needed that takes into account the reality that most databases are in fact legacy temporal databases and that the applications running on them are in fact legacy temporal database applications. In contrast, most research so far has assumed that applications will be designed using a new temporal data model, implemented using novel temporal query languages, and run on as yet nonexistent temporal DBMSs. In

the short to medium term, this is an unrealistic assumption. Indeed, perhaps in part because of this and despite the obvious need in the marketplace, as yet no prominent commercial temporal relational DBMS exists.

The recent growth in database architectures, including the various types of middleware, prompts a need for increased *architecture-awareness*. Studies are needed that provide the concepts, approaches, and techniques necessary for third-party developers to efficiently and effectively implement temporal database technology while maximally exploiting available architectural infrastructure, as well as the functionality already offered by existing DBMSs. The resulting temporal DBMS architectures will provide a highly relevant alternative to the standard integrated architecture that is generally assumed. As a next step, research is needed on how to exploit existing and novel performance-improving advances, such as temporal algebraic operator implementations and indices, in these architectures. Finally, approaches for transitioning legacy applications will become increasingly sought after as temporal technology moves from research to practice.

Also, there has been little work on adding time to so-called fourth-generation languages that are revolutionizing the user interfaces of commercially available DBMSs. There is a clear need for ways to *visualize temporal data*. Scrolling down a relation with additional timestamp attributes is not an effective means of visualizing the temporal variation in the data.

The results on the *conceptual design* of temporal databases as reported in the literature have potential for finding application in practice, but additional research is needed. When database designers actually understand the core temporal database concepts, perhaps most prominently valid and transaction time, they are able to design better databases using existing models and tools. A central challenge is to provide complete conceptual models, with associated design tools, that cover all aspects of designing a temporal database; empirical evaluation of these by real users is needed to provide essential insights. Reengineering of legacy databases is also a very relevant challenge in this context.

Concerning *performance*, more empirical studies are needed to compare techniques, and to possibly suggest even more efficient techniques. Indexing techniques is an important aspect. While preliminary performance studies have been carried out for all or most of the proposed temporal indexes in isolation, there has been little effort to empirically compare them. More work is also needed on exploiting temporal in-

dexes in algebraic operations other than selection. Finally, there has been little work in refining and validating cost models of temporal operators, or of developing and maintaining database statistics.

A number of research areas that are either separate within temporal databases, overlap with this area, or take temporal databases as their point of departure also pose important challenges. Although not mentioned exhaustively in the coverage below, these areas are slated to offer sample challenges.

In the area of *active databases*, rules responding to database changes and external events are a focus. These may be extended to take into account prior history and temporal trends. For example, an absolute temperature reading in a nuclear power plant may be acceptable if it is part of a decreasing trend, but may signal a problem if it represents an increase. Some initial work has been reported in this area, but as yet there has been little integration of rule constructs and temporal constructs.

The area of *spatiotemporal databases* is becoming increasingly important. Providing built-in support for both space and time makes it convenient to manage objects with extents in physical space and time, enabling new database applications.

For example, many *moving objects* such as people, animals, cars, aircraft, and ships will be equipped with wireless devices (e.g., GPS) that track their positions and make these available for storage in databases. The continued advances in wireless communications and other key technology areas constitute powerful drivers for such applications. While we are already witnessing the appearance of some transportation-related systems, a very substantial growth can be expected in the number, sizes, diffusion, and diversity of moving-object applications.

Next, *multimedia* presentations and *virtual reality* scenarios are in fact special breeds of spatiotemporal databases. And, again, there are powerful enablers of these kinds of spatiotemporal database applications. We are witnessing continued advances in data storage, processor, network, and user interaction technologies. In short, the integration of temporal databases with spatial databases offers exciting new challenges and promises to become an important research area in the future.

The area of *temporal data mining* is relatively new. While extracting static associations from a mass of data is an important goal, more effort needs to be focused on associations that capture time-varying behavior.

The focus among vendors, users, and researchers alike on *data warehousing* has brought new prominence to temporal databases. W. H. Inmon, known as the founder of data warehousing, mentions time vari-

ance as one of four salient characteristics of a data warehouse, and there is general consensus that a data warehouse is likely to contain several years of time-referenced data.

Being temporal, data warehouses are thus prime candidates to benefit from the advances in temporal databases. But cross-fertilization between temporal databases and data warehousing is largely absent. In fact, some of the original impetus for a separate data model and query language for data warehouses arose from a perceived lack of temporal support in the relational model and SQL. Few attempts have been made to exploit the advances in temporal databases in the context of data warehousing, although exceptions do exist. The special architecture of a data warehouse and the emphasis on supporting advanced query functionality, e.g., application-specific time-series analysis, bring novel challenges to temporal database researchers. Reconciling the differences between general relational database schemas and specialized star schemas would help enable users and developers achieve an integrated view of an enterprise.

Adopting a longer term and more abstract perspective, it is likely that new database management technologies and application areas will continue to emerge that provide “temporal” challenges. Due to the ubiquity of time and its importance to most database management applications, and because built-in temporal support generally offers many benefits and is challenging to provide, research in the temporal aspects of new database management technologies may be expected to continue to be needed.

SEE ALSO THE FOLLOWING ARTICLES

Data Modeling: Entity-Relationship Data Model • Data Modeling: Object-Oriented Data Model • Structured Query Language (SQL) • Transaction Processing Systems

BIBLIOGRAPHY

- Clifford, J., and Tuzhilin, A., eds. (1995). *Recent advances in temporal databases: Proceedings of the international workshop on temporal databases*. Workshops in Computing Series. New York: Springer-Verlag.
- Etzion, O., Jajodia, S., and Sripada, S., eds. (1998). *Temporal databases: Research and practice*. LNCS 1399, New York: Springer-Verlag.
- Jensen, C. S., and Dyreson, C. E., eds. (1998). *A Consensus Glossary of Temporal Database Concepts*, 2, pp. 367–405.
- Jensen, C. S., and Snodgrass, R. T. (1996). Semantics of time-varying information. *Information Systems*, 21(4):311–352.

- McKenzie Jr., L. E., and Snodgrass, R. T. (1991). Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys*, 23(4):501–543.
- Özsoyoglu, G., and Snodgrass, R. T. (1995). Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532.
- Roddick, J. F., and Patrick, J. D. (1992). Temporal semantics in information systems—A survey. *Information Systems*, 17(3):249–267.
- Snodgrass, R. T., ed. (1993). *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*. Arlington, TX.
- Snodgrass, R. T. (1995). Temporal object oriented databases: A critical comparison. In *Modern database systems: The object model, interoperability and beyond* (W. Kim, ed.), pp. 386–408, Reading, MA: Addison-Wesley/ACM Press.
- Snodgrass, R. T., ed. (1995). Ahn, I., Ariav, G., Batory, D., Clifford, J., Dyreson, C. E., Elmasri, R., Frandi, F., Jensen, C. S., Käfer, W., Kline, N., Kulkarni, K., Leung, T. Y., Lorentzos, N., Roddick, J. F., Segev, A., Soo, M. D., and Sripada, S. M. *The TSQL2 Temporal Query Language*. Dordrecht: Kluwer Academic Publishers.
- Snodgrass, R. T. (1997). Temporal databases. Part II. *Advanced Database Systems* (C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, eds.), San Mateo, CA: Morgan Kaufmann Publishers.
- Snodgrass, R. T. (2000). *Developing time-oriented database applications in SQL*. San Mateo, CA: Morgan Kaufmann Publishers.
- Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., and Snodgrass, R. T., eds. (1994). *Temporal databases: Theory, design, and implementation*. Redwood City, CA: Benjamin-Cummings.
- Theodoulidis, C. I., and Loucopoulos, P. (1991). The time dimension in conceptual modelling. *Information Systems*, 16(3):273–300.
- Wu, Y., Jajodia, S., and Wang, X. S. Temporal database bibliography update, pp. 338–366.

