TEMPORAL DATABASES

Time is an important component of many real-world applications. Applications like accounting, billing, and marketing deal with current as well as past (historical) data, or even data about the future. Database systems store information about the real world and provide easy and efficient access to it. Conventional databases, however, cannot efficiently support temporal applications. Instead, *temporal databases* have been proposed (1). The term temporal database refers to "a database that supports some aspect of time" (2). In the late 1990s, research in the area of temporal databases has shown an enormous growth, evidenced by the many publications, books (3–6), and conferences related to the field. A recent bibliography (7) accounts for approximately 1,100 temporal database publications until the end of 1995. As it is practically infeasible to cover all this research, this article attempts a general overview; for more detailed coverage of the various topics, appropriate references are provided. Furthermore, there are other computing areas where time plays an important role, for example, temporal reasoning and planning, spatiotemporal databases (8).

The issues involved in managing temporal data include: (1) how the temporal information is conceptually and physically represented, (2) in what form user requests for information are presented to the database management system (DBMS), and (3) how these requests are processed by the system. The first two issues are the focus of *temporal data models* research, whereas the last issue regards *system implementation* and *query optimization.* We start our discussion with a description of the time domain. Then we introduce temporal data models and discuss temporal database design. Implementation and query processing are covered next. Finally, we examine the notion of time as it appears in real-time and active databases.

THE TIME DOMAIN

The explicit time dimension in a temporal database is usually represented by labeling nontemporal information by time stamps. The first question is what constitutes a time stamp, or, equivalently, what is the domain for time stamps.

There are two philosophical views regarding the flow of time. One is continuous, and the other is discrete. Translating to the choice of domain for time stamps is either to use a continuous set (like the reals) or to use a discrete set (like the integers). In practice, however, since any number stored in a computer system has to be discretized, most researchers in the field of temporal databases have adopted a discrete set as the time domain. The study of temporal logics accommodates other views of time flow (e.g., branching and complete flows). See Ref. 9 for a more extensive discussion.

Another question regarding a time domain is what an element in the time domain represents. For example, each element in the chosen time domain can represent a particular second as in the Unix operating system. There the time domain used to label operating system events (such as the creation of a file) is the set of positive integers, and each number in the set represents the number of seconds since 00 : 00 : 00 UTC of January 1, 1970. Or more precisely, each number represents the particular period of time represented by that particular second. Here we see a problem of indeterminacy (10). That is, for example, we know that a file is created within a particular second, but we are not sure of at which moment within the second. This indeterminacy is often desirable since the information available may not or need not have to be precisely identified. This leads to multiple granularities in the domain to allow different degrees of indeterminacy.

There are other reasons to have multiple granularities. For example, some facts may hold over the entire time of a month. Instead of labeling the facts by the time period (starting second and ending second), we may simply label them by the month. It is much easier to use and semantically more meaningful.

In the current Structured Query Language (SQL) standard, the data type of a column can be `date` (with year, month, and day components), `time` (with hour, minute, and second components), or `timestamp` (i.e., a combination of `date` and `time`). (For the sake of completeness, we want to mention that SQL also has an `interval` data type, i.e., the directed distance between two time points, e.g., 3 months.) However, a `date` value does not pinpoint down to a particular period of time unless the time zone is specified. In a sense, for each time zone, there is a `date` granularity in SQL.

The research of multiple granularities for databases started fairly recently, perhaps with an article by Clifford and Rao in 1987 (11). The formal treatment of multiple granularities described below is mostly from Ref. 12.

A *basic time domain* is a totally ordered set. This totally ordered set can be dense or discrete. The basic time domain serves as the underlying absolute time flow. A *granularity* is a pair (I, G), where I is a discrete totally ordered set, called the *index set,* and G is a mapping from I to all the subsets (including the empty set) of the basic time domain such that the following two conditions are satisfied: for each pair i and j in I, if $i < j$, $G(i) \neq 0$, and $G(j) \neq 0$, then each element in $G(i)$ is less than all the elements in $G(j)$, and for i, j, and k in I, if $i < k < j$, $G(i) \neq 0$ and $G(j) \neq 0$, then $G(k) \neq 0$ (13).

The first condition states that the mapping G should be monotonic, and the second condition imposes that the indices that are mapped to nonempty sets be contiguous. It is rather clear that `days`, `months`, and `weeks` and all the everyday granularities satisfy the above definition.

There are natural relationships between granularities. The first one is so-called *finer-than.* Granularity (I_1, G_1) is said to be finer than granularity (I_2, G_2) if for each index i in I_1 there exists j in I_2 such that $G_1(i) \subseteq G_2(j)$. For example, day is finer than week and business-day is finer than month. Another relationship is *group-into.* Granularity (I_1, G_1) is said to group into granularity (I_2, G_2) If for each j in I_2 there exists a subset S of I_1 such that $G_2(j) = \cup_{i \in S} G(i)$. For example, day groups into month, however, business-day does not group into week. A special case of group-into relationship is interesting because it deals with *periodicity.* A granularity (Z, G_1) is said to group periodically into a granularity (Z, G_2) where Z is the set of the integers, if: (i) (Z, G_1) groups into (Z, G_2) and (ii) there exist positive integers n and m, where n is less than the number of nonempty granules of G_2, such that for all i in Z, if

$$G_2(i) = \bigcup_{r=0}^{k} G_1(j_r) \quad \text{and} \quad G_2(i+n) \neq \varnothing$$

then

$$G_2(i+n) = \bigcup_{r=0}^{k} G_1(j_r + m)$$

In other words, the pattern of how G_1 groups into G_2 repeats indefinitely. Most everyday granularities show such a relationship. For example, day groups periodically into week. In this case, $n = 1$ and $m = 7$. Also, day groups periodically into year. In this case, $n = 400$ and $m = 14{,}697$. These numbers are rather large because we have to incorporate the leap years.

TEMPORAL DATA MODELS

In any existing data model, one can always store time information as regular data. For example, in a relational database, one can set up a column in a table (relation) to store time or date values. Many of the temporal information management requirements can actually be handled by using such a nontemporal data model. However, due to the special nature of the time dimension, temporal information is generally inadequately supported in a nontemporal environment, resulting in many difficulties in system development and maintenance, as well as inefficiency in system operations (14).

As an example (15), suppose an employee table has four columns: `Name`, `Manager`, `Dept`, and `When`, where `When` is of `period` type. [A time *period* is defined as the time between two instants (2).] A record in this table records an employee with his/her manager at a department together with a time period. The query "give the history of the number of employees in each department" is very difficult to express in the query language of the current SQL standard. A temporal data model can be considered as an attempt to correct the above situation by giving a special status to the time dimension and providing special language constructs and software mechanisms to handle the temporal information in a convenient and efficient manner.

Typically, a temporal data model is formed by adding a time dimension to an existing nontemporal data model, for example, the relational data model. The query language of the nontemporal data model is then extended with constructs to handle the temporal dimension. Here we consider temporal extensions to the relational model (extensions for other models are summarized later).

A relational database consists of a collection of relation names, each of which is associated with a relation (i.e., a set of tuples). We use R, possibly with subscripts, to denote relation names and, when no confusion arises, to denote their associated relations.

There are in general two families of temporal data models, the abstract and the concrete (16). An abstract model is intended as an abstract vehicle to study the properties of the time dimension and to set up guidelines for designing a concrete data model, while a concrete data model is more concerned with the management and manipulation of the temporal information in a practical setting.

Abstract Temporal Data Models

To simplify the discussion, we assume only one granularity is used and the index set for the granularity is the integers. An extension to multiple granularities and other index sets is not difficult to work out.

Conceptually, a relation in a temporal database can be viewed as consisting of many nontemporal relations, namely one nontemporal relation for each time point. More specifically, assume that the relation scheme is $R = \langle A_1, \ldots, A_n \rangle$. Then for each integer t, there is a relation $r[t]$, and the temporal relation is an infinite sequence $\ldots, r[0], \ldots, r[t], \ldots$. Certainly, some of the relations in the sequence may be empty. In this snapshot view, given a time point t, we know what facts hold at that time.

As an example, consider the facts listed in Table 1. It is easily seen that in terms of the snapshot view, we have two nonempty snapshots, namely for year 1990 and year 1991. Table 2 shows the two nonempty snapshots. For all other years, the associated relations are all empty and omitted.

With the above snapshot view, it is easy to formulate queries on a temporal database. First-order logic with linear order has been used. The basic idea is that each relation name is a predicate with the last place being a temporal sort. For example, we can use a five place predicate T to represent the information of Table 1. Hence,

T(Baltimore, Portable Humidifier, \$5M, 10%, 1990)

is true since in 1990, the Portable Humidifier produced by the Baltimore plant did have a cost of \$5M with a profit of 10%. With this predicate view, we can easily formulate queries. For

Table 1. Facts about Profits

Plant	Product	Cost	Profit	Year
Baltimore	Portable Humidifier	\$5M	10%	1990–1991
Baltimore	Cellular CarPhone	\$10M	20%	1991
Baltimore	Digital Amplifier	\$2M	15%	1991
Baltimore	Digital Amplifier	\$1.5M	15%	1990
Los Angeles	Paper Towel	\$1M	5%	1990–1991
Los Angeles	Portable Humidifier	\$3M	11%	1991
Denver	Rock Candy	\$0.5M	30%	1990

Table 2. Two Snapshot Views of Table 1

$t = 1990$

Plant	Product	Cost	Profit
Baltimore	Portable Humidifier	\$5M	10%
Baltimore	Digital Amplifier	\$1.5M	15%
Los Angeles	Paper Towel	\$1M	5%
Denver	Rock Candy	\$0.5M	30%

$t = 1991$

Plant	Product	Cost	Profit
Baltimore	Portable Humidifier	\$5M	10%
Baltimore	Cellular CarPhone	\$10M	20%
Baltimore	Digital Amplifier	\$2M	15%
Los Angeles	Paper Towel	\$1M	5%
Los Angeles	Portable Humidifier	\$3M	11%

example, the query "which plant (and in which year) increased its cost on the Digital Amplifier from the previous year?" can be expressed as follows:

$$\{\langle p, y'\rangle \mid \exists c, c', x, x', y, [c < c' \wedge y = y' - 1 \\ \wedge\, T(p, \text{"Digital Amplifier"}, c', x', y') \\ \wedge\, T(p, \text{"Digital Amplifier"}, c, x, y)]\}$$

(Note that, the subformula $y = y' - 1$ is an abbreviation of $[y < y' \wedge \neg\exists y'' \,(y < y'' \wedge y'' < y')]$, that is, y' is greater than y and yet there is nothing between y and y'.) Intuitively, this query is to find plant p and year y' such that plant p in year y' had a cost c' on the Digital Amplifier, whereas in the previous year (y), the plant had the smaller cost (c).

Algebraic query languages can also be easily formulated (17,18). The operations of the algebra may simply extend the relational algebra operations, namely projection, selection, join, union, intersection and difference, to work on each snapshot of temporal relations. For example, the temporal projection of a temporal relation gives a sequence of relations, each of which is the regular projection of the corresponding snapshot of input temporal relation. Using the plant-profit example given earlier, we see that the projection to attributes "Plant" and "Profit" will give the temporal relation consisting of two nonempty snapshots corresponding to the projection $\pi_{\text{Plant,Profit}}$ on the two snapshots shown in Table 2.

The algebraic query language outlined above consists of operations that map temporal relations to temporal relations. However, it seems that such a query language cannot describe some of the queries that are expressed using the calculus query given earlier (19). The basic reason is that each operation preserves the temporal relation (i.e., the number of time columns in the input as well as in the output is only one). This limits the expressiveness of the algebra. Extensions of the above simple algebraic query language have been considered in order to capture more queries.

The above temporal relational data model only regards the time when a fact holds. This time is called *valid* time, that is, the time when the fact is valid in the real world. Another kind of time is the *transaction* time, which is the time when the fact is stored in the database (1) or equivalently, when the database "knows" about the fact. Transaction times correspond to the commit time of the transaction that entered the information about the fact into the database. Thus they are generated by the DBMS and are monotonically increasing. Note that past transaction times cannot be changed. If our knowledge about the fact changes, the new knowledge will be entered into the database by a new transaction which will be assigned by the DBMS a later commit time. A fact is then assigned a transaction time interval that corresponds to the period during which the database knew about this fact. If a fact is never changed after entered into the database, the database knows about it for all larger transaction times after it was entered.

The two kinds of time assigned to a fact may not coincide: the transaction time of a fact may start before or after the start of the valid time and may end before or after the end of the valid time. For example, a raise of salary may be known to the database before it takes effect (20), and a transition of temperature reading may happen long before the value can be entered into the database.

The *bitemporal data model* is an effort to add both valid and transaction time (2). Conceptually, each (nontemporal) fact is labeled by two timestamps, namely valid time and transaction time, specifying the time the fact holds and the time that the database knows it. From the transaction-time point of view, a bitemporal relation is a sequence of valid-time temporal relations, each of which gives the temporal facts known at the corresponding (transaction) time. From the valid-time point of view, on the other hand, a bitemporal relation is a sequence of transaction-time temporal relations, each of which gives the facts holding at the (valid) time together with the time(s) when the corresponding facts are known to the database. It is interesting to note that in a bitemporal relation, a fact (together with its valid time) known to the database at (transaction) time i but unknown at (transaction) time $i + 1$ can be seen as (logically) deleted at (transaction) time $i + 1$.

A temporal database is categorized as a transaction-time, valid-time, or bitemporal database, according to which temporal dimension(s) it supports. A transaction-time database represents the history of the database rather than real-world history. Because previously entered transaction times cannot be changed, the past is retained and a transaction-time database can answer queries about what it "knew" as of some past transaction time. In contrast, a valid-time database maintains the entire history of an enterprise as best known now, that is, it stores our current knowledge about the enterprise's current, past, and/or future. Any errors discovered in this history, are corrected by modifying the database. If a correction is applied on a valid-time database, previous values are not retained; therefore it is not possible to view the database as it was before the correction. Clearly both time dimensions are needed to accurately model reality. In a bitemporal database one can query tuples that are valid at some (valid) time as known at some other (transaction) time.

A calculus query language can be formulated along the line similar to the calculus language presented for the valid-time temporal relations shown earlier. For bitemporal relations, predicates have two time positions. For example, SALARY(John, 50K, v_i, t_i) is true if and only if at time t_i the database knows that John's salary is 50K at time v_i.

Detailed studies of abstract temporal data models can be found in the literature (e.g., in Refs. 16 and 21). Also, expressiveness of query languages on abstract data models has been studied (22,23).

Concrete Data Models

The temporal data models discussed in the previous subsection may have an infinite time dimension. For example, a valid-time relation may be an infinite sequence of nontemporal relations. To physically realize such a temporal relation, some finite representation is needed. It should be noted that it is not necessary to restrict the sequence to be finite. Indeed, for example, if each snapshot after time T in the temporal relation is exactly the same as the snapshot at time T, we can then represent these infinite number of snapshots by the snapshot at time T and remember that this snapshot actually holds for time period $[T, +\infty)$.

On the other hand, simply restricting the number of snapshots in a temporal relation to be finite while storing each snapshot as a separate (nontemporal) relation is not a satisfactory solution. Obviously, for a valid time relation, a fact is likely to hold over a time period rather than simply at a time point. It is then more economical to store this tuple once but remember the time period in which the tuple holds.

Hence, a temporal relation is usually stored in a form similar to Table 1. The simplest model is to have a temporal relation defined as a nontemporal relation with an additional column (timestamp) whose data type is a period of time (the end points of the period can be $\pm\infty$).

The SQL query language on nontemporal relations can be extended to handle the above model. As an example, we look at SQL/Temporal, which is a proposal to add valid-time support to SQL3 (15). In SQL/Temporal, a nontemporal relation can be added with an implicit additional column usually referred to as `Valid` to become a valid time relation. This implicit column usually registers a time period for each (nontemporal) tuple. This implicit column is not normally accessed as a regular attribute. Rather, some facilities are provided to handle the time dimension. A regular SQL query (i.e., no special facility provided by SQL/Temporal is used) accessing a valid time relation will only work on the snapshot that corresponds to the time "now." This provides a way to access the current information without having to deal with the temporal aspect of the relations.

When the key word VALIDTIME is added to the SQL query (before SELECT), then the SQL query works on every snapshot of the temporal relations, and the result is a valid-time relation. This way, snapshot-wise queries can be specified rather easily. To access information across snapshots, SQL/Temporal provides the key words NONSEQUENCED VALIDTIME. In this mode, the valid-time relations are accessed as if the valid-time column was available. The query makes use of the column by saying VALIDTIME(S), where S is the alias for the valid-time relation. Various period comparisons (before, after, meet, contain, etc.) can be applied to the valid-time column.

There are also other proposals of extensions to SQL such as TempSQL (24), TSQL (25), HSQL (26), IXSQL (27), TOSQL (28). Efforts to extend Quel to handle temporal relations are also reported (29). A new effort on extending SQL is reported in Ref. 30. The basic idea of an extension is to treat the temporal dimensions in a special way with special operators for comparing temporal domain values; such as "before" and "after," etc. Aggregation along the time dimension is also proposed to handle queries such as monthly mean, yearly total, and cumulative sum.

Algebraic query languages on temporal relations have also been studied extensively. Reference 18 is a good survey of earlier algebras, totally 12 of them. In general, an algebraic query language extends (standard) relational algebra operations to handle the relations with a time dimension. The abundance of temporal algebras is due to the fact that there are many ways to add the time dimension into the relations and each different way requires different language constructs. Algebras that work on abstract models are also investigated (19).

When time periods are used and manipulated by the query constructs, care must be taken because two different temporal relations may be equivalent if the snapshot view is taken. For example, from the snapshot point of view, the two tuples (a, [1, 5]) and (a, [6, 7]) represent the same temporal information as one tuple (a, [1, 7]) when the domain of time is the integers. This brings in the need of a "coalescing" operation, which reduces into one tuple any group of tuples with the same nontemporal attribute values but consecutive timestamps (17).

Due to the above consideration, it is beneficial to collect all the tuples having the same nontemporal attribute values into one. This obviously requires some extension of allowed data types for the timestamp column, because a nontemporal fact may hold over two disjoint time intervals (e.g., on [1, 6] and [10, 20]). A proposal was to use finite unions of intervals (i.e., *temporal elements*) (31). Each temporal element syntactically is a finite union of intervals, whereas its semantics are a set of time points. The set operations, namely union, intersection, and difference, are all defined on the sets that these temporal elements represent. The collection of all the temporal elements has the property that it is closed under all set operations. That is, given two sets that are represented by temporal elements, the union, the intersection, and the difference of the two sets all provide a set that can be represented by temporal elements. Furthermore, it is easily seen that the resulting temporal element can be computed easily from the two input temporal elements.

When temporal elements are used as timestamps, we can require that each temporal relation does not contain two different tuples with the same nontemporal attribute values. In other words, we may require that the system always performs coalescing. A further coalescing has also been proposed that collects all the tuples in the relation about one entity into one tuple. For example, in a temporal relation with nontemporal attributes `Name`, `Department`, `Salary` that records the job and salary histories of employees, one may collect all the information about a particular employee into one tuple. An example of such a tuple is:

John [1, 100]	Toy Dept [1, 30] Shoe Dept [31, 100]	30K [1, 40] 32K [41, 80] 33K [81, 100]

It is easily seen that the temporal relation with such a tuple is not in the first normal form. This kind of relation is seen as *attribute timestamping* (31) in contrast to the relations with *tuple timestamping* as discussed earlier. Query languages are available for attribute timestamped relations. Also, a comparison between attribute timestamping and tuple timestamping can be found in Ref. 14.

Other Research

Although most of the research in temporal database modeling concentrates on extending the relational model, extensions to other models have also been investigated. For example, temporal extensions are done on object-oriented data models (32). Also, a logic-based temporal data model is discussed in Ref. 33. On the other hand, temporal information management has been among the topics studied in the constraint database research (34).

TEMPORAL DATABASE DESIGN

Before creating a database, first its *database schema* is specified by the Database Administrator. The schema (metadata) is a description of the database data in terms of a particular data model (relational, object-oriented, etc.) For example, in a relational database, the schema includes the names of the relations and their attributes as well as attribute data types, index information, and so on. The schema information is stored in the system *catalog* and is heavily used during data processing.

The efficiency of a database is affected by how well its schema is designed, because the schema identifies which relations are formed and with what attributes, as well as which attributes are indexed. In general, database design is a multistep process. The very first step, called the *requirements analysis* (35), is to understand what data the database will store and what constraints apply to this data. Using this information, the second step (called the *conceptual design*) develops a high-level description of the data along with the constraints over it. This step is usually carried out using a high-level data model, the entity-relationship (ER) model (35). Because there are no database systems that directly support the ER model, the high-level description (ER diagram) is translated (using mapping algorithms) to the particular data model (relational, object-oriented, etc.) with which the database will be implemented (36). This process results into an initial database schema. This schema is then improved using the identified data constraints. This step is called *schema refinement* (or *normalization*) and is usually based on dependencies (functional, multivalued, etc.) and normal forms (like Third Normal Form (3NF) or Boyce–Codd Normal Form (BCNF) (35). The final step is the *physical database design* where the database schema is further refined by considering expected workloads that the database must support.

Research in temporal database design has been concentrated on temporal ER modeling and on temporal normalization. Capturing the temporal aspects of a database in a traditional ER diagram is difficult; it usually results in difficult-to-comprehend diagrams (37). A simple solution would be to use nontemporal ER diagrams with textual notations indicating that the particular ER diagram has temporal support. Of course this approach leaves the burden of translating the ER diagram to the actual data model on the database administrator/programmer. The research community has developed a number of temporally enhanced ER models that attempt to model the temporal aspects of information more naturally (38–41).

Most of these proposals add new temporal constructs to the ER model. Some have changed the semantics of the ER model, whereas others have retained the traditional ER semantics. Almost all models assume valid time [with the exception of TempEER (39), which assumes both vaid and transaction time]. Typically, the existing proposals assume that their temporal ER schemas are mapped to the relational model. Their mapping algorithms simply add time-valued attributes to relational schemas (which, however, are not interpreted by the relational model, i.e., they have no built-in semantics in the relational model). Clearly, more research is needed in this area as none of the existing proposals uses one of the existing temporal relational data models as their implementation model. For details we refer to Ref. 37 which presents a detailed comparison of various temporal ER proposals according to nineteen criteria.

Traditional relational normalization concepts are not directly applicable to temporal data models, because such models employ relational structures that are different from conventional relations. For example, consider a temporal relation that contains a column whose data type is a time period; a conventional functional dependency will treat this column as just another attribute, with values like "1990–1992" being atomic (i.e., like strings). In addition, the data constraints may have a temporal aspect that is not present in traditional integrity constraints. Various notions of temporal dependencies have been proposed, like the *dynamic functional dependency* (42), the *temporal dependency* (43), the *interval* and *point* functional dependencies (44), and the *temporal dependencies* (45 and 46). The *temporal functional dependencies* of Ref. 47 further generalize this framework to support temporal granularities. Similarly, various temporal normal forms have been proposed, including the *time normal forms* (43,48), the *first temporal normal form* (49) and the *P* and *Q normal forms* (44). Most of these proposals are in the context of a particular temporal data model. Reference 50 assumes the largest common denominator of existing temporal models (the bitemporal conceptual data model) and presents a general framework to define temporal keys, dependencies, and normal forms.

TEMPORAL QUERY PROCESSING

We first review query processing in a traditional database management system (DBMS). We then discuss how the basic components of a DBMS are affected by supporting time and finally we concentrate on the special characteristics of *temporal* query optimization. The presentation here has been influenced by Refs. 3, 51, and 52; we thus refer the reader to these references for a more detailed discussion.

Basics of Database Management Systems

Database users submit queries to a DBMS in high-level query languages (like SQL), which are user friendly and easy to express queries with. Then the DBMS performs query processing, which first lexically and syntactically analyzes the query and then transforms it into a sequence (query plan) of more primitive operators (usually relational algebra operators) for accessing the stored data. Often the hardest part of query processing is to find an efficient query plan to execute the given query; this is called *query optimization* (35).

The basic architecture of a DBMS is shown in Fig. 1. There are three basic components, the query manager, the storage manager, and the transaction manager. At the bottom there is the disk where data and the catalog (metadata or database schema) reside. Arcs identify the ways that the different parts of the system communicate.

Users prepare queries in two possible ways, either as ad hoc queries through a generic query interface or as part of a program through an application program interface. Application programs query the database through special calls to the DBMS. Since application programs are usually written in a programming (host) language that embeds query language (SQL) statements, they first go through a precompiler that separates the database calls (SQL queries) from the host language statements. Queries are then passed to the query manager for analysis and optimization. Despite its name, the query manager handles also requests for modification of the data or the metadata. Its task is to find the best way to carry out a request (query or modification).

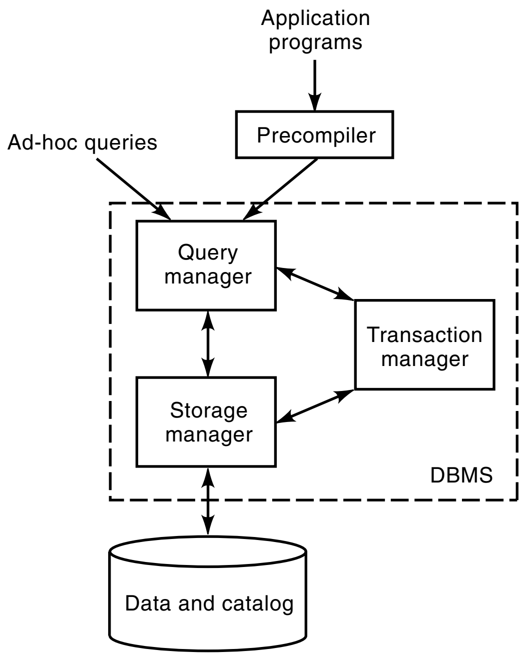

Figure 1. The basic architecture of a database management system (DBMS).

A given query can have many equivalent query plans; such plans can be derived by using well-known equivalence properties of relational algebra operators (this is called *algebraic* optimization). It is the responsibility of the optimization process to enumerate (find) alternative query plans, estimate the cost of executing each such plan and select the best plan. Estimating the cost of a given plan involves estimating the cost of each primitive operator that is part of it (this is called operator *evaluation*). Usually more than one implementation per primitive operator are available. Moreover, other information like the size of the relation or whether there exists an index, plays a key role in the query manager's search for the best query plan. As the number of possible query plans in general can be exponentially large (in terms of the size of the query) and the optimization must be fast, in practice the query manager picks a very efficient (probably not always the best) query plan. It then issues commands to the storage manager that will carry out the request according to the chosen plan.

The storage manager in a simple database system can be nothing more than the file system of the underlying operating system. For efficiency though, DBMSs usually control the database data/metadata on the disk directly. There are two basic subcomponents of a storage manager, the file manager and the buffer manager. The file manager keeps track of the location of database files (that store data, metadata, or indices) on the disk and can obtain the pages of such a file upon request. Since accessing data on the disk (performing disk inputs–outputs, or I/Os) is expensive, the DBMS uses buffers that hold in main memory pages that have been read and may be needed later. Which pages are kept in main memory and for how long is under the control of the buffer manager.

The transaction mangager is responsible for the integrity of the DBMS. It must ensure correct DBMS operation even if several queries are run simultaneously. It should also ensure that data is not lost or corrupted even when a system failure occurs. A typical DBMS allows a user to group together one or more queries and/or modifications into what is called a *transaction.* Through the transaction manager the DBMS guarantees that each transaction will be executed as an *atomic, consistent, isolated,* and *durable* operation. Atomicity implies that either all or none of the transaction is executed. Consistency means that a transaction leaves the database in a consistent state. With isolation, each transaction executes with no interference from other concurrently executed transactions. Finally, if a transaction successfully completes its work, its effects should not be lost even if the system fails. The transaction manager interacts with the query manager so as to control the execution of queries (at times it may delay a query if it conflicts with another concurrent query). It also interacts with the storage manager, because it stores a log of changes to the data needed to protect against data losses or corruptions from system failures.

Temporal Database Management System Implementation Issues

For practical experiences with implementing a temporal database we refer the interested reader to the various temporal database system prototypes that have appeared in the literature. A recent list of such prototypes has been presented elsewhere (53). The discussion below concentrates on how each component of a DBMS is affected by the addition of temporal support.

The only change in the *catalog* of a temporal DBMS is the addition of transaction-time relations. It is possible that as time proceeds, the schema of the database changes (by adding new relations, attributes, etc.); this is called *schema versioning* (54,55). Note that such schema changes are related to transaction time only (i.e., schema evolution does not involve valid time). The catalog should also include statistical and other information about the data stored in the database, which can be useful for query optimization. Examples include the lifespan over which a relation is defined and temporal data granularities and distributions.

At the lower DBMS level (storage manager), performance is affected by the way temporal data is actually stored on disk. The most common approach is *tuple timestamping,* where each tuple (data record) is augmented by two atomic temporal attributes per time dimension supported. For example, in a valid-time database each record would be augmented with the beginning and the end of the record's validity interval. Then the page layout of a temporal relation does not differ from the layout of a conventional relation. (The page layout becomes more complex if the *attribute timestamping* is used; then each record attribute can be associated with its own temporal attributes. For simplicity we consider only tuple timestamping in our discussion.)

Various file structures have been proposed for more efficient accesses to temporal data. One approach termed *temporal partitioning* (56) keeps the current data in a separate store than past data. Current data is assumed to be queried more often, so by separating it from past data, the size of the search structure is decreased and queries for current data become faster. New records are inserted in the *current* store. When a record is updated, its version existing in the current store is moved to the *past store* and a new (updated) version replaces it in the current store. All previous versions of a given record are linked together in reverse chronological order. This is called *reverse chaining* and was introduced in Ref. 48 and further developed in Ref. 56. Reverse chaining can be further improved by the introduction of *accession lists* (57). An accession list clusters together all version numbers (timestamps) of a given record. Each timestamp is associated with a pointer to the accompanying record version which is stored in the past store; thus finding a past version of a given record becomes faster. At worst, versions of a given record could be assigned to different disk pages. Performance can be further improved if such versions are *clustered* or *stacked* together and clusters are linked using *cellular chaining.* These approaches are compared analytically in Ref. 57.

Clearly, there is not a single file structure for a temporal database that universally dominates all the rest. The suitability of a structure depends on the data and the most frequent access pattern (queries) on it. For example, the reverse chaining is a good approach for querying the complete history (past versions) of a given record. In a different approach (58), the evolution of a record is viewed as a (time) sequence of its values. Many such sequences can be grouped together as they share the time dimension. This creates a two-dimensional array whose value at point (r, t) is the value of record r at time t. Schemes that map data points from this two-dimensional array to data pages on the disk are presented in Ref. 59.

For many temporal applications (especially the ones that support transaction time), the data accumulated on disk tends to increase with time. To make space available for new data, the file organization should support means by which "old" temporal data could be easily moved to another medium (tape or optical disk) or even physically deleted from disk. This process is called *vacuuming,* which uses two basic approaches: (a) With the manual approach, a process is manually invoked and will vacuum all records that are "old" when the process is invoked; this vacuuming process can be invoked at any time. (b) With the automated approach, such "old" records are automatically migrated to the optical disk during a data update (52,60,61).

The issues related to the temporal transaction manager deal with timestamping, concurrency, and recovery. Since in databases that support transaction time temporal data is timestamped by the transaction that enters it in the database, the first question is whether to stamp the data at the beginning of the transaction or at the end of the transaction. Note that a major characteristic of transaction-time databases is that past data can be read but is never changed. Because read-only access does not create any concurrency problems, we have to concentrate only on the data that can be updated (i.e., the most current data). This implies that a traditional concurrency control scheme can be used for temporal databases, too. Among many alternatives, experience from conventional databases has shown that two-phase locking (2PL) is more robust and efficient than timestamping-ordering protocols. Using 2PL for concurrency implies that the above question on when to timestamp has to be answered in the context of 2PL. As timestamps are used for updates only (when the data is inserted/updated) they should correctly reflect the serialization order imposed on transactions by 2PL. As a result, the choice of the timestamp to accompany the data records updated by a transaction is delayed until the transaction is being committed (62). Since the transaction time is not known until after commit, it is, of course, impossible to post the transaction time with the data at the time of the update. One approach is to initially assign some identifier to the data a transaction updates and subsequently replace this identifier with the transaction timestamp. This may require some data that has been read earlier by the transaction to be read again in order to assign the timestamp.

Because of its large amounts, temporal data is usually indexed by a temporal index. Concurrency control for temporal indexing has been addressed in Ref. 63.

Conventional DBMS take periodic backups to guard the system against disk failures. The backup reflects the database at a past time. If a media failure occurs, the backup together with the transaction log are used to bring the database into a consistent state. An interesting application of transaction-time databases is that historical data (past states) can be used to support recovery.We refer to Ref. 62 for a detailed description of this idea.

Transaction support can be incorporated in a layered temporal DBMS, where a commercial relational DBMS has been extended to include temporal support (64). This approach exploits the transaction features of the underlying commercial DBMS to perform operations on the temporal relations.

The query manager is probably the component that is most affected from adding temporal support to a DBMS. Any new constructs of the temporal query language have to be incorporated so that the query manager can perform the lexical and syntactic analysis of temporal queries. Furthermore, the query manager should be extended to support temporal query optimization. This is a hard problem for which approaches from conventional databases usually fail (51). For example, traditional query managers focus on optimizing equality predicates, because these are the most frequent query predicates. In temporal queries though, the most frequent predicates involve time intervals (which are inequality-based predicates), making traditional techniques very inefficient.

There are two characteristics of temporal queries that the query manager can benefit from. First, in most temporal applications time is assumed to be always increasing. This property must be utilized during optimization for efficient data clustering, ordering, and so on. It has led to very efficient temporal indices (access methods) (52). Second, temporal queries usually deal with much larger relations (for example, transaction-time or bitemporal relations grow in size as time proceeds), which means that unoptimized temporal queries take much longer to run. As a result the optimizer can spend more time searching for an efficient temporal query plan. (In contrast, the processing time spent for traditional query optimization is usually limited, which implies that various query plans are not considered during optimization). Because of its importance to efficient temporal database implementation, we examine temporal query optimization in more detail.

Temporal Query Optimization

A given temporal query written in a high-level temporal query language, is transformed by the temporal query manager into an equivalent temporal algebra expression (i.e., some initial query plan). The temporal query manager must then enumerate alternative equivalent query plans. This implies the use of equivalences among the temporal algebra operators. It is usually the case that the temporal algebra introduces new operators for which equivalences have to be derived. Such equivalences are known for most of the proposed temporal query languages. Examples of various temporal algebras and their equivalences are presented in Refs. 65–68. Reference 18 presents desirable algebraic equivalences that a temporal algebra should have and compares proposed relational algebras on their ability to support such equivalences.

In assessing the cost of a query plan the temporal query manager needs cost estimates for executing each individual temporal operator that appears in the plan. Such estimation is based on determining the cardinality of the base relation(s) involved in the operator, the cardinality of the result, whether the operator is implemented using an index, and so on. Cost models for the temporal operators involved in TQuel (17) and estimates for the size of the results of various temporal join operators are presented in Refs. 69 and 70, respectively.

The first study on temporal query optimization was performed in Ref. 71, where the performance of a brute-force approach to add time support on a conventional relational DBMS was analyzed. The relational DBMS was minimally extended to support a basic temporal query language (TQuel). Transaction-time, valid-time, and bitemporal relations were implemented. A collection of temporal queries were run using the optimization techniques of the relational DBMS. The results were discouraging, because traditional approaches like sequential scanning, hashing, or indexing suffered a lot due to the ever growing characteristic of temporal data. [For example, if we consider transaction-time data, past data has to be retained and queried as well as current data. If a traditional B-tree is used as an index, values that appeared at different times will be placed under the same B-tree leaf (i.e., the time period where each value appeared is not directly indexed).] This work however emphasized the importance of efficient query optimization for temporal databases.

Since then there has been substantial work on the subject. Several ways to optimize temporal query blocks (including plan generation and selection) are available (71–75); we discuss these general approaches in this section. A large amount of research has concentrated on implementing individual temporal operators (76–82) and on inventing temporal indices (52). An index (or *access method*) is an additional data structure that enables various selection-based queries to run faster. We examine these categories in separate subsections.

Reference 72 presents a general framework for query optimization in a transaction-time database. This framework integrates conventional query processing techniques with techniques from *differential computation* of queries. Differential computation allows queries to be computed incrementally from cached and indexed results of previously computed queries. An internal algebra for transaction-time relations is presented and enlarged with differential operators (in order to take advantage of previously cached results when implementing a query). A new formalism (the *state transition network STN*) is used to enumerate the set of equivalent query plans to the original query. A dynamic programming approach is used to generate and select query plans. Pruning rules are introduced to reduce potentially large STNs by cutting away parts of STNs that contain inferior query plans.

Based on the temporal algebra of Ref. 68, Ref. 73 presents an algorithm that converts a given query to another equivalent expression, which would execute more efficiently. The conversion uses a collection of equivalencies and various heuristics. In particular the algorithm attempts to perform selections and projections as early as possible and tries to reduce the size of cross-products, and a special *restructuring* operator of the temporal algebra.

Instead of introducing new temporal algebra operators that would then need new optimization techniques, Ref. 74 takes a different approach. The type system of an existing object-oriented DBMS is extended to include temporal functions and constraints. A temporal query is then written in the system's object-oriented query language and translated using the system's existing object-oriented algebra. Traditional optimization techniques of the object-oriented DBMS can then be used to optimize temporal queries, too. However, as the physical representation of temporal data and temporal indices are different than for other data, Ref. 74 agrees that new algorithms still need to be developed for evaluating the existing algebraic operators but when they are applied to temporal data.

A framework for optimizing *sequence* queries is presented in Ref. 75. This is a view of temporal data as *time series* (instead of temporal relations). An example of a time series is the ordered sequence of prices for a given stock over a year. This is a positional view of temporal data. For every time instant (say a day) there is a record value (the stock price for that day). In contrast, a temporal relation is a record-oriented view of temporal data as it associates time elements to records. A number of sequence operators have been introduced that are useful for expressing sequence queries (75). Critical in optimizing such queries is the notion of *operator scope* (basically how many positions are important for evaluating a given operator). Using operator scopes, an algorithm is then presented for optimizing general sequence queries. Because sequences are ordered, the optimization process takes advantage of this order.

Optimizing Individual Temporal Operators. Because data in a relational database is organized in relations, the only way to combine selected data from more than one relations is by the *join* operator. Joins are probably the most important relational operators. A straightforward way to implement a join operator is to first perform a Cartesian product among the relations involved in the join. This is clearly inefficient. Due to the importance and frequency of join-related queries, a large amount of research has been performed for efficient join implementations (83). Joins are also very important in temporal databases. Temporal joins are more difficult to implement than traditional joins, because the join condition may include a predicate on the record timestamps. Among the proposed approaches, those in Refs. 76 and 78 have generally been extensions to nested-loop or sort-merge joins, whereas those in Ref. 75 use a partition-based join approach and temporal indexed joins are considered in Ref. 80.

An analysis of the characteristics and processing requirements for the *time-intersection equijoin* (76) and the *event join* (or *entity join*) (77) have been presented. The time-intersection equijoin is the temporal equivalent of the standard equijoin: two tuples from the joining relations are joined if their join attribute values are equal and their time intervals intersect. If no attribute values take part on the join predicate (i.e., if only the time interval intersection is used to join two tuples), we have a time-intersection join. An event join groups several temporal attributes of an entity into a single relation. This operation is useful because temporal attributes belonging to the same entity may be stored in separate relations and be assigned their own time intervals. Based on the physical organization of the relations that take part in a temporal join (i.e., whether the relations are timestamp-ordered or not) several algorithms are presented for processing such joins efficiently. If a relation is timestamp-ordered and of an append-only nature, a special index called the *append tree* is used to facilitate event joins. There are several optimization techniques for temporal joins involving three or more relations (multi-way joins) (77).

The approach of Ref. 78 takes advantage of the special characteristics of time-evolving data and introduces the notion of *stream processing.* A *stream* is an ordered sequence of data objects. Because temporal data is often ordered by time, treating temporal relations as ordered sequences of tuples (i.e., streams of tuples) suggests that stream processing can be effective for temporal queries, too. Reference 78 shows how to use stream processing techniques to efficiently optimize various temporal joins like the *contain join,* the *contain semijoin,* the *intersect join* and many more. The contain join outputs the concatenation of two tuples x, y if the time interval of x contains the time interval of y. The contain semijoin selects those tuples x whose time interval contains that of any tuple y. The intersect join is similar to the time-intersection join of Ref. 77. Under stream processing, each relation is treated as a stream and a processor joins the two relations by combining their streams (much like a conventional merge join). The processor is also allowed to maintain local information about the state of each joined relation as the streams are processed. More complex queries involving intervals were also addressed.

An algorithm to efficiently evaluate time-intersection equijoins (also termed *valid-time natural joins*) is based on tuple partitioning (79). Tuples with similar valid time intervals are first clustered together, and the corresponding partitions of the input relations are then joined together. The efficiency of a partition-based join depends on how well the partitions are created (ideally each partition should have approximately an equal number of tuples per relation). An obvious way is to sort the intervals of the two relations; however, this may be expensive. A better solution is to choose partitioning intervals that with high probability are closed to the optimal ones. An efficient method for approximate partitioning is based on random sampling (79).

In an index-based time-intersection-join implementation (80), the index (called *TP-index*) maps valid time intervals to points in a two-dimensional space and partitions this space into subspaces. If both relations to be joined have been indexed by the TP-index, a partition in one relation needs to be partitioned with a predetermined set of partitions from the other relation.

A temporal query may also involve *aggregate* operators like `avg`, `min`, `max`, `sum`, and `count`. Traditionally aggregates are computed using a sorting, hashing, or indexing approach (35). Such approaches are not efficient if the temporal query involves temporal grouping (i.e., grouping the results by time). Efficient techniques for computing temporal aggregates appear in Ref. 81.

Another temporal operator that appears in temporal queries is the *coalescing* operator introduced earlier (17). Coalescing is similar to duplicate elimination in conventional databases and is needed before other operators like aggregation or selection. Operators with a similar effect to coalescing are included in various temporal algebras, like the COMPRESS of Ref. 43, the Coalesce of Ref. 26, and the FOLD operator of Ref. 84. Efficient implementations for the coalescing operator are presented in Ref. 82.

Temporal Indexing. Any index used to organize time-evolving data is characterized by the following costs: *space* (the space consumed by the index in order to keep such data), *update time* (the time needed to update the method's data structures for data changes), and *query time* (the time needed to compute a temporal query). All three costs are functions of three basic parameters: the query answer size s, the total number of changes (updates) n in the time evolution of the database and the page (block) size b. The answer size s is the number of objects satisfying the query predicate. A change is the addition, deletion, or modification of a record. We say that an index is the I/O optimal solution for a given query if it minimizes the number of I/Os needed to answer the query while using *linear* $[O(n/b)]$ space.

Given the usually large size that temporal data attains, finding efficient temporal indices is important. A variety of temporal indices have been proposed in recent years. The worst-case performance of such indices has been compared (52). Most approaches directly support a single time axis; the majority of these indices assume that time is always increasing and/or updates are always applied on the latest state (i.e., the past is not changed). These are characteristics of transaction time. Assuming a transaction-time database, a common query is the *pure-snapshot* query. For example: "find *all* employees recorded as working on January 1, 1990." More general is the *range-snapshot* query, where the predicate adds a condition on the objects' attribute space: "find all employees recorded as working on January 1, 1990 with salary between 30K and 45K."

Various methods have been proposed to solve the *pure-snapshot* query (78,85–88). Among them, the Snapshot Index (88) provides the I/O optimal solution for this query: it uses $O(n/b)$ space, $O(1)$ processing per update, and $O(\log_b n + s/b)$ I/Os for answering a query. Here s is the number of all employees that were working in the company on January 1, 1990.

Methods that optimize the *range-snapshot* query include (89–94). The I/O optimal solution for this query is provided by the Multi-Version B-tree (89) and the Multiversion Access method (94). Both use $O(n/b)$ space, logarithmic update processing per change and $O(\log_b n + s/b)$ query time. Using the employee example, s is now the number of employees that were working on January 1, 1990 and in addition had salaries in the requested range (i.e., since we discuss a range-snapshot query, s is not the total number of employees working on January 1, 1990). The Time-Split B-tree (92) is another efficient solution.

Indexing the records in valid-time databases can be performed using a multidimensional dynamic index like the R-tree (95). Note that in valid-time databases updates do not happen in order as in transaction-time databases. R-trees will work well for most practical cases; however they do not provide I/O optimal solutions as the transaction-time indices discussed above. Note that an R-tree has the advantage of being a multidimensional index; however, due to overlapping among the areas covered by its nodes, an R-tree cannot guarantee that a single path will be followed to answer a given query.

Bitemporal indexing is addressed in Refs. 96 and 97. A bitemporal database can be visualized as a sequence of states (indexed by transaction time) where each state contains the valid time intervals known to the database at that transaction time. Reference 96 flashes various such states on disk and logs the changes between them. Each state is individually indexed. The effectiveness of this approach depends on how often states are flashed to disk; however, this implies increased storage. The approaches proposed in Ref. 97 all use space linear to the number of updates in the bitemporal evolution. The first approach visualizes each bitemporal object as having two intervals, one for transaction time and one for valid time, and stores it in a multidimensional structure like the R-tree. Although this approach has the advantage of using a single index to support both time dimensions, the characteristics of transaction time create an overlapping problem that affects the query performance of the R-tree. To avoid overlapping, the use of two R-trees (2-R approach) has also been proposed. The third and most efficient approach uses the *bitemporal R-tree* which is an R-tree that keeps its past states as it evolves over transaction time. The advantage of the bitemporal R-tree is that queries to any past state are efficient while the space remains linear.

In addition to indexing, another popular method to efficiently address *membership* queries is external hashing (35). Given a dynamic set S of objects, the traditional membership query asks whether an object with identity k in the most current S. However, if set S is time-evolving, the problem becomes: "find whether object with identity k was in the set S at time t." An efficient solution to the *temporal-hashing* query has been presented (98).

Individual works have used ad hoc approaches to refer to subsets of temporal queries, sometimes employing conflicting or counterintuitive terminology. In the discussion above, we used terms like pure-snapshot, range-snapshot, or temporal-hashing queries but many more terms have been utilized! To avoid considerable confusion when referring to temporal queries a notation is introduced in Ref. 99. The notation uses one entry for the explicit (nontemporal) attribute, and one entry per temporal dimension, as in: *Expl_attr / Valid / Transaction.* An entry can take a value from the set: $\{V, R, S, *, -\}$ where each value has a different meaning. For example, V stands for a single attribute value or time instant, R for a range of attribute values or a time interval, and '–' means that the entry is not applicable. Then the (transaction-time) range-snapshot query is represented as $R/-/V$, because the query specifies a range of values for the explicit attribute (i.e., salary range between 30K and 45K) and a transaction time (January 1, 1990), and no valid time is specified. The notation is easily extensible to cover spatiotemporal queries as well (99).

Most transaction-time indices take advantage of the time-ordered changes to achieve good update performance. Faster updating may be achieved if updates are buffered and then applied to the index in bulks of work. The Log-Structured History Data Access Method (LHAM) (100) and the bulk loading of Ref. 101 are two methods designed to support high update rates.

Temporal query performance can be improved if parallelism is used. This is possible if historical data is spread across a number of disks that can be accessed in parallel. This idea was explored in Ref. 102, where a way to efficiently decluster the Time-Split B-Tree (92) is presented, and in Ref. 103, which examines declustering of the Time Index (85).

Most of the research on temporal indexing assumes a *linear* transaction-time evolution (51). This implies that a new database state is created by updating only the current database state. Another option is the so-called *branching* transaction time, by which evolutions can be created from any past database state. Such branched evolutions form a tree of evolutions that resembles the version trees found in versioning environments. A version tree is formed as new versions emanate from any previous version (assume that no version merging is allowed). There is, however, a distinct difference that makes branched evolutions a more difficult problem. In a version tree, every new version is uniquely identified by a successive version number that can be used to directly access it (91). In contrast, branched evolutions use timestamps. These timestamps will enable queries about the evolution on a given branch. However, timestamps are not unique. The same time instant can exist as a timestamp in many branches in a tree of evolutions simply because many updates could have been recorded at that time in various branches. We are aware of only two works that address problems related to indexing branching transaction time (namely Refs. 104 and 105). In the first, version identifiers are replaced by (branch identifier, timestamp) pairs (104). Both a tree access method and a forest access method are proposed for these branched versions. In the second, data structures are provided for (a) locating the relative position of a timestamp on the evolution of a given branch and (b) locating the same timestamp among sibling branches (105).

Although the above approaches deal with temporal databases, recently various research has addressed the problem of indexing time series (106–108). Given a time series (an evolution) and a pattern, the typical sequence query asks for all those times that a similar pattern appeared in the time series. The search involves some distance criterion that qualifies when a pattern is similar to the given pattern. The distance criterion guarantees no false dismissals (false alarms are eliminated afterward). Whole-pattern matching (106) and submatching (107) queries have been examined. Searching that allows sequence transformations like moving average and time warping is examined in Ref. 108. Such time-series queries are reciprocal in nature to the temporal queries (which usually provide a time instant and ask for the pattern at that time).

TEMPORAL ISSUES IN REAL-TIME AND ACTIVE DATABASES

The notion of time appears also in *real-time* database systems but in a different sense than in temporal databases. For a comprehensive introduction to real-time databases, the reader is referred to Refs. 51 and 109; in particular Ref. 51 surveys real-time databases as related to many temporal database issues. Generally speaking, real-time databases are databases where transactions have time constraints (deadlines) that must be met. This is a characteristic for applications that require timely access or processing of data. Such examples appear in navigation systems (airplane automatic pilots), dialed number services (“800 directory” look-ups), automated factory management (where timely object recognition and appropriate response is needed), and so on. Note that the term real-time does not necessarily mean fast; rather it denotes the need to finish a task before some explicit time constraint.

Typically, a real-time system consists of a *controlled environment* (like a factory floor) and a *controlling* system (usually a computer and its interfaces that enable controlling the operations in the factory floor). The controlling system interacts with its controlled environment through sensors that measure parameters of the environment (for example, temperature sensors or cameras). The sensed data is stored in a real-time database and is further processed to derive new data and possibly set some of the environment parameters through specialized controllers. Timely monitoring and processing of the sensed data is thus necessary. Depending on the application, the timing constraint may apply to one or more database operations like querying (as the “800 directory” look-up), processing insertions, deletions, or updates (as in airplane databases), or enforcing data integrity.

Past research in real-time databases has not explicitly distinguished between valid- and transaction-time dimensions. However the sensors observe the real world environment, which clearly corresponds to valid time. Since transactions are used to record the sensed data, access it from the real-time database, or set parameters through the specialized controllers, time in these settings corresponds to transaction time. The deadlines in database transactions are sometimes specified with respect to a given valid time. Such time constraints basically relate valid with transaction time in that the transaction commit time must be before the specified valid time (51).

The basic problem in real-time databases is how to guarantee that transaction time constraints are satisfied. A transaction can be distinguished by the effect of missing its deadline; usually this is done by assigning a value to each transaction. A *hard deadline* transaction is one that may result in a catastrophe if the deadline is missed (usually applies to safety-critical activities). A large negative value is assigned to such a transaction. A *soft deadline* transaction has some positive value even after its deadline. Typically this value drops to zero at a certain point past its deadline. For example, a transaction may have components that did not meet their individual (soft) deadlines but the overall transaction could still meet its deadline. A *firm deadline* transaction can be viewed as a special case of a soft deadline transaction where the value drops to zero at the transaction’s deadline. For example, a transaction that has to recognize an object while it passes in front of the camera has a firm deadline as it must finish before the object goes outside the camera’s view.

Transaction scheduling must take into account the above transaction characteristics. A key issue in real-time transaction processing is *predictability* (109). We would like to predict beforehand whether a transaction will meet its deadline. This prediction is possible only if we know the worst-case execution of the transaction. However in a database system there are various sources of unpredictability; among others, the central processing unit and input–output usage, transaction aborts, transaction arrival patterns (periodic, sporadic), data conflicts, and the dependence of the transaction execution sequence on the data items accessed. For hard deadline transactions, complete knowledge of the worst-case execution is needed. For soft deadline transactions various priority assignment policies are used for conflict resolution. Examples include the earliest-deadline-first, highest-value-first, and longest-executed-transaction-first. Transaction processing is more complex in the case of distributed real-time databases (110,111). Sometimes, it is acceptable to produce a partial result before the deadline rather than the complete result after the deadline. Timeliness can be achieved by trading off completeness, accuracy, consistency, or currency (109).

Active database systems is another area with a notion of time. Again, we only present the basics so as to discuss the issues related to time. For a detailed coverage of active databases we refer to Refs. 112–114. Such databases are used for applications that need to continuously monitor changes in the database state and initiate actions based on these changes. The basic building blocks in an active database are the event-condition-action (ECA) rules. An ECA rule has the form:

on *event*
if *condition*
then *action*

When the event occurs, the rule is *triggered.* Once the rule is triggered the condition is checked. If the condition holds the action is executed. The above paradigm provides a good mechanism by which database systems can perform a number of useful tasks in a uniform way. Such tasks are enforcing integrity constraints, monitoring data access, maintaining derived data, enforcing protection schemes, and so on.

The event can be arbitrary, including external events (events detected outside the scope of the database, but the rule is processed by the DBMS), database events (such as the begin or commit of a transaction that inserts, deletes, or modifies data), and, for the interests of this article, temporal events. Typically temporal events are triggered at particular absolute times or relative to some time interval. Periodic events are also possible. POSTGRES supports specific temporal events like time and date (115). Among various active database system prototypes, HiPAC (116) allows for the most complex triggering events, including support for temporal events. Triggers on temporal aggregate events are examined in Refs. 117 and 118.

Another notion of time emerges in the specification of conditions and actions that may refer to old or new database states with respect to the triggering event (also known as ECA binding) (113). Usually, there is a mechanism by which conditions in rules triggered by data modifications can refer to the modified data (new database state) or to data preceding the triggering modification (old database state). Similarly actions can refer to the data whose modification caused the rule to trigger. Temporal conditions that can refer to past database states (from the history of evolution of states) have also been proposed (117).

Temporal issues arise in the rule execution semantics, too. For many applications that require timely response to critical events, it may be important to evaluate the condition immediately after an event has occurred, and to execute the action immediately after the condition is evaluated. HiPAC provides for *coupling* modes between event-condition and condition-action that specify when the condition is checked with respect to the triggering event and when the action is executed with respect to its condition. Each coupling mode is either *immediate* (indicating immediate execution as above), *deferred* (indicating execution at the end of the current transaction), or *decoupled* (execute in a separate transaction). Not all combinations of coupling modes are allowed (see Ref. 116 for details).

Recently, the integration of real-time and active databases has been proposed for applications that combine both the timely monitoring of events (for example emergency events) and the provision for timely response. The functionality of such databases is discussed in Ref. 119.

CONCLUSIONS

Time is an important aspect of many real-world applications but is not efficiently supported by conventional databases. Temporal databases have been proposed instead. The field of temporal databases has seen an enormous growth in research: a number of prototypes have already being built (53); a variety of data models and languages have been proposed (51); temporal query processing and indexing have been studied extensively (52). This article provides a quick overview of the field; the interested reader should follow the many references provided for more detailed coverage of particular subjects. The field of temporal databases remains a very active research area. Recently, commercial database vendors have started including temporal support to their products, and this trend is expected to increase.

BIBLIOGRAPHY

1. R. T. Snodgrass and I. Ahn, Temporal databases, *IEEE Comput.,* **19** (9): 35–42, 1986.
2. C. S. Jensen and C. E. Dyreson, The consensus glossary of temporal database concepts, in O. Etzion, S. Jajodia, and S. Sripada (eds.), *Temporal Databases: Research and Practice,* Berlin: Springer-Verlag, 1998, pp. 367–405.
3. A. Tansel et al., *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993.
4. J. Clifford and A. Tuzhilin (eds.), *Recent Advances in Temporal Databases,* Proc. Int. Workshop Temporal Databases, Zurich, 1995, Springer, Workshops in Computing Series.
5. R. T. Snodgrass (ed.), *The TSQL2 Temporal Query Language,* Boston: Kluwer, 1995.
6. O. Etzion, S. Jajodia, and S. Sripada (eds.), *Temporal Databases: Research and Practice,* Berlin: Springer-Verlag, 1998.
7. V. J. Tsotras and A. Kumar, Temporal database bibliography update, *ACM SIGMOD Rec.,* **25** (1): 41–51, 1996.
8. K. K. Al-Tara, R. T. Snodgrass, and M. D. Soo, A bibliography on spatio-temporal databases, *Int. J. Geogr. Inf. Syst.,* **8**: 95–103, 1994.
9. D. M. Gabbay, I. Hodkinson, and M. Reynolds, *Temporal Logic: Mathematical Foundations and Computational Aspects,* Vol. 1. Oxford, U.K.: Oxford Science, 1994.
10. C. E. Dyreson and R. T. Snodgrass, Supporting valid-time indeterminacy, *ACM Trans. Database Syst.,* **23** (1): 1998.
11. J. Clifford and A. Rao, A simple, general structure for temporal domains, *Proc. IFIP TC 8/WG 8.1 Working Conf. Temporal Aspects Inf. Syst.,* Sophia-Antipolis, France, 1987, pp. 17–28.
12. C. Bettini, X. Wang, and S. Jajodia, A general framework for time granularity and its application to temporal reasoning, *Ann. Math. Artif. Intell.,* **22** (1–2): 29–58, 1998.
13. C. Bettini et al., A glossary of time granularity concepts, in O.Etzion, S. Jajodia, and S. Sripada (eds.), *Temporal Databases: Research and Practice,* Berlin: Springer-Verlag, 1998, pp. 406–413.
14. J. Clifford, A. Crocker, and A. Tuzhilin, On completeness of historical relational query languages, *ACM Trans. Database Syst.,* **19** (1): 64–116, 1994.
15. R. T. Snodgrass et al., *Adding Valid Time to SQL/Temporal,* Proposal. MAD-146, Madrid. International Organization for Standardization, 1997.
16. C. S. Jensen, M. Soo, and R. T. Snodgrass, Unifying temporal data models via a conceptual model, *Inf. Syst.,* **19** (7): 513–547, 1994.
17. R. T. Snodgrass, The temporal query language TQuel, *ACM Trans. Database Syst.,* **12** (2): 247–298, 1987.
18. E. McKenzie and R. T. Snodgrass, Evaluation of relational algebras incorporating the time dimension in databases, *ACM Comput. Surv.,* **23** (4): 501–543, 1991.
19. X. Wang, Algebraic query languages on temporal databases with multiple time granularities, *Proc. Conf. Inf. Knowl. Manage.,* 1995, pp. 304–311.
20. C. S. Jensen and R. T. Snodgrass, Temporal specialization and generalization, *IEEE Trans. Knowl. Data Eng.,* **6** (6): 954–974, 1994.
21. X. Wang, S. Jajodia, and V. S. Subrahmanian, Temporal modules: An approach toward federated temporal databases, *Inf. Sci.,* **82**: 103–128, 1995.
22. S. Abiteboul, L. Herr, and J. Van den Bussche, Temporal versus first-order logic to query temporal databases, *Proc. 15th ACM Symp. Prin. Database Syst.,* 1996, pp. 49–57.
23. D. Toman and D. Niwinski, First-order queries over temporal databases inexpressible in temporal logic, *Proc. Int. Conf. Extend. Database Technol.,* 1996, pp. 307–324.
24. S. Gadia and S. Nair, Temporal databases: A prelude to parametric data, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 28–66.
25. S. B. Navathe and R. Ahmed, Temporal extensions to the relational model and SQL, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 92–109.
26. N. Sarda, HSQL: A historical query language, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 110–140.
27. N. A. Lorentzos and Y. Mitsopoulos, SQL extension for interval data, *IEEE Trans. Knowl. Data Eng.,* **9**: 480–499, 1997.
28. G. Ariav, A temporally oriented data model, *ACM Trans. Database Syst.,* **11** (4): 499–527, 1986.
29. R. T. Snodgrass, An overview of TQuel, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 141–182.
30. D. Toman, Point-based temporal extension of temporal SQL, *Proc. 5th Int. Conf. Deduct. Object-Orient. Databases,* 1997, pp. 103–121.
31. S. Gadia, A Homogeneous relational model and query languages for temporal databases, *ACM Trans. Database Syst.,* **13** (4): 1988, pp. 418–448.
32. R. T. Snodgrass, Temporal object-oriented databases: A critical comparison, in W. Kim (ed.) *Modern Database Systems,* Reading, MA: Addison-Wesley, 1995, pp. 386–408.

33. M. Baudinet, J. Chomicki, and P. Wolper, Temporal deductive databases, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 294–320.

34. P. Kanellakis, G. Kuper, and P. Revesz, Constraint query languages, *Proc. ACM Symp. Princ. Database Syst. (PODS),* 1990, pp. 299–313.

35. R. Ramakrishnan, *Database Management Systems,* 1st ed. New York: McGraw-Hill, 1997.

36. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* 2nd ed. Menlo Park, CA: Benjamin/Cummings, 1994.

37. H. Gregersen and C. S. Jensen, *Temporal Entity-Relationship Models—A Survey,* TimeCenter Tech. Rep. TR-3, 1997. Available http://www.cs.auc.dk/research/DBS/tdb/TimeCenter/publications.html

38. R. Elmasri and G. Wuu, A temporal model and query language for ER databases, *Proc. IEEE Conf. Data Eng.,* 1990, pp. 76–83.

39. V. S. Lai, J. P. Kuilboer, and J. L. Guynes, Temporal databases: Model design and commercialization prospects, *Data Base,* **25** (3): 6–18, 1994.

40. J. F. Roddick and J. D. Patrick, Temporal semantics in information systems—a survey, *Inf. Syst.,* **17** (3): 249–267, 1992.

41. C. I. Theodoulidis, P. Loucopoulos, and B. Wangler, The time dimension in conceptual modeling, *Inf. Syst.,* **16** (3): 273–300, 1991.

42. V. Vianu, Dynamic functional dependencies and database aging, *J. ACM,* **34**: 28–59, 1987.

43. S. B. Navathe and R. Ahmed, A temporal relational model and a query language, *Inf. Sci.,* **49** (2): 147–175, 1989.

44. N. A. Lorentzos, *Management of Intervals and Temporal Data in the Relational Model,* Tech. Rep. 49, Athens, Greece: Agricultural Univ. of Athens, 1991.

45. J. Wijsen, J. Vendenbulcke, and H. Olivie, Functional dependencies generalized for temporal databases that include object-identity, *Proc. Int. Conf. Entity-Relationship Approach,* 1993, pp. 99–109.

46. J. Wijsen, Design of temporal relational databses based on dynamic and temporal functional dependencies, *Proc. Int. Workshop Temporal Databases,* Zurich, 1995, pp. 61–76.

47. X. S. Wang et al., Logical design for temporal databases with multiple granularities, *ACM Trans. Database Syst.,* **22**: 115–170, 1997.

48. J. Ben-Zvi, The time relational model, Ph.D. dissertation, *UCLA,* 1982.

49. A. Segev and A. Shoshani, The representation of a temporal data model in the relational environment, *Proc. Int. Conf. Stat. Sci. Data Manage.,* 1988, pp. 39–61.

50. C. S. Jensen, R. T. Snodgrass, and M. D. Soo, Extending existing dependency theory to temporal databases, *IEEE Trans. Knowl. Data Eng.,* **8**: 563–582, 1996.

51. G. Özsoyoğlu and R. T. Snodgrass, Temporal and real-time databases: A survey, *IEEE Trans. Knowl. Data Eng.,* **7**: 513–532, 1995.

52. B. Salzberg and V. J. Tsotras, A comparison of access methods for temporal data, *ACM Comput. Surv.* (to be published March 1999); TimeCenter Tech. Rep. TR-18, 1997. Available: http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/publications.html

53. M. H. Böhlen, Temporal database system implementations, *ACM SIGMOD Rec.,* **24** (4): 53–60, 1995.

54. E. McKenzie and R. T. Snodgrass, Schema evolution and the relational algebra, *Inf. Syst.,* **15** (2): 207–232, 1990.

55. J. F. Roddick, Schema evolution in database systems—An annotated bibliography, *ACM SIGMOD Rec.,* **21** (4): 35–40, 1992.

56. V. Lum et al., Designing DBMS support for the temporal database, *Proc. ACM SIGMOD Conf.,* 1984, pp. 115–130.

57. I. Ahn and R. T. Snodgrass, Partitioned storage for temporal databases, *Inf. Syst.,* **13** (4): 369–391, 1988.

58. A. Shoshani and K. Kawagoe, Temporal data management, *Proc. 12th Conf. Very Large Data Bases,* 1986, pp. 79–88.

59. D. Rotem and A. Segev, Physical organization of temporal data, *Proc. 3rd IEEE Int. Conf. Data Eng.,* 1987, pp. 547–553.

60. M. Stonebraker, The design of the Postgres storage system, *Proc. 13th Conf. Very Large Databases,* 1987, pp. 289–300.

61. C. Kolovson and M. Stonebraker, Indexing techniques for historical databases, *Proc. 5th IEEE Int. Conf. Data Eng.,* 1989, pp. 127–137.

62. D. Lomet and B. Salzberg, Transaction-time databases, in A. Tansel et al. (eds.), *Temporal Databases; Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 388–417.

63. D. Lomet and B. Salzberg, Concurrency and recovery for index trees, *VLDB J.,* **6**: 224–240, 1997.

64. C. Vassilakis, N. A. Lorentzos, and P. Georgiadis, Transaction support in a temporal DBMS, *Proc. Int. Workshop Temporal Databases,* Zurich, 1995, pp. 255–271.

65. J. Clifford and A. Tansel, On an algebra for historical relational databases: Two views, *Proc. ACM SIGMOD Conf.,* 1985, pp. 247–265.

66. A. Tansel and L. Garnett, Nested historical relations, *Proc. ACM SIGMOD Conf.,* 1989, pp. 284–293.

67. A. Tuzhilin and J. Clifford, A temporal relational algebra as basis for temporal relational completeness, *Proc. VLDB Conf.,* 1990, pp. 13–23.

68. S. K. Gadia and C. S. Yeung, A generalized model for a relational temporal database, *Proc. ACM SIGMOD Conf.,* 1988, pp. 251–259.

69. I. Ahn and R. T. Snodgrass, Performance analysis of temporal queries, *Inf. Sci.,* **49** (1–3): 103–146, 1989.

70. A. Segev et al., Selectivity estimation of temporal data manipulations, *Inf. Sci.,* **74** (1–2): 111–149, 1993.

71. I. Ahn and R. T. Snodgrass, Performance evaluation of a temporal database management system, *Proc. ACM SIGMOD Conf.,* 1986, pp. 96–107.

72. C. S. Jensen and L. Mark, Differential query processing in transaction-time databases, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 457–491.

73. S. S. Nair and S. K. Gadia, Algebraic optimization in a relational model for temporal databases, *Proc. Int. Workshop Infrastruc. Temporal Databases,* Arlington, VA, 1993, pp. 390–397.

74. U. Dayal and G. Wuu, A uniform approach to processing temporal queries, *Proc. VLDB Conf.,* 1992, pp. 407–418.

75. P. Seshadri, M. Livny, and R. Ramakrishnan, Sequence query processing, *Proc. ACM SIGMOD Conf.,* 1994, pp.430–441.

76. A. Segev and H. Gunadhi, Event-join optimization in temporal relational databases, *Proc. 15th Conf. Very Large Data Bases,* 1989, pp. 205–215.

77. A. Segev, Join processing and optimization in temporal relational databases, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 356–387.

78. T. Y. C. Leung and R. R. Muntz, Stream processing: Temporal query processing and optimization, in A. Tansel et al. (eds.), *Temporal Databases: Theory, Design, and Implementation,* Menlo Park, CA: Benjamin/Cummings, 1993, pp. 329–355.

79. M. D. Soo, R. T. Snodgrass, and C. S. Jensen, Efficient evaluation of the valid-time natural join, *Proc. IEEE Conf. Data Eng.,* 1994, pp. 282–292.

80. H. Lu, B. C. Ooi, and K. L. Tan, On spatially partitioned temporal join, *Proc. VLDB Conf.,* 1994, 546–557.

81. N. Kline and R. T. Snodgrass, Computing temporal aggregates, *Proc. IEEE Int. Conf. Data Eng.,* 1995, pp. 222–231.

82. M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, Coalescing in temporal databases, *Proc. VLDB Conf.,* 1996, pp. 180–191.

83. P. Mishra and M. H. Eich, Join processing in relational databases, *ACM Comput. Surv.,* **24** (1): 63–113, 1992.

84. N. A. Lorentzos and R. G. Johnson, Extending relational algebra to manipulate temporal data, *Inf. Syst.,* **13** (3): 289–296, 1988.

85. R. Elmasri, G. Wuu, and Y. Kim, The time index: An access structure for temporal data, *Proc. 16th Conf. Very Large Databases,* 1990, pp. 1–12.

86. H. Gunadhi and A. Segev, Efficient indexing methods for temporal relations, *IEEE Trans. Knowl. Data Eng.,* **5**: 496–509, 1993.

87. V. J. Tsotras, B. Gopinath, and G. W. Hart, Efficient management of time-evolving databases, *IEEE Trans. Knowl. Data Eng.,* **7**: 591–608, 1995.

88. V. J. Tsotras and N. Kangelaris, The snapshot index, an I/O-optimal access method for timeslice queries, *Inf. Syst. Int. J.,* **20**: 237–260, 1995.

89. B. Becker et al., An asymptotically optimal multiversion B-tree, *VLDB J.,* **5**: 264–275, 1996.

90. C. Kolovson and M. Stonebraker, Segment indexes: Dynamic indexing techniques for multi-dimensional interval data, *Proc. ACM SIGMOD Conf.,* 1991, pp. 138–147.

91. S. Lanka and E. Mays, Fully persistent B^+ trees, *Proc. ACM SIGMOD Conf.,* 1991, pp. 426–435.

92. D. Lomet and B. Salzberg, Access methods for multiversion data, *Proc. ACM SIGMOD Conf.,* 1989, pp. 315–324.

93. Y. Manolopoulos and G. Kapetanakis, Overlapping B^+ trees for temporal data, *Proc. 5th Jerusalem Conf. Inf. Technol.,* Jerusalem, 1990, pp. 491–498.

94. P. J. Varman and R. M. Verma, An efficient multiversion access structure, *IEEE Trans. Knowl. Data. Eng.,* **9**: 391–409, 1997.

95. A. Guttman, R-trees: A dynamic index structure for spatial searching, *Proc. ACM SIGMOD Conf.,* 1984, pp. 47–57.

96. M. Nascimento, M. H. Dunham, and R. Elmasri, M-IVTT: A practical index for bitemporal databases, *Proc. Database Expert Syst. Appl. (DEXA) '96,* Zurich, 1996, pp. 779–790.

97. A. Kumar, V. J. Tsotras, and C. Faloutsos, Designing access methods for bitemporal databases, *IEEE Trans. Knowl. Data Eng.,* **10**: 1–21, 1998.

98. G. Kollios and V. J. Tsotras, *Hashing Methods for Temporal Data,* TimeCenter Tech. Rep. TR-24, 1998. Available: http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/publications.html

99. V. J. Tsotras, C. S. Jensen, and R. T. Snodgrass, A notation for spatiotemporal queries, *ACM SIGMOD Rec.,* **27** (1): 47–53, 1998.

100. P. O'Neil and G. Weikum, A log-structured history data access method (LHAM), *High Perform. Trans. Syst. Workshop,* Asilomar Conference Center, Pacific Grove, CA, 1993.

101. J. Van den Bercken, B. Seeger, and P. Widmayer, A generic approach to bulk loading multidimensional index structures, *Proc. VLDB Conf.,* 1997, pp. 406–415.

102. P. Muth, A. Kraiss, and G. Weikum, LoT: A dynamic declustering of TSB-tree nodes for parallel access to temporal data, *Proc. Extending Database Technol.* (*EDBT*) *Conf.,* 1996, pp. 553–572.

103. V. Kouramajian, R. Elmasri, and A. Chaudhry, Declustering techniques for parallelizing temporal access structures, *Proc. 10th IEEE Conf. Data Eng.,* 1994, pp. 232–242.

104. B. Salzberg and D. Lomet, *Branched and Temporal Index Structures,* Tech. Rep. NU-CCS-95-17, Boston: Northeastern Univ., College Computer Science, 1995.

105. G. M. Landau, J. P. Schmidt, and V. J. Tsotras, On historical queries along multiple lines of time evolution, *VLDB J.,* **4**: 703–726, 1995.

106. R. Agrawal, C. Faloutsos, and A. Swami, Efficient similarity search in sequence databases, *Proc. Conf. Foundations Data Organization and Algorithms (FODO) Conf.,* 1993, pp. 69–84.

107. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, Fast subsequence matching in time-series databases, *Proc. ACM SIGMOD Conf.,* 1994, pp. 419–429.

108. D. Rafiei and A. O. Mendelzon, Similarity-based queries for time series data, *Proc. ACM SIGMOD Conf.,* 1997, pp. 13–25.

109. K. Ramamritham, Real-time databases, *Distrib. Parallel Databases,* **1** (2): 199–226, 1993.

110. B. Kao and H. Garcia-Molina, Deadline assignment in a distributed soft real-time system, *Proc. 13th Int. Conf. Distrib. Comput. Syst.,* 1993, pp. 428–437.

111. V. Kanitkar and A. Delis, A case for real-time client-server databases, in A. Bestavros and S. Wolfe (eds.), *Real-Time Database Systems: Issues and Applications,* Boston: Kluwer, 1997, pp. 395–408.

112. S. Chakravarthy, *A Comparative Evaluation of Active Relational Databases,* Tech. Rep. UF-CIS-TR-93-002, Gainesville: Univ. of Florida, 1993.

113. U. Dayal, E. N. Hanson, and J. Widom, Active database systems, in W. Kim (ed.), *Modern Database Systems: The Object Model, Interoperability and Beyond,* Reading, MA: Addison-Wesley, 1995, pp. 434–456.

114. J. Widom and S. Ceri (eds.), *Active Database Systems: Triggers and Rules for Advanced Database Processing,* San Mateo, CA: Morgan Kaufmann, 1996.

115. S. Potamianos and M. Stonebraker, The POSTGRES rules system, in J. Widom and S. Ceri (eds.), *Active Database Systems: Triggers and Rules for Advanced Database Processing,* San Mateo, CA: Morgan Kaufmann, 1996, pp. 43–61.

116. U. Dayal, A. P. Buchmann, and S. Chakravarthy, The HiPAC Project, in J. Widom and S. Ceri (eds.), *Active Database Systems: Triggers and Rules for Advanced Database Processing,* San Mateo, CA: Morgan Kaufmann, 1996, pp. 177–206.

117. P. Sistla and O. Wolfson, Temporal conditions and integrity constraints in active database systems, *Proc. ACM SIGMOD Conf.,* 1995, pp. 269–280.

118. I. Motakis and C. Zaniolo, Temporal aggregation in active database rules, *Proc. ACM SIGMOD Conf.,* 1997, pp. 440–451.

119. K. Ramamritham et al., Integrating temporal, real-time, and active databases, *ACM SIGMOD Rec.,* **25** (1): 8–12, 1996.

VASSILIS J. TSOTRAS
Department of Computer Science and Engineering
University of California

X. SEAN WANG
Information and Software Engineering Department
George Mason University