

59

Temporal Databases

59.1 Introduction.....	59-1
59.2 History.....	59-2
59.3 Temporal Data Semantics.....	59-5
59.4 Temporal Data Models	59-6
Points • Intervals • Temporal Elements • Attribute Value Timestamping	
59.5 Temporal Query Languages.....	59-11
Motivation • Sample Queries in the Video Rental Example • Auxiliary Constructs • Approach I: Abstract Data Types • Approach II: Folding and Unfolding • Approach III: Point Timestamps • Approach IV: Syntactic Defaults • Approach V: Statement Modifiers • Approach VI: Temporal Expressions	
59.6 Evaluation Algorithms for Temporal Operators ..	59-28
Temporal Join • Temporal Aggregation	
59.7 Summary	59-34
59.8 Outlook.....	59-35
References	59-37

Michael H. Bohlen	
<i>Free University of Bozen/Bolzano</i>	
Johann Gamper	
<i>Free University of Bozen/Bolzano</i>	
Christian S. Jensen	
<i>Aalborg University</i>	

Database applications typically manage time-referenced, or temporal, data. The research area of temporal databases is concerned with the semantics of temporal data and the provisioning of concepts and techniques for the modeling, representation, update, and querying of temporal data. This chapter introduces the reader to temporal data models, query languages, and query processing techniques. In doing so, 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.

59.1 Introduction

Many or even most applications of database technology rely on the management of time-referenced data, termed temporal data. Examples include financial applications such as portfolio management, accounting, insurance, and banking; record-keeping applications such as personnel, medical-record, and inventory management; scheduling applications such as airline, train, and hotel reservations and project and business-process management; and scientific applications such as weather and climate monitoring.

Studies have shown that built-in support for the management of temporal data has the potential for simplifying application code and, in particular, the development of application code [Sno99]. In spite of this, the support for temporal data management found in today's commercially available software tools and data management systems continues to

be relatively limited and to leave much to be desired. In part as a result, temporal database management remains an active field of research.

Borrowing from and extending previous work by the authors [BJ03, BGJ06a], this chapter provides an introduction to central concepts and issues in temporal data models, query languages, and query processing algorithms. More specifically, the chapter examines in turn the semantics of temporal data, temporal data models, and temporal query languages, which are central areas of temporal database research. Each area is first motivated, and then sample contributions are surveyed, the purpose being to give the reader a feel for the challenges and existing solutions in each particular area. The chapter proceeds to consider temporal query processing techniques, and it concludes with a summary and a look into the possible futures of temporal databases.

Rather than attempting to offer a coverage of all areas and being complete within each area, the chapter focuses on selected, central areas; and within each such area, it aims to cover key contributions and to illustrate the breadth of the contributions.

The chapter does not cover the conceptual and logical modeling of temporal data [GJ99, JS00], nor does it explore the relations to active areas of research such as data mining, data streams, and spatio-temporal databases.

The chapter also does not cover the issues related to the representation of the time domain such as whether time is assumed to be continuous or discrete. It 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 notion of the current time *now*. The chapter focuses on the recording of phenomena that have a duration in time, and it largely avoids issues specific to the recording of instantaneous events and time series.

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

59.2 History

We proceed to provide a short historical account of the research in temporal databases. We point to a series of bibliographies that list work published in the area; we list and summarize collections, textbooks, and surveys on temporal databases; and we list events and efforts that included the temporal database community at large.

While our focus is on the results obtained by the database community, researchers from other communities have also investigated the role of time in information processing. In *database systems*, focus is typically on general and scalable solutions to deal with information that start and stop to be true. In *artificial intelligence*, focus is often on models that not only describe a snapshot of the real world, but also include the history and evolution of the world. In *logic*, a large number of temporal logics have been investigated since the classical logic was considered inadequate for capturing the meaning of time-varying information. In *linguistics*, the semantics of time-varying statements has been a key interest of research.

Temporal databases has been an active area of research since the 1980's, and the temporal database community has produced more than 2000 papers. Many of these are listed in a series of seven cumulative bibliographies (the newest one [WJW98] provides pointers to its predecessors).

1983 A. Bolour, T. Anderson, L. Dekeyser, and H. Wong, *The Role of Time in Information Processing: A Survey*, ACM SIGMOD Record, 12(3): 27–50.

- 1986** E. McKenzie, *Bibliography: Temporal Databases*, ACM SIGMOD Record, 15(4): 40–52.
- 1988** R. Stam and R. T. Snodgrass, *A Bibliography on Temporal Databases*, IEEE Data Engineering Bulletin, 7(4): 53–61.
- 1991** M. Soo, *Bibliography on Temporal Databases*, ACM SIGMOD Record, 20(1): 14–23.
- 1993** N. Kline, *An Update of the Temporal Database Bibliography*, ACM SIGMOD Record, 22(4): 66–80.
- 1996** V. Tsotras and A. Kumar, *Temporal Database Bibliography Update*, ACM SIGMOD Record 25(1): 41–51.
- 1998** Y. Wu, S. Jajodia, and S. Wang: *Temporal Database Bibliography Update*, Temporal Databases: Research and Practice, LNCS 1399, pp. 338–366.

During the last two decades, there were a number of events and activities that included the temporal database community at large. These events stand out, as they involved people with different backgrounds and interests, and they contributed to shaping the area.

- 1987** *IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems, Sophia-Antipolis, France*. This conference covered topics such as requirements for temporal data models and information systems, temporal query languages, versioning, implementation techniques, as well as temporal logic, constraints, and relations to natural language. A 249-page proceedings documents the results presented [RBL87].
- 1993** *ARPA/NSF International Workshop on an Infrastructure for Temporal Databases, Arlington, USA*. This workshop gathered researchers in temporal databases with the very ambitious goal of consolidating the different approaches to temporal data models and query languages. A substantial proceedings containing more than 40 papers resulted [Sno93], and a special section on temporal database infrastructure appeared in the March 1995 issue of the ACM SIGMOD Record. This workshop also gave birth to the TSQL2 query language [PSE⁺94, Sno95].
- 1995** *International Workshop on Temporal Databases, Zurich, Switzerland*. This workshop was the first to be co-located with the VLDB conference and supported by the VLDB Foundation. The success of the workshop triggered a discussion as to whether to establish a series of workshops on temporal databases. The majority decided against this. A prevailing argument was that it is important that temporal database results appear in the top database outlets. A 360-page proceedings was published by Springer [CT95].
- 1996** *SQL/Temporal: Part 7 of SQL3*. R. T. Snodgrass initiated and led an effort to transfer results of temporal database research into SQL3. The first step was a proposal of a new part to SQL3, termed SQL/Temporal. This new part was accepted as Part 7 of the SQL3 specification and includes the PERIOD data type. It was discussed subsequently how to add valid- and transaction-time to SQL/Temporal. Two change proposals, ANSI-96-501 and ANSI-96-502, were accepted by ANSI and forwarded to ISO. Due to disagreements within the ISO committee on how to support time in SQL3, the initiative came to a hold near the end of 2001.
- 1997** *Dagstuhl seminar on Temporal Databases, Dagstuhl, Germany*. O. Etzion, S. Jajodia, and S. Sripada organized a Dagstuhl seminar to discuss future directions of temporal databases, with respect to both research issues and the means to

incorporate temporal databases into mainstream application development. A 439-page proceedings was published by Springer [EJS98].

The conference series *International Symposium on Temporal Representation and Reasoning* (<http://time.dico.unimi.it>) has been held annually since 1994. While this series emanated from outside the temporal database community, it has seen increasing participation from the temporal database community in recent years, in terms of organizers, program committee members, and participants.

The TimeCenter consortium (<http://www.cs.aau.dk/TimeCenter>) was created in 1997 with the objective of increasing the impact on database management practice of temporal database research. TimeCenter makes available a number of resources, including almost 100 papers.

In addition to the proceedings and resources mentioned already, the community has produced several collections, textbooks, and surveys on temporal databases. An edited volume captures state of the art circa 1993 [TCG⁺93], and a consensus effort has resulted in a comprehensive glossary of terminology [JD98] that is available online at <http://www.cs.aau.dk/~csj/Glossary>. The topic of temporal databases is now covered in many textbooks, and the first book dedicated entirely to temporal databases has appeared [Sno99, Sno97]. The listing that follows covers only resources that appeared after 1990.

- 1993** *Temporal Databases: Theory, Design, and Implementation*, A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass (editors) [TCG⁺93]. This collection of contributions to temporal databases describes primarily a number of data models and query languages produced during the first ten years of temporal database research.
- 1995** *The TSQL2 Temporal Query Language*, R. Snodgrass (editor), Kluwer Academic Publishers [Sno95]. TSQL2 is an effort to design a consensus model and query language. TSQL2 includes many of the concepts that have been proposed by early temporal data models and query languages.
Temporal and Real-Time Databases: a Survey, G. Özsoyoglu and R. T. Snodgrass, *IEEE TKDE* 7(4) [ÖS95]. This article surveys temporal database and the related area of real-time databases. It is part of a special issue.
Temporal Database System Implementations, M. Böhlen, *ACM SIGMOD Record*, 24(4) [Böh95]. This article provides pointers to prototypes related to temporal database systems and discusses their respective features and properties.
- 1999** *Temporal Data Management*, C. S. Jensen and R. T. Snodgrass, *IEEE TKDE*, 11(1) [JS99]. This article is part of a volume that surveys research areas. The article offers a brief introduction to temporal databases, covering data semantics, conceptual and logical design, query languages, and implementation techniques.
Developing Time-Oriented Database Applications in SQL, R. Snodgrass, Morgan Kaufmann Publishers [Sno99]. This book explains how to record temporal data using SQL data types, define appropriate integrity constraints, update temporal tables, and query temporal tables with interactive and embedded SQL. The book and its accompanying CD-ROM contains substantial SQL code with detailed explanations.
- 2003** *Temporal Data Model and Query Language Concepts*, M. Böhlen and C. S. Jensen, *Encyclopedia of Information Systems, Volume 4*, Academic Press, Elsevier Science [BJ03]. This article offers a brief introduction to central concepts in temporal data models and query languages. The present chapter substantially extends this article.

59.3 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 with data, 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, time values are associated with database entities. A time value associated with an entity is termed a *timestamp* of that entity. 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 entity, say “*e*,” is the valid time of the related fact, “*e 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 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. These properties provide 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 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 SQL standards. Third, a variety of different structures have been imposed on time. Most often, time is assumed to be totally ordered (sometimes termed linear time), but various partial orders (sometimes termed branching time) 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 closest spatial equivalent, *here*, simply fails to possess similar properties. The uniqueness of *now* may be viewed as 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 (non-temporal) data models and query languages. Accommodating the time-varying nature of the mini-world 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 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 59.4 considers temporal data models, Section 59.5 then covers query languages based on these models, and Section 59.6 provides an overview of algorithms for computing temporal joins and temporal aggregation.

59.4 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, and we generally assume a discrete, totally ordered, and bounded time domain, as does probably most data model research.

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 2007. 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 graphical representation of this **CheckOut** instance is shown in Figure 59.1, where timestamps are represented as horizontal lines.

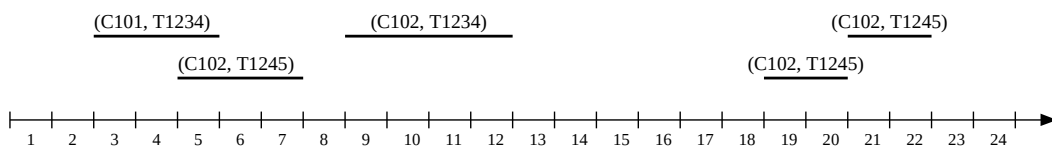


FIGURE 59.1 Graphical Representation of the **CheckOut** Relation

59.4.1 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 Figure 59.2.

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 59.2 The Point Model of the **CheckOut** Instance

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, $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$, 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 Figure 59.2, 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.

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.

59.4.2 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 59.3 illustrates the approach.

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 59.3 An Interval Model of the **CheckOut** Instance

When employing intervals rather than time points as timestamps, two timestamps satisfy precisely one of the following thirteen relationships, first enumerated by James Allen [All83]: *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.

The notion of *snapshot equivalence*, which reflects a time-point-based view of the association of time with data, establishes a correspondence between the point-based and interval-based models. Informally, two relations are snapshot equivalent if, at every time point, their snapshots are equal, where a snapshot at a time instant t is defined as the set of tuples (reduced to the non-timestamp attributes) that hold at t . Consider the relations in Figure 59.4. The relations are different, but snapshot equivalent. Specifically, the relation in Figure 59.4(b) is a coalesced version of the relation in Figure 59.4(a). (In *coalescing*, tuples with the same non-timestamp values, termed value-equivalent tuples, and adjacent or overlapping timestamps are merged.)

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 Figure 59.5.

These models are termed *weakly interval-based*. They 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

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

(a) Instance 1

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

(b) Instance 2

FIGURE 59.4 Different But Snapshot Equivalent **CheckOut** Instances

CustID	TapeNum
C101	T1234

(a) Snapshot at Time 4

CustID	TapeNum
C101	T1234
C102	T1245

(b) Snapshot at Time 5

CustID	TapeNum
--------	---------

(c) Snapshot at Time 8

FIGURE 59.5 Instances of the **CheckOutSnap** Relation, Representing Different Snapshots of the **CheckOut** Relation

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 Figure 59.4. 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 Figure 59.3 that uses the additional **SeqNo** attribute is thus the appropriate representation of the rental scenario.

In an *inherently interval-based* model, an interval is not merely a representational device—it carries 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 (cf. Figure 59.4(a)) or for one longer interval (cf. Figure 59.4(b)). Inherently interval-based models do not enforce coalescing. They 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 Figure 59.4(a) is the appropriate representation of the rental scenario.

59.4.3 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 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. Note that many sets of intervals generally exist that are equivalent to the same temporal element. It should also be noted that the definition is meaningful for both a discrete time domain and a continuous time domain.

With temporal elements, it is natural to use point-based semantics. To see why, observe that the set $\{[19, 20], [21, 22]\}$ of two intervals and the set $\{[19, 22]\}$ of one interval represent the same temporal element because $[19, 20] \cup [21, 22] = [19, 22]$. So two tuples are needed to capture the two consecutive two-day rentals in our example, which is not particularly intuitive. However, (inherently) interval-based semantics can be supported with temporal elements.

Figure 59.6(a) uses temporal elements to capture the rental scenario assuming the interval-based semantics. Under the point-based semantics (or weakly interval-based semantics), an interval is identical to a set of points. This implies that the instance in Figure 59.6(a) is equivalent to the one shown in Figure 59.6(b). With the point-based semantics it is possible and common to enforce a canonical representation that does not permit value-equivalent tuples. The instance in Figure 59.6(b) is then the only valid representation of the **CheckOut** relation.

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

(a) Interval View

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

(b) Point View

FIGURE 59.6 Temporal Element Representation of the **CheckOut** Instance

We consider next the bitemporal conceptual data model (BCDM) [JSS93], which has been used for TSQL2, to be discussed in Section 59.5.7. This model adopts a point-based view. So the **SeqNo** attribute is introduced in Figure 59.7, which shows the BCDM relation instance that captures the rental scenario.

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 59.7 Using the BCDM to Model the **CheckOut** Instance

The BCDM timestamps facts with values that are sets of time points. (We use 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 relations 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.

59.4.4 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. (The less accurate terminology

“attribute timestamping” is also being used for this.) 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 non-temporal database. In other words, a temporal database is a non-temporal database that is time parameterized.

The relation in Figure 59.8 is thought of as recording information about some types of objects. Specifically, 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 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.

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 59.8 Parametric Representation of the **CheckOut** Instance by “grouping” on **CustID**

It should also be noted that different groupings of the information into tuples are possible for this attribute value timestamping model. For example, Figure 59.9 illustrates a relation instance that is snapshot equivalent to the one in Figure 59.8. 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 59.5.9 elaborates on this.)

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 59.9 Parametric Representation of the **CheckOut** Instance by “grouping” on **TapeNum**

59.5 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 with eight queries, which are used in the sequel for a comparative discussion of six different approaches to the creation of a temporal query language.

59.5.1 Motivation

Given the prevalence of existing applications that 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 Figure 59.3, which records check-out intervals for video rentals, and **CheckOutSnap** in Figure 59.5(b), 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 non-trivial. The reader may verify that the result in Figure 59.10 is correct. (The tuples with a count value of 0 might or might not be included in the result; we do not consider this as a major difference in the analysis of the query languages, and we use $-\infty$ and ∞ as the smallest and largest time value in the time domain, respectively). Although possible, expressing this query in SQL is quite difficult. The authors are aware of one step-wise and systematic solution that consists of six steps and takes up 35 lines of complex SQL code. A similar solution has been proposed recently in [Zim06].

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	$[23, \infty]$

FIGURE 59.10 A Time-Varying Count

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. In one formulation, it takes a complicated, twelve-line SQL assertion to express this constraint [Sno97, Ch. 5.3].

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 would be hard pressed to express the examples above in SQL. Given also the ubiquity of temporal information, this indicates that temporal support beyond what SQL offers today is highly desirable.

59.5.2 Sample Queries in the 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.

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

CustID	TapeNum	T
C102	T1245	[5,7]
C102	T1234	[9,12]

Query Q1 asks for all available information about check-outs that overlap the interval [7, 9]. Returning the non-temporal information of a check-out is straightforward. Returning the temporal information is more subtle, as we will see in Section 59.5.6.

Q2 *All 2-day check-outs.*

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

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 4-day check-out from two consecutive 2-day check-outs.

Q3 *Which tapes were checked out together with tape T1234?*

TapeNum	T
T1245	[5,5]

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

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

See Figure 59.10.

Query Q4 is the first of four aggregation queries. Like Query Q3, it is an example of an ordinary query that must be applied to each state of a temporal database. The difference with respect to Q3 is the non-temporal query, which is now an aggregation. Query Q4 is an example of an *instant* aggregation query [KS95, MVI03]. The result at a specific time point is computed over all tuples that are valid at that time point. In the result relation, tuples over consecutive time points that have the same aggregate value and the same lineage information [CWW00] are coalesced into tuples over maximal time intervals, termed *constant intervals*. Notice that in Fig. 59.10 the tuples (1, [19, 20]) and (1, [21, 22]) are not coalesced, since they have different lineage information, i.e., they derive from different data tuples.

Q5 *What is the weekly number of tapes that have been checked out?*

Cnt	T
2	[1,7]
1	[8,14]
2	[15,21]
1	[22,28]

Query Q5 is another example of a temporal aggregation query, also termed *span aggregation*, where the partitioning of the time line (here, into weeks) is predefined and independent of the actual data. It has the same non-temporal part as Q4, but here the query explicitly specifies *time spans* over which to evaluate the non-temporal aggregation, i.e., the weeks for which there are tuples in the **CheckOut** relation. The result for a specific week is computed over all tuples that overlap that week.

Q6 *At each time, what is the number of tapes that have been checked out in the last seven days?*

Cnt	T
0	[1,2]
1	[3,4]
2	[5,8]
3	[9,11]
2	[12,13]
1	[14,18]
1	[19,20]
2	[21,26]
1	[27,28]

Query *Q6* is yet another temporal aggregation query, termed *moving-window aggregation* (or *cumulative aggregation*). It slides along the time line, computing at each time point an aggregate that takes into consideration all tuples that were valid at some point during the last seven days. In general, the value of a moving-window aggregate at time point t is computed over all tuples with a valid time that overlaps with the interval $[t - w, t]$, where w is a positive window offset. In the result relation, tuples over consecutive time points that have the same aggregate value(s) and lineage information are coalesced.

Q7 *How many check-outs were made in total?*

Cnt
5

Query *Q7* is the last aggregation query we consider. Here, the aggregation is to be applied independently of any temporal information. Instead of a time-varying count of the check-outs, we determine the total number of check-outs.

Q8 *List all (current) check-outs.*

CustID	TapeNum
C101	T1234
C102	T1245

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

59.5.3 Auxiliary Constructs

We introduce a couple of concepts that are fundamental when expressing temporal aggregation queries and that are either not supported or barely supported in current temporal query languages. These concepts concern the computation of the timestamps for the result tuples that depend on the data in the argument relation and possibly also on the query. We present these concepts next and use them in the subsequent coverage of query languages.

An abstract unary **Chron** relation has been proposed that has a single temporal attribute **T** that stores all possible chronons (time points) of the temporal universe [Tom97]. Such a **Chron** relation is helpful when expressing certain queries. For the use of this relation to be practical, implementation level solutions have to be developed that do not require the materialization of the **Chron** relation.

Consider Query *Q5*, which explicitly involves the periods during which a result tuple is expected, i.e., every week where a tuple is valid. Using the **Chron** relation, we can construct these weeks as follows:

```
create view Week(T) as
  select distinct week(a.T) as T
  from   Chron as a, Checkout as b
  where  overlaps(week(a.T),b.T)
```

The function **week**(*T*) takes as argument a chronon and returns the week to which this chronon belongs. For example, **week**(2) returns the first week represented as an interval: [1, 7].

Again, note that if a specific query uses non-temporal grouping attributes, they have to be considered in the above view definition, which precludes a general solution for all queries with specified timestamps.

Another non-standard feature that facilitates the formulation of temporal queries is the availability of generative constructs: general functions that return sets of values that are then further processed. Examples include functions that generate all time points included in an interval or all weeks covered by an interval.

Being more precise, we assume a user-defined function *f* that takes as input a timestamp *T* and returns a set of intervals, i.e.,

$$f(T) = \{I_1, \dots, I_m\}$$

This function can then be used in the query language and has the following semantics:

$$SQL(f(T)) \equiv SQL(I_1) \cup \dots \cup SQL(I_m)$$

That is, we evaluate the SQL query for each of the intervals returned by *f* and form the union of the result tuples.

59.5.4 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 Figure 59.4(a).

Q1. Formulations of predicates on time-interval data types have been influenced by Allen's 13 interval relationships [All83]: 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.

```
Q1SQL:    select *
           from   CheckOut
           where  T overlaps [7,9]
```

The result of this query depends on how the rental information is represented—cf. Figure 59.4, which illustrates that the same information (from a point-based view) may be represented in more than one way.

Q2. A selection that limits the duration of an interval follows the same approach:

*We use the *overlaps* predicate with the semantics defined in the TSQL2 language, i.e., the predicate holds if the two timestamps being related share one or more time points. Notice that this definition is different from the definition provided by Allen.

59-16

```
Q2SQL:    select TapeNum, T
           from   CheckOut
           where  duration(T) = 2
```

Q3. The third query can be expressed using selection, projection, and join. Its formulation is simplified if predicates and functions on intervals are available:

```
Q3SQL:    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
```

Q4. As mentioned in Section 59.5.1, 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*). The hard part is that of expressing the computation of the constant intervals for each aggregate value in the result, while expressing the computation of the aggregate function is quite straightforward and needs just a Boolean function to test the overlapping of timestamps.

Q5. Query *Q5* explicitly specifies the periods for which a result tuple is expected, i.e., for every week. With the **Week** view introduced above we can express the query as follows:

```
Q5SQL:    select  count(*) as Cnt, a.T
           from    Week as a, CheckOut as b
           where   a.T overlaps b.T
           group by a.T
```

The only difference to Query *Q4* is in the computation of the timestamps of the result tuples, which requires a **Chron** relation and a **week** function as introduced above.

Q6. Query *Q6* is similar to *Q4* in that the aggregate function is computed for each time point, and consecutive time points with the same result value and identical lineage are coalesced. The timestamps of the result tuples can be computed from the argument tuples similarly to how this was done for the constant intervals. However, the length of the moving window has to be considered, and the timestamps of the result tuples extend beyond the timestamps of the argument tuples, which makes the computation of the timestamps even more complicated. Assuming a view **MWConstantInt** for the timestamps of the result tuples (the hard part), where *Ts* and *Te* denote the start and end point of the timestamps, respectively, we can formulate Query *Q6* as follows:

```
Q6SQL:    select  count(*) as Cnt, a.T
           from    MWConstantInt as a, CheckOut as b
           where   [a.Ts-6,a.Te] overlaps b.T
           group by a.T
```

For a result tuple, we aggregate over all argument tuples that overlap the time interval that extends the timestamp of the result tuple backwards by six days.

Q7. In contrast to Queries *Q4*–*Q6*, counting the check-outs independently of the temporal references is easy:

```
Q7SQL:    select count(*) as Cnt
           from   CheckOut
```

Q8. Retrieving the current check-outs is done as follows:

```
Q8SQL:      select CustID, TapeNum
              from   CheckOut
              where  T overlaps [now,now]
```

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–Q6*.

Let us consider this point in a bit more detail, as it exposes 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 being generalized: a join (*Q3*) versus an aggregation (*Q4*).

In spite of the close relationship between these two queries, 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 59.5.1. Indeed, the many years of temporal database research have not led to an abstract data type that allows one to conveniently express temporal queries.

59.5.5 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 timestamp 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 [LJ88, DDL02], the earliest and most radical approach is to introduce the two functions **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 Figure 59.4 yields the relation in Figure 59.2 (modulo the sequence number attribute), which may be folded to yield the relation in Figure 59.4(b). 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 together with the **CheckOut** relation in Figure 59.4(a). IXSQL inherits and

extends the semantics of SQL. Thus, each SQL query statement is also an IXSQL query statement.

Q1, Q2, Q3. 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 59.5.4.)

```
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
```

Q4. 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(*) as Cnt, T
           from    ( select  *
                     from    CheckOut
                     reformat as unfold T) as a
           group by T
           reformat as fold T
```

The inner query unfolds the argument relation yielding the point-based representation shown in Figure 59.2. Then the aggregation is computed on this relation, and the **fold** function is used for transforming it back into an interval-stamped relation.

Note that the result obtained is different from the intended result in Figure 59.10. This is because the normalization step does not carry over any lineage information. The combination of unfolding and folding yields maximal intervals of snapshot equivalent tuples independently of the argument tuples that produced the result. In particular, the third and second last tuples are merged into the single tuple (1, [19, 22]).

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

Q5. To express Query *Q5*, a slightly different pattern is applied that uses only the **unfold** operator. Moreover, the function **week** that generates weeks is needed together with a join.

```

Q5IXSQL:  select  count(*) as Cnt, a.W as T
           from    ( select  distinct week(T) as W
                     from    CheckOut
                     reformat as unfold T ) as a,
                     CheckOut as b
           where    a.W cp b.T
           group by a.W

```

Again, the `unfold` function first transforms the interval-timestamped relation into a point-timestamped relation, from which the different weeks are extracted. Then the join operator selects for each week the relevant tuples (with interval timestamps) over which the aggregation is then computed. The second relation in the join needs not be unfolded.

Q6. The moving-window aggregation query combines the patterns of the previous two queries: we unfold the `CheckOut` relation so that we can work with time points. We then use a join to match this with tuples within the specified window. Finally, we transform the result back to the interval representation.

```

Q6IXSQL:  select  count(*) as Cnt, a.T
           from    Chron as a,
           ( select  *
             from    CheckOut
             reformat as unfold T ) as b
           where    a.T-6 <= b.T and b.T <= a.T
           group by a.T
           reformat as fold a.T

```

Notice that the transformation into interval-timestamped result tuples by the `fold` operation yields a result relation where the tuples $(1, [14, 18])$ and $(1, [19, 20])$ are coalesced into $(1, [14, 20])$, which is not the intended result.

Q7, Q8. 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.

```

Q7IXSQL:  select count(*) as Cnt
           from    CheckOut

Q8IXSQL:  select CustID, TapeNum
           from    CheckOut
           where    T cp [now,now]

```

Summary. A language enriched with folding and unfolding may offer a systematic approach to formulating temporal queries, in particular temporal queries that generalize non-temporal queries.

For the time domains available in current systems, unfolded relations are so large that storing them is impractical. The efficient evaluation of queries formulated using fold and unfold must thus avoid unfolding. How to accomplish this in a general setting is an interesting challenge.

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 Figure 59.4(a) yields the instance in Figure 59.4(b).

59.5.6 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 [Tom97] takes this approach to generalizing queries on non-temporal 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 Figure 59.2 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. *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 * from CheckOut where T = 7 or T = 8 or T = 9`. However, 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 distinct a.*
           from   CheckOut a, CheckOut b
           where  a.SeqNo = b.SeqNo and (b.T = 7 or b.T = 8 or b.T = 9)
```

It should also be noted that the `distinct` keyword in the `select` clause is required in order to remove the duplicates that will be introduced if a tuple shares more than one time point with the interval [7, 9].

Q2. In SQL/TP, duration queries are formulated as aggregation 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 that distinguishes the individual check-outs. If this additional attribute is omitted from the final projection, the 2-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.

Q3, Q4. 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.

```
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(*) as Cnt, T
           from    CheckOut
           group by T
```

Query *Q3* uses a self-join to determine the time points when tape T1234 was checked out together with another tape. Unlike in Query *Q1*, there are no problems with duplicates. This is so because the timestamps of the tuples retrieved are reduced to the overlapping interval, which is expressed by the equality on the timestamps of the two input relations. A slightly modified query that is to retrieve the original tuples that overlap with the T1234 rental will introduce duplicates as in *Q1*.

In Query *Q4*, the grouping takes care of isolating the database states from each other. The query is restricted to bounded time domains, to avoid infinite relations.

Q5. Span aggregates are more difficult to express. In Query *Q5* we have to group the tuples into weeks, and each time point of a week must produce the same aggregate value.

```
Q5SQL/TP: select  count(distinct b.SeqNo) as Cnt, a.T
              from    Chron as a, CheckOut as b
              where    a.T-1 div 7 = b.T-1 div 7
              group by a.T
```

The condition in the **where** clause groups the argument tuples by week. The aggregate function is computed over these groups and assigned to each time point in the week.

Notice that the join with the **Chron** relation is needed to guarantee a result tuple for each day in a week including those days where no argument tuples are present.

Q6. SQL/TP does not provide any natural support for the formulation of moving-window queries, i.e., it offers no mechanism for moving a window of fixed size over the time line while producing a result at each time point.

```
Q6SQL/TP: select  count(distinct SeqNo) as Cnt, a.T
              from    Chron as a, CheckOut as b
              where    a.T-6 <= b.T and b.T <= a.T
              group by a.T
```

Again, a join with the **Chron** relation is needed to generate result tuples at time points where no data tuples are present. Specifically, in moving-window queries an input tuple produces a result tuple at time points that are beyond the input tuple's timestamp.

Q7. 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 values. 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 additional attribute in the argument relation to support time-invariant aggregation.

```
Q7SQL/TP: select count(distinct SeqNo) as Cnt
              from    CheckOut
```

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 yield the intended result. However, if an attribute is available that identifies the real-world facts, such as **SeqNo**, counting the facts is possible.

Q8. The current state of the **CheckOut** relation is retrieved as follows:

```
Q8SQL/TP: select CustID, TapeNum
              from    CheckOut
              where    T = now
```

Summary. 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?* illustrates 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. However, if we include the sequence number attribute in the `select` clauses, the temporal difference is effectively disabled, and the returned result is simply 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 only 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.

59.5.7 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. Frequently used defaults concern access to the current state of a temporal database and temporal generalizations of non-temporal queries such as joins. The most comprehensive approach based on syntactic defaults is the TSQL2 language, which we use for exemplification. As TSQL2 adopts a point-based view, we assume the `CheckOut` instance in Figure 59.3, where the intervals are a shorthand representation of time points.

Q1. In TSQL2, a `valid` clause that by default is present implicitly 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'
```

Q2. 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
           from   CheckOut
           where  cast(valid(T) as interval) = 2

```

The returned result must include the `SeqNo` attribute to properly show the 2-day check-outs (cf. Section 59.5.6).

Q3. The default behavior of the implicit `valid` clause was designed with snapshot reducibility in mind. This shows nicely when 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 `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.

Q4. To formulate an instantaneous aggregation, we use a `valid` function that returns the implicit timestamp attribute value of an argument tuple. In the query, the `using instant` is in fact the default and could be omitted (we added it for clarity). Thus, `valid(CheckOut)` denotes the original timestamps of check-outs, and we want to group according to time points, not the original timestamps.

```

Q4TSQL2:  select  count(*) as Cnt
           from    CheckOut
           group by valid(CheckOut) using instant

```

Q5. The grouping into intervals is supported through the `using` clause. In this case, we specify a grouping of 7 chronons (i.e., one week). In passing, we mention that we are uncertain whether this indeed yields [1, 7], [8,14], etc. or whether shifted weeks might result.

```

Q5TSQL2:  select  count(*) as Cnt
           from    CheckOut
           group by valid(CheckOut) using 7 days

```

Q6. TSQL2 provides native support for moving-window aggregates. Specifically, the `group by` clause allows the specification of a leading and trailing time interval for a moving window. Hence, *Q6* can be expressed as follows:

```

Q6TSQL2:  select  count(*) as Cnt
           from    CheckOut
           group by valid(CheckOut) leading 6 days

```

Q7, Q8. The default behavior of TSQL2 is to return temporal relations. The `snapshot` keyword is used to retrieve non-temporal relations, exemplified in the last two queries.

```

Q7TSQL2:  select snapshot count(*) as Cnt
           from    CheckOut

Q8TSQL2:  select snapshot *
           from    CheckOut
           where  valid(CheckOut) overlaps [now,now]

```

Summary. 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 is challenging, perhaps even impossible, to be comprehensive and systematic in the specification of such defaults, and to ensure that the defaults do not interact in unattractive 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.

59.5.8 Approach V: Statement Modifiers

ATSQL introduces temporal statement modifiers to add temporal support to SQL [BJS00]. In contrast to syntactic defaults, statement modifiers are semantic in that they apply in the same manner to any statement they modify.

The basic idea in statement modifiers is to offer a systematic means of constructing temporal queries from non-temporal queries, the motivation being that queries that are easily formulated in SQL on non-temporal relations are very difficult to formulate on temporal relations, as discussed in Section 59.5.1. With statement modifiers, one thus formulates a temporal query by first formulating the corresponding non-temporal query (i.e., assuming that there are no timestamp 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 non-temporal 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.

Unlike the languages that merely 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 Figure 59.4(a) is assumed in the following.

Q1, Q2. 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 non-temporal query on a sequence of non-temporal relations (cf. Section 59.5.6).

```
Q1ATSQL:  seq vt
           select *
           from  CheckOut
           where T overlaps [7,9]
```

```
Q2ATSQL:  seq vt
           select TapeNum
           from  CheckOut
           where duration(T) = 2
```

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.

Q3,Q4. The next two statements are temporal generalizations of non-temporal queries. Thus, they can be formulated by prepending the non-temporal 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(*) as Cnt
           from   Checkout
```

Q5. By default, the `seq vt` clause operates at the lowest granularity and computes constant intervals. This behavior can be extended by allowing the user to specify different granularities or, in the general case, fixed intervals. Below we show an extended modifier that specifies the intervals for which a result tuple is to be produced.

```
Q5ATSQL:  seq vt for weeks(T)
           select count(*) as Cnt
           from   Checkout
```

In this query, the `weeks` function is a generative function that returns all weeks that a given interval timestamp spans, e.g., `weeks([1, 10]) = {[1, 7], [8, 14]}`.

Q6. Moving-window aggregation is an extension of regular instantaneous aggregation. Various syntactic constructs have been proposed for moving-window aggregation. We use the syntax of the Oracle OLAP extensions to enable such aggregates in the context of modifiers.

```
Q6ATSQL:  seq vt with range between interval '6' days preceding and current
           select count(*) as Cnt
           from   Checkout
```

Q7. Query `Q7` is to be evaluated independently of the time attribute values of the tuples. This is achieved by using the `nseq vt` modifier (short for “non-sequenced valid time”), which indicates that what follows should be treated as a regular SQL query.

```
Q7ATSQL:  nseq vt
           select count(*) as Cnt
           from   Checkout
```

Q8. In ATSQL, a query without any modifiers considers only the current states of the argument relations. This is done to aid in ensuring that legacy queries on non-temporal relations are unaffected if the non-temporal relations are made temporal by the addition of a timestamp attribute. The last query may then be formulated as follows.

```

Q8ATSQL:  select *
           from  CheckOut

```

Summary. Statement modifiers offer systematic support for formulating 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 where it is sufficient to extend the base language constructs with new data types, predicates, and functions.

59.5.9 Approach VI: Temporal Expressions

The notion of temporal expression was originally advocated by Gadia and is supported in the TempSQL language [TCG⁺93, p.28ff], which is based on the data model described in Section 59.4.4. 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 59.8 exemplifies such 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 Figure 59.8. This can be achieved by applying a special restructuring operation to the **CheckOut** instance on **SeqNo**, thus obtaining the result in Figure 59.11, which is used in the following discussion.

SeqNo		CustID		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 59.11 Relation **CheckOut:SeqNo**

Q1, Q2. 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., it 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 $\{\langle [19, 22] \text{ T1245} \rangle\}$.

Q3. 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  [ a.TapeNum ]  $\cap$  [ b.TapeNum ]
           from    CheckOut:SeqNo as a, CheckOut:SeqNo as b
           where   a.TapeNum = T1234 and b.TapeNum  $\neq$  T1245
```

Q4, Q7. The fourth query may be formulated as an aggregate, where the argument relation is keyed on the sequence number. For such aggregate queries TempSQL automatically performs an instant aggregation [TCG⁺93, p.42].

```
Q4TempSQL: select count(*) as Cnt
           from      CheckOut:SeqNo
```

Since the above query returns the result of the instant temporal aggregation a different query must be used to determine the time-invariant count in *Q7*. One possibility would be to formulate a query that first drops or equalizes all timestamps and then performs the above aggregation.

Q5, Q6. For query *Q5* we assume an appropriate representation of the view **Week** with a single attribute **W** that stores the four weeks covered by the **CheckOut** relation, i.e., $\{([1, 7] \ 1), ([8, 14] \ 2), ([15, 21] \ 3), ([22, 28] \ 4)\}$. Similarly, we assume a view **MWConstantInt** with a single attribute **CInt** that represents the constant intervals of the moving-window aggregation (the hard part). We need also a function **expand** that expands a time interval into the past and the future, thus allowing to deal with the window offset in moving-window queries. With these views and functions in place we can formulate queries *Q5* and *Q6* as follows.

```
Q5TempSQL: select count(*) as Cnt
           while    [ a.W ]
           from      Week as a, CheckOut:SeqNo as b
           where     [ a.W ]  $\cap$  [ a.SeqNo ]  $\neq \emptyset$ 
           group by a.W

Q6TempSQL: select count(*) as Cnt
           while    [ a.CInt ]
           from      MWConstantInt as a, CheckOut:SeqNo as b
           where     expand([ a.CInt ], 6, 0)  $\cap$  [ a.SeqNo ]  $\neq \emptyset$ 
           group by a.CInt
```

The **expand** function in *Q6* expands the interval of a result tuple by 6 days into the past (and 0 days into the future).

Q8. 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.

```
Q8TempSQL: select *
           from    CheckOut:SeqNo
```

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.

59.6 Evaluation Algorithms for Temporal Operators

We have so far characterized concepts related to the logical representation of temporal data and, in particular, the querying of temporal data.

We proceed to consider selected techniques that aim at supporting the efficient processing of temporal functionality that can be expected to be used widely, that is potentially expensive to support, but has the potential for being supported efficiently, and that is not supported well by existing, non-temporal techniques.

Temporal joins and temporal aggregation operations are prominent examples of functionalities that fit the bill. This section thus offers an overview of join and aggregation techniques.

59.6.1 Temporal Join

Joins are used when correlating data stored in different relations. A join on two relations r and s returns those pairs of tuples from r and s that satisfy some join predicate p on the tuples of these relations. In an equi-join, predicate p is of the form $r.A = s.B$ where A and B are attributes of r and s . Thus, to perform a join, all matching pairs of tuples from the two argument relations must be identified, which is a potentially expensive undertaking.

The temporal generalization, in terms of snapshot reducibility, of a conventional join involves an additional intersection predicate on the timestamps of the tuples, and the resulting tuples are timestamped with the intersections of the timestamps of the argument tuples.

Temporal joins differ from conventional joins and offer new challenges. First, conventional join techniques are designed for the evaluation of joins with equality predicates, rather than intersection predicates, which translate into inequality predicates on the start and end times of the interval timestamps. Second, temporal databases may often be larger than their non-temporal counterparts. These factors indicate that specialized techniques are needed to efficiently evaluate temporal joins.

Conventional-Join Algorithms

It is possible to create temporal-join algorithms by building on conventional-join algorithms, which may be categorized as nested-loop, partitioning, sort-merge, and index-based joins.

In nested-loop join, blocks of tuples from the outer relation are read into main memory so that there is room for one block of tuples from the inner relation and for a block of result tuples. The inner tuples are iterated through, and result tuples are produced. Then a new batch of tuples from the outer relation are read into memory, and the inner tuples are iterated through.

Partition-based joins assign the tuples of each argument relation to buckets according to their values for the join attributes. Corresponding pairs of buckets of the argument relations then contain all tuples that could possibly match with one another, and the buckets are constructed to best utilize the available main memory buffer space. The result is produced by performing an in-memory join of each pair of corresponding buckets from the argument relations.

Sort-merge joins also divide the argument relation, but use physical memory loads as the units of division. The memory loads are sorted, producing sorted runs, and written to disk. The result is produced by merging the sorted runs. During the merging, matching tuples are found and tuples are produced.

Index-based join evaluation utilizes indexes defined on the join attributes of the input

relations to locate joining tuples efficiently. The index could be pre-existing or built on the fly.

Index-Based Temporal-Join Algorithms

Elmasri et al. [EWK90] propose a temporal join algorithm that utilizes a two-level so-called *time index*, which uses a B^+ -tree to index the non-temporal join attribute in the upper level, with the leaves referencing other B^+ -trees indexing time points. Son and Elmasri [SE96] revise the time index to require less space and utilize this modified index to determine the partitioning intervals in a partition-based algorithm.

Van den Bercken and Seeger propose several temporal join algorithms based on multi-version B^+ -tree (MVBT) [dBS96]. Zhang et al. [ZTS02] describe several algorithms based on B^+ -trees, R^* -trees, and the MVBT for a join that requires joining tuples to have key values that belong to a specified range and have time intervals that intersect a specified interval. The MVBT assumes that updates arrive in increasing time order, which is not the case for transaction-time data, but not valid-time data.

Non-Index-Based Temporal-Join Algorithms

Different authors have proposed temporal join algorithms that build on the conventional nested-loop, sort-merge, and partitioning join algorithms [GJSS05].

The nested-loop algorithm simply includes the intersection predicate on the tuple timestamps. When using sort-merge and partitioning joins for computing temporal joins, there are several options. First observe that the join predicate involves an equality predicate on non-temporal attributes and inequality predicates on the start and end times of the interval timestamps. Thus, for sort-merge joins, it is possible to simply use the non-temporal attributes (and apply the temporal predicate subsequently). Or the argument relations could be ordered on the start (or end) times (then the non-temporal predicate is applied subsequently). Finally, it is possible to order first on the non-temporal attributes and then on time, or vice versa. This yields four ways of using sort-merge join. The same applies to partitioning joins, yielding another four temporal joins. Finally, two different buffer allocation strategies, called hybrid and GRACE, can be used by these algorithms, yielding a total of 17 temporal join algorithms, counting also nested-loop join.

The key challenge is that of dealing with intervals and intersection predicates. With the sort-merge join, the problem is that many tuples of the outer relation may have intervals that intersect with the intervals of many tuples of the inner relation. With the partitioning join, the buckets are assigned intervals that partition the time line. Each bucket then holds all tuples with intervals that intersect the bucket's interval. One problem is then that a tuple generally cannot be assigned to a single bucket. Another problem is that buckets may be overly large.

59.6.2 Temporal Aggregation

In part due to the increasing diffusion of business intelligence, aggregate computation is becoming increasingly important. An aggregate operator transforms an argument relation into a summary result relation. This is traditionally done by first partitioning the argument relation into groups of tuples with identical values for one or more attributes, then applying an aggregate function, e.g., sum or average, to each group in turn. However, in temporal databases, aggregation is more complex because the tuples' timestamps, in addition to the non-temporal attributes, can also be used for defining the groups of tuples on which to compute the aggregate functions.

Instant, Moving-Window, and Span Aggregation

Various forms of temporal aggregation have been investigated during the past two decades, and different names have been used to refer to them. A core concept that is shared by almost all approaches is *temporal grouping*, which is defined as the process whereby the time line is partitioned and the argument tuples are grouped according to these partitions into so-called *aggregation groups*. Then the aggregate functions are computed over each of these aggregation groups. The different kinds of temporal groupings possible lead to different forms of temporal aggregation: instantaneous aggregation, moving-window aggregation, and span aggregation. Examples that illustrate these forms of temporal aggregation are the Queries *Q4*, *Q6*, and *Q5*, respectively.

In *instantaneous* temporal aggregation (or simply instant aggregation), each time instant t is associated with the set of all tuples that hold at t . Then the aggregate functions are subsequently evaluated on each aggregation group. Finally, identical aggregate results with consecutive time instants are coalesced into so-called constant intervals that are maximal intervals over which all result values are constant.

With an inherently interval-based model, the coalescing step preserves the intervals of the data tuples and does not coalesce two consecutive result tuples if they have different *lineage*, meaning that they are produced by different argument tuples.

In point-based or weakly interval-based models, only the aggregate values are considered for the maximal intervals. Query *Q4* is an example of an instantaneous temporal aggregation query. The result of evaluating *Q4* in a point-based model is shown in Fig. 59.12(a).

Next, with *moving-window* temporal aggregation (also termed *cumulative* temporal aggregation), a time window is used to determine the aggregation groups. For each time instant t an aggregation group is defined as the set of data tuples that hold in the interval $[t-w, t]$, where $w \geq 0$ is called a window offset. In some work, e.g., [VSM05], positive and negative window offsets are used, yielding a window $[t-w, t+w']$, $w \geq 0$, $w' \geq 0$, when determining the aggregation groups.

After computing the aggregate functions for each aggregation group, coalescing is applied similarly to how it was done in instantaneous aggregation to obtain result tuples over maximal time intervals. Hence, the important difference between instantaneous and moving-window aggregation is in how the aggregation groups for each time instant are defined. Figure 59.12(b) shows the result of computing the moving-window **count** over the **CheckOut** relation, where an offset of 3 into the past is used.

Finally, for *span* temporal aggregation, the time line is first partitioned into predefined intervals that are independent of the argument relation. Example intervals include years, months, and days. For each resulting interval, an aggregation group is then given as the set of all argument tuples that overlap the interval. Next, a result tuple is produced for each interval by evaluating the aggregation functions over the corresponding aggregation group. Figure 59.12(c) shows the result of computing the **count** function after applying span grouping on the **CheckOut** relation, where the time span is defined as a period of length six.

Temporal Aggregation Processing Techniques

The processing of temporal aggregations poses new challenges. Most importantly, the time intervals of the result tuples can depend on the argument tuples and are thus not known in advance. Moreover, a temporal aggregation result might be larger than the data relation. Specifically, for instantaneous temporal aggregates that are grouped into so-called constant intervals, the result relation size can be up to twice that of the data relation.

Two passes: The earliest proposal for computing instantaneous temporal aggregates

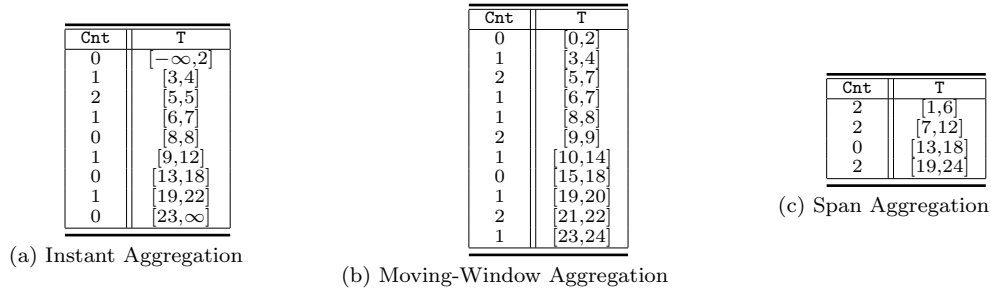


FIGURE 59.12 Results of Different Forms of Temporal Aggregation

was presented by Tuma [Tum92] and is based on Epstein's algorithms for computing aggregation over explicit attributes [Eps79]. Tuma's algorithm requires two scans of the input relation—one for computing the time intervals of the result tuples and one for computing the aggregate values. After Tuma's pioneering work, research concentrated on the development of algorithms that construct main-memory data structures on the fly that make it possible to perform both steps at once, thus requiring only one scan of the argument relation.

Aggregation tree: The *aggregation tree* algorithm by Kline and Snodgrass [KS95] incrementally constructs a tree structure in main memory while scanning the data relation. The tree stores a hierarchy of intervals and partial aggregation results. The intervals at the leaf nodes encode the constant intervals of the result tuples. A depth-first traversal of the tree at the end then yields the final result tuples in chronological order. Each leaf node produces a result tuple, where the value of the aggregate function is computed as the sum of the partial results along the path from the root to the leaf.

Figure 59.13(a) shows the aggregation tree after scanning the first two data tuples of the **CheckOut** relation. The path from the root to the leaf with time interval $[5, 5]$ produces the result tuple $(2, [5, 5])$. Since the tree is not balanced, the worst case time complexity is $O(n^2)$ for n data tuples. As the entire aggregation tree is kept in main memory, the data relations that can be handled by this approach are limited by the main memory size.

An improvement, although with the same worst case complexity, is the k -ordered aggregation tree described in the same paper. It reduces the memory requirements by exploiting the chronological ordering of the tuples for garbage collecting old nodes that will not be affected by any future tuples. For this approach to work, the tuples in the table must be chronologically ordered to some degree.

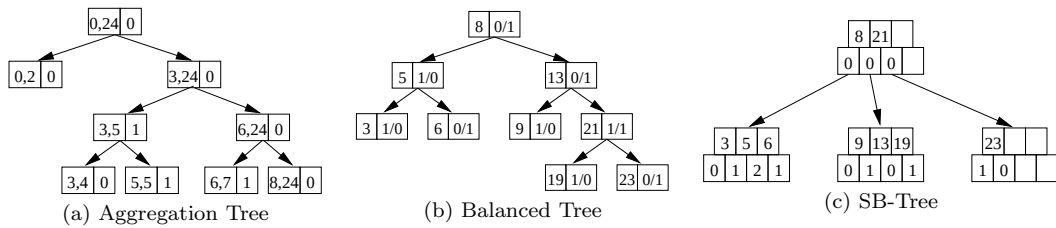


FIGURE 59.13 Different Forms of Data Structures for Temporal Aggregation

Balanced tree: Moon et al. [MVI03] propose the *balanced tree* algorithm for the evaluation of instantaneous temporal aggregation in main memory. The algorithm is based on

timestamp sorting. As the tuples are scanned, the start and end times of their intervals are stored in a balanced tree together with two values for each aggregate function being computed: one storing the sum of values that start here and one storing the sum of values that end here. The result relation is computed by performing an in-order traversal of the tree and accumulating the partial aggregate values in the nodes. Whenever a node is visited, a result tuple is produced over the constant interval that is formed by the time point of the previously visited node and the time point immediately preceding the time point of the current node. Before we proceed to the next node, the sum of the values that start at the current node are added to the aggregate results, and the sum of the values that end here are subtracted from the aggregate results.

Figure 59.13(b) shows the balanced tree after scanning the entire **CheckOut** relation. Then the aggregate value of the first three result tuples $(1, [3, 4])$, $(2, [5, 5])$, and $(1, [6, 7])$ are computed as $1 - 0 = 1$, $1 - 0 + 1 - 0 = 2$, and $1 - 0 + 1 - 0 + 0 - 1 = 1$, respectively.

The balanced tree algorithm works only for the aggregate functions *sum*, *avg*, and *count*. For *min* and *max*, a merge-sort like algorithm is proposed. Both algorithms have a time complexity of $\mathcal{O}(n \log n)$ for n argument tuples. Like the aggregation tree, the balanced tree is kept in main memory.

For large-scale aggregation, an efficient bucket algorithm is proposed. This algorithm assigns the argument tuples to different buckets according to a partitioning of the time line and maintains a meta-array structure in order to efficiently store information about long-lived tuples that cover multiple partitions. Aggregation is then performed on each bucket in isolation using one of the main-memory based solutions, taking into consideration also the data in the meta-array. The algorithm requires access to the entire database three times, provided that a bucket fits in main memory together with the meta-array.

SB-tree: Yang and Widom [YW03] have more recently proposed a new data structure, termed the *SB-tree*, together with algorithms for the incremental computation and maintenance of instantaneous and cumulative temporal aggregates. This disk-based structure combines features from both segment trees and B-trees. The SB-tree stores a hierarchy of time intervals associated with partially computed aggregates.

Figure 59.13(c) shows the SB-tree for the **CheckOut** relation. To answer an instant aggregation query at a time instant t , the partial aggregate values associated with the time intervals that contain t are accumulated, which can be done by a recursive search from the root to a leaf node.

To compute the aggregate at time 10, we start at the root node and collect the value 0 that is associated with the interval $[8, 20]$. We follow the link to the corresponding leaf node and add the value 1 that is associated with the interval $[9, 12]$, thus getting $0 + 1 = 1$ as the aggregate value at time 10. The SB-tree algorithm is particularly well suited for data warehouse applications where modifications are mainly insertions and that require fast lookup of temporal aggregate results that can be maintained efficiently when the data change. Since the *min* and *max* aggregation functions cannot be maintained incrementally, they are not supported if deletions occur.

MVSB-tree: With the SB-tree, aggregate queries are always applied to an entire data relation. As a result, it is not possible to (efficiently) include selection predicates. The *multi-version SB-tree* (MVSB-tree) [ZMT⁺01] tackles this problem and aims to support temporal aggregation coupled with range predicates that select the tuples over which an aggregate should be computed. The MVSB-tree is logically a sequence of SB-trees, one for each timestamp. For a range r of a non-timestamp attribute and a time interval t , the MVSB-tree computes aggregates for all argument tuples that are within r and that overlap with t .

The main drawbacks of this approach are: the tree might be larger than the argument

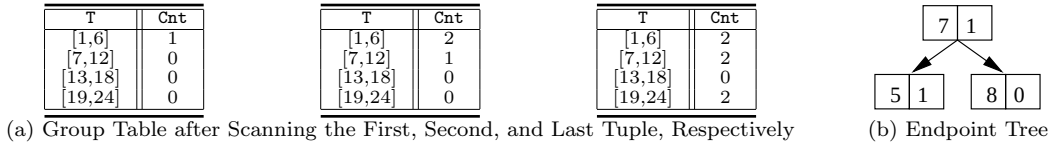


FIGURE 59.14 Evaluation of the TMDA Operator.

relation, the range restriction is limited to predicates over a single non-timestamp attribute, and the timestamp of a data tuple is deleted once it is selected, yielding a non-temporal result relation. Thus the temporal evolution of the aggregate values is lost. Tao et al. [TPF04] propose an approximate variant of the MVSB-tree that uses less space than the multi-version SB-tree.

TMDA: All approaches described so far share three properties. First, with a few exceptions they compute only instantaneous temporal aggregation. Second, only point-based and weakly interval-based data models are supported. Third, grouping by non-temporal attributes (in addition to temporal grouping) is usually not supported.

Böhlen et al. [BGJ06b] advocate an alternative framework, termed *multi-dimensional temporal aggregation operator* (TMDA), that overcomes these limitations and generalizes the aggregation operators discussed above. It provides orthogonal support for two important aspects of temporal aggregation: the definition of *result groups* that induces also the partitioning of the time line for which to report one or more aggregate values; and the definition of *aggregation groups*, i.e., collections of argument tuples that are associated with the result groups and over which the aggregate functions are computed. This enables the specification of result tuples over possibly overlapping time intervals and makes it possible to control the size of the result relation.

Two evaluation algorithms for the TMDA operator are provided: one for constant-interval semantics (covering instantaneous and moving-window aggregation) and one for span aggregation. Two main data structures are used: a group table and, for constant-interval semantics, an additional so-called endpoint tree.

For span aggregation, the group table stores the result groups that are extended by an additional column for each aggregate value being computed. As the argument relation is scanned, the group table is extended with new entries, and for each argument tuple, all aggregate values to which this tuple contributes are updated. At the end, the group table contains the result relation. To obtain an efficient solution, it is important to construct one or more indexes on the group table.

Figure 59.14(a) shows three steps in the evolution of the group table for the computation of the span aggregation result shown in Figure 59.12(c). The left-hand side of the table contains the result groups (in this case only a simple partitioning of the time line), and the right-hand side is initialized to 0. For each tuple of the data relation, this data structure is updated. The memory requirements of this algorithm are independent of the argument relation and only depend on the size of the result relation. The average case time complexity is $n \log m$, where n is the number of argument tuples and m is the number of result groups.

The algorithm for constant interval semantics exploits the sorting of the argument relation and is based on the following observation: if the argument relation is scanned in chronological order of the start times of the tuples, it is possible to compute at any time point t the result tuples that end before t (assuming that an argument tuple does not contribute to the result before its starting point).

Hence, as the argument relation is being scanned, result tuples are produced in chronological order, and only information about those tuples that are valid at the current time point

t , termed the *open tuples*, are kept in main memory in a data structure, called an *endpoint tree*. Each node in the endpoint tree represents an end point t of a constant interval and maintains the (partial) aggregate result evaluated over all tuples that end at t . An argument tuple inserts two nodes into the tree, namely the time point immediately preceding its start point and its end point. In the former case, the aggregate value is empty, in case no argument tuples end at the time point.

Figure 59.14(b) shows the endpoint tree after scanning the first three tuples of the **CheckOut** relation. The first two result tuples in Figure 59.12(a) have already been produced, and we insert into the endpoint tree a new node with timestamp 8 (the time point immediately preceding the third argument tuple). Now we can close all tuples in the time interval $[5, 8]$ (the time interval from the end of the last produced result tuple to the start of the last scanned data tuple). We traverse the endpoint tree in depth-first order. For each node (with a time) in $[5, 8]$ we produce a result tuple by accumulating all (partial) aggregate values in the tree; then we remove the node. In our example, we first produce the tuple $(2, [5, 5])$, then the tuple $(1, [6, 7])$.

This way, the memory requirements are determined by the maximal number of open tuples at any time instant. The average time complexity is $n * n_o$, where n is the number of argument tuples and n_o is the number of open tuples, i.e., the maximal number of overlapping tuples at a specific time point. The worst-case complexity is $O(n^2)$ if the start and end points of all argument tuples are different and there exists a time instant where all tuples hold.

59.7 Summary

This chapter has introduced the reader to temporal data management with an emphasis on temporal-data semantics, temporal data model, query language concepts, and algorithms for the efficient processing of temporal join and aggregation queries.

Based mainly on the three central notions of *fact*, *valid time*, and *transaction time*, the chapter offers 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 aspects. 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 yielded a foundation for a study of important approaches to the querying of temporal data. With the purpose of illustrating diverse query language concepts, we considered the formulation of eight queries in concrete language contexts that embody the concepts. The different query language concepts explored assume different relation constructs and data models, although they may make sense for other relation constructs than the ones used for illustrating them in this chapter.

In the following we provide a short summary of our investigation of these query languages and data models by formulating a few key observations.

Observation 1: *When transitioning from intervals to points, any semantics associated with the intervals is lost.* Although temporal relations capture the temporal aspects of data using time points, time intervals, and so-called temporal elements, the chapter 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 representations 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 for timestamping and allow relations

that are not coalesced. We have also seen that the interval semantics may, to some extent, be simulated in point-based models by using an additional “sequence number” attribute.

Observation 2: *Adding a new ADT to SQL is attractive because it has limited impact on the language design and because the extension of SQL with new data types with accompanying predicates and functions is fairly well understood.* The abstract data type approach to querying temporal data is simple, but also limited in the support offered.

Observation 3: *Extending SQL with the functions `fold` and `unfold` is attractive because of its conceptual simplicity.* 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. This approach is limited in that it leaves some “technicalities” (which are tricky at times) to be addressed by the application programmer.

Observation 4: *SQL/TP does not permit the association of information with intervals, and the semantics of SQL/TP statements is defined with respect to the point representation, which is different from the presentation of a temporal relation.* The approach that solely uses point timestamps, such as SQL/TP, assumes that representation and display are beyond the scope of the query language. This yields a clean, point-based query language, but also a rather “theoretical” one.

Observation 5: *Well-chosen syntactic defaults yield a language that allows to succinctly formulate common temporal queries. However, adding temporal support to SQL in manner is difficult since the non-temporal constructs do not permit a systematic and easy way to express the defaults.* 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.

Observation 6: *Statement modifiers are orthogonal to the SQL language and adding them to SQL is less understood than adding a new ADT.* The notion of statement modifiers offers a wholesale approach to rendering a query language temporal: modifiers are introduced that control the semantics of any query language statement. The strong point is the support for intervals and the ability to easily construct temporal queries that generalize snapshot queries.

Observation 7: *Temporal expressions, which return the temporal elements during which a logical expression is true, are convenient in formulating a range of queries.* Temporal expressions along with temporal elements, which are finite unions of intervals, fit well into the point-based framework. They often enable the elegant formulation of queries. However, only little research work has been done to further explore temporal expressions and to include them into query languages.

59.8 Outlook

Although the research in temporal database management thus far has led to many important insights and results, 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 chapter. 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 existing architectures. 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.

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 in a comprehensive manner. More work is also needed on exploiting temporal indexes 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 ample challenges.

In *active databases and event processing*, 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 are now equipped with on-line positioning devices that track their positions and make these

available for storage in databases. The continued advances in wireless communications, positioning, and other key technology areas constitute powerful drivers for such applications. 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 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 variance 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.

References

1. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. M. H. Böhlen, J. Gamper, and C. S. Jensen. How would you like to aggregate your temporal data? In *International Symposium on Temporal Representation and Reasoning*, pages 121–136, 2006.
3. M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *Extending Database Technology*, pages 257–275, 2006.
4. M. H. Böhlen and C. S. Jensen. Temporal data model and query language concepts. In *Encyclopedia of Information Systems*, pages Volume 4, 437–453. Academic Press, 2003.
5. M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Transactions on Database Systems*, 25(4):407–456, 2000.
6. M. H. Böhlen. Temporal database system implementations. *SIGMOD Record*, 24(4):53–60, 1995.

7. J. Clifford and A. Tuzhilin, editors. *Recent Advances in Temporal Databases, Proceedings of the International Workshop on Temporal Databases*, Workshops in Computing, 1995.
8. Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.
9. J. Van den Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *Very Large Data Bases*, pages 168–179. Morgan Kaufmann Publishers, 1996.
10. C. J. Date, H. Darwen, and N. Lorentzos, editors. *Temporal Data & the Relational Model*. Morgan Kaufmann Publishers, 2002.
11. O. Etzion, S. Jajodia, and S. Sripada, editors. *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
12. R. Epstein. Techniques for processing of aggregates in relational database systems. Technical Report UCB/ERL M7918, University of California, Berkeley, 1979.
13. R. Elmasri, G. T. J. Wu, and Y.-J. Kim. The time index: An access structure for temporal data. In *Very Large Data Bases*, pages 1–12. Morgan Kaufmann Publishers, 1990.
14. H. Gregersen and C. S. Jensen. Temporal entity-relationship models – a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(3):464–497, 1999.
15. D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. *VLDB Journal*, 14(1):2–29, 2005.
16. C. S. Jensen and C. Dyreson. A Consensus Glossary of Temporal Database Concepts—February 1998 Version. In [EJS98], pages 367–405. Springer Verlag, 1998.
17. C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
18. C. S. Jensen and R. T. Snodgrass. Temporally enhanced database design. In *Advances in Object-Oriented Data Modeling*, pages 163–193. The MIT Press, 2000.
19. C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unification of temporal data models. In *International Conference on Data Engineering*, pages 262–271, 1993.
20. N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *International Conference on Data Engineering*, pages 222–231, 1995.
21. N. A. Lorentzos and R. G. Johnson. Extending relational algebra to manipulate temporal data. *Information Systems*, 13(3):289–296, 1988.
22. B. Moon, I. F. Vega Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):744–759, 2003.
23. G. Özsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.
24. N. Pissinou, R. T. Snodgrass, R. Elmasri, I. S. Mumick, M. Tamer Özsu, B. Pernici, A. Segev, B. Theodoulidis, and U. Dayal. Towards an infrastructure for temporal databases: Report of an invitational ARPA/NSF workshop. *ACM SIGMOD Record*, 23(1):35–51, 1994.
25. C. Rolland, F. Bodart, and M. Léonard, editors. *Temporal Aspects in Information Systems, Proceedings of the IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems*, 1987.
26. D. Son and R. Elmasri. Efficient temporal join processing using time index. In *Statistical and Scientific Database Management*, pages 252–261. IEEE Computer Society, 1996.
27. R. T. Snodgrass, editor. *Proceedings of the International Workshop on an Infras-*

- structure for *Temporal Databases*, 1993.
28. R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
 29. R. T. Snodgrass. *Advanced Database Systems*, chapter Temporal Databases. Morgan Kaufmann Publishers, 1997.
 30. R. T. Snodgrass, editor. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, 1999.
 31. A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, Inc., 1993.
 32. D. Toman. Point-based temporal extensions of sql and their efficient implementation. In *Temporal Databases, Dagstuhl*, pages 211–237, 1997.
 33. Y. Tao, D. Papadias, and C. Faloutsos. Approximate temporal aggregation. In *International Conference on Data Engineering*, pages 190–201, 2004.
 34. P. A. Tuma. *Implementing Historical Aggregates in TempIS*. PhD thesis, Wayne State University, 1992.
 35. I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, 2005.
 36. Y. Wu, S. Jajodia, and S. Wang. Temporal database bibliography update. In [EJS98], pages 338–366. Springer Verlag, 1998.
 37. J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB Journal*, 12(3):262–283, 2003.
 38. E. Zimányi. Temporal aggregates and temporal universal quantification in standard SQL. *ACM SIGMOD Record*, 35(2):16–21, 2006.
 39. D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Symposium on Principles of Database Systems*, pages 237–245, 2001.
 40. D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *International Conference on Data Engineering*, pages 103–, 2002.