Time-by-Example Query Language for Historical Databases

ABDULLAH U. TANSEL, M. EROL ARKUN, AND GULTEKIN OZSOYOGLU

Abstract—**Time-by-Example (TBE) is a user-friendly query language designed specifically for historical relational databases. It follows the graphical structure and the example query concept of QBE, and employs the hierarchical arrangement of subqueries of Abe and STBE. Similar to STBE, TBE handles set- and simple-valued attributes. In addition, to handle time, TBE is capable of manipulating triplet- and set-triplet-valued attributes.**

The underlying data model used in TBE is an extended relational data model in which nonfirst normal form relations and attribute time stamping (in contrast to tuple time stamping) are used. A triplet is a 3-tuple whose components are the lower and upper bounds of a time interval and a value valid over the interval. A triplet is used as a time-stamped value of a time-dependent attribute. Set-valued time-dependent attributes are modeled by sets of triplets.

To process TBE queries and to define a historical relational algebra (HRA), standard operators of the relational algebra and the pack/unpack operators of [21] are augmented by triplet-decomposition, triplet-formation, slice, and drop-time operators. Methodologies for translating TBE queries into HRA expressions and for constructing their parse trees are presented.

Index Terms—**Databases, historical databases, query languages, query processing.**

I. Introduction

A relational database is defined as a set of time-varying relations of assorted degrees [9]. However, a relational database models the real world by keeping only its most recent snapshot. As events take place, the real world changes states, and new values are incorporated into the database to form the most recent snapshot. Thus, a relational database as defined by Codd keeps only the current values of the attributes of a tuple. As soon as a new value is assigned to an attribute, its previous value is erased. To retain complete information, the current value of an attribute along with its past values should be stored in the database. There are many applications where the need for complete information, for both the present and the past, exists.

Manuscript received May 29, 1987; revised October 30, 1987. The work of A.U. Tansel was supported in part by the PSC-CUNY Research Award Program of the City University of New York under Grant 667299. The work of G. Ozsoyoglu was supported by the National Science Foundation under Grants DCR-8306616 and DCR-8605554.

A. U. Tansel is with the Department of Statistics and Computer Information Systems, Benard M. Baruch College, City University of New York, New York, NY 10010.

M. E. Arkun is with the Department of Computer Engineering, Bilkent University, Ankara, Turkey.

G. Ozsoyoglu is with the Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH 44106.

IEEE Log Number 8826227.

The attributes of an object (i.e., an entity or a relationship), some of which assume different values over time, form the history of that object. A database that maintains object histories is called a historical database (HDB). A historical relational database (HRDB) utilizes the relational database theory to model object histories. The issue of incorporating time into data models has attracted little attention until recently. Extensions to the relational data model as well as to some existing query languages have been proposed for dealing with historical data. However, we believe that the time dimension should be developed as a coherent theory for both the relational data model and the associated query languages rather than by extension to the existing query languages specifically tailored for the manipulation of current data.

In this paper, we propose a new graphical query language, Time-by-Example (TBE), which has suitable constructs for interacting with HRDB's is a natural way. TBE is a user-friendly query language for historical relational databases. It follows the graphical, two-dimensional approach of such previous languages as Query-by-Example (QBE) [41], Aggregation-by-Example (Abe) [17], and Summary-Table-by-Example (STBE) [23]. TBE also uses the hierarchical window (subquery) concept of Abe and STBE. TBE manipulates triplet-valued (set-triplet-valued) attributes and historical relations. Set-theoretic expressions are also allowed to deal with time intervals.

Early studies have conceptualized time, and identified issues related to it [4], [5], [35], [40]. In recent years, researchers have studied the issue of time for incorporating the temporal dimension into different data models [2], [13], [16], [30]. History carrying relations can be visualized as three-dimensional structures (cubes), the third dimension representing the time. Some form of the tuple time stamping is used to simulate the three-dimensional structure and to convert it into two-dimensional (flat) tables. Clifford and Warren augment each relation with existence and state attributes and use intensional logic to capture the semantics of time [7]. Ben-Zvi and Ariav add implicit time attributes to each relation, and extract static relations for manipulation by traditional relational algebra operations [3], [6]. Snodgrass keeps each time-dependent attribute along with the key as a separate relation and adds implicit time attributes to each relation [33]. Chaining of history tuples to current tuples in reverse time order is proposed by Lum *et al.* [19]. Implementation strategies for this approach are explored in [10]. Tuple time stamp-

ing creates undue data redundancy and complexity in the model. Time fits naturally at the attribute level when non-first normal form relations are allowed.

There are three recent studies which consider time stamping at the attribute level. Clifford views attribute values as functions from time points to simple-valued domains, and gives examples for the operations of a possible historical relational algebra [8]. Gadia [11] and Tansel [36], [38] both view attribute values as functions from time intervals to the value domains. Gadia defines a temporal domain which is a finite union of intervals, and extracts a snapshot before applying the traditional algebra and calculus operations. On the other hand, Tansel first normalizes the historical relations and then applies algebra and calculus operations [38]. Different attribute types are also allowed to coexist in the same relation.

TBE is a query language based on the relational calculus. The underlying data model used in TBE incorporates time into the relational model in a natural way. The model employs attribute time stamping, which is a better abstraction of reality than tuple stamping and allows a three-dimensional view of historical relations at the user level. Time-dependent and time-independent attributes can coexist in the same relation. Furthermore, there is a well-defined historical relational algebra which allows optimization of query processing in TBE, which is not the case in other languages. Algebra of [3], [6] is restrictive, and the model of TQUEL [33] does not have an algebraic language. Moreover, TQUEL uses tuple substitution in query processing which does not lend itself to optimization easily.

Our extension to the relational model allows one-level nesting, i.e., sets of atomic (i.e., time-independent values and triplets (i.e., time-dependent values). We think that one-level nesting of attributes is simple, and powerful enough to model a large number of time-related data processing applications. Allowing arbitrary levels of nesting (for both time-dependent and time-independent attributes) gives way to a more generalized model which can handle time variations of complex objects. This extension is beyond the scope of this paper.

The main contribution of this work is to demonstrate that a powerful and user-friendly graphical query language can be designed for historical databases within the framework of relational database theory. This paper combines the results of our previous works [22], [26], [36] in the context of historical databases, as well as modifying and refining them. We introduce the idea of a graphical historical query language. TBE follows the architecture of STBE [22], and additionally includes triplet and set triplet-valued attributes and the associated temporal constructs and operations for them. Translating TBE queries into historical relational algebra (HRA) is based on the query flattening methods developed in [26]. These methods are modified to handle semantics of temporal data as well as the new operations of HRA.

Section II is a brief overview of the historical relational model. Section III gives historical relational algebra operations. Time-by-Example is described in Section IV along with illustrative examples. The complete syntax of TBE is defined in the Appendix. Section V covers the translation of TBE queries into equivalent historical relational algebra expressions. Section VI discusses the expressive power of TBE, and Section VII compares the TBE to QUEL-based temporal languages. Section VIII concludes the paper.

II. The Historical Relational Model

A database is a set of relation instances. A relation instance (or a relation) is a table with each column labeled by a distinct attribute. Each attribute has an associated domain of values. Each row (tuple) of a relation is unique (i.e., a relation is a set). A relation scheme is a set of attributes. $\mathrm{Atr}(R)$ denotes the attribute set of relation R. Let U be the set of values regarded as elementary (such as reals, integers, etc.) and the value null ("$-$"). An attribute A is *simple-valued* (atomic) if $D_M \subseteq U$ where D_M is the domain of the attribute A. If $D_M \subseteq P(U)$ (where P is the power set), then A is a *set-valued attribute*.

Let T be a set of totally ordered discrete time points relative to an origin zero, that is, $T = \{0, 1, 2, \cdots, now\}$ where *now* denotes the present time. $[l, u)$ is an interval which includes the time points t such that $l \leq t < u$. l is the lower bound of the interval, and u is its upper bound. The interval $[l, u)$ is closed at the lower bound and open at the upper bound. $< [l, u), v >$ where $[l, u)$ is a time interval and $v \in U$ is a triplet denoting that the value v is valid over the time interval $[l, u)$. Any interval which includes *now* as its upper bound is an expanding interval. As time advances, the interval $[l, now]$ also expands. Unlike the other intervals, $[l, now]$ is a closed interval. Two triplets are equivalent if and only if their corresponding components are the same.

t_b is the time when an object o (corresponding to an entity or a relationship) is introduced to the database. All of o's attributes have t_b as the initial time stamp. Each of o's attributes has either a non-null value or the null value, or a combination of these, in the interval $[t_b, now]$. The null value indicates that the attribute value is unknown. The time reference of relationships can similarly be captured by 1) using the time stamps of time-varying attributes of a relationship relation and/or 2) by adding a set-valued attribute which includes the life (validity period) of the relationship in the relationship relation [38].

Let TU be a set of triplets defined over T and U. An attribute is *triplet-valued* if it receives its values from a domain which is a subset of TU. An attribute is *set-triplet-valued* if its domain D_M is a subset of $P(TU)$. The values of set-triplet-valued attributes are sets of triplets. Moreover, the time intervals of the triplets in a set are disjoint. Fig. 1 shows a relation EMP with the scheme EMP, (E#, ENAME, *$SALARY, *$MANAGER, *FOREIGN-LANGUAGES) where E# and ENAME are simple-valued attributes, *FOREIGN-LANGUAGES is set-valued, and *$MANAGER and *$SALARY are set-triplet-valued attributes. As a notion, set-valued, triplet-valued, and set-triplet-valued attributes

E#	ENAME	*SSALARY	*SMANAGER	*FOREIGN-LANGUAGES
133	TOM	{<15,25,14K>, <25,now,18K>}	{<15,19,RON>, <19,22,GARY>, <22,now,AL>}	{FRENCH, SPANISH}
140	ANN	{<10,now,12K>}	{<10,now,MARY>}	{ '-' }

Fig. 1. An example historical relation.

are prefixed with "*," "$," and "$," respectively. For the triplet-valued attribute $\$A$, $\$A_l$, $\$A_u$, and $\$A_v$ denote its lower bound, upper bound, and the value components, respectively. A_T is used to denote the interval components of $\$A$. Similarly, $\$x_l$, $\$x_u$, and $\$x_v$ represent lower bound, upper bound, and the value components, respectively, for the triplet $\$x$, and $\$x_T$ is the interval component of $\$x$. It is convenient to view a set of triplets $*\$x$ as a union of three partitions that correspond to lower bounds, upper bounds, and data values. Hence, $*\$x_v$ is the set which contains the value components of the triplets in $*\$x$. $*\$x_l$, $*\$x_u$, and $*\$x_T$ are defined analogously. A formal definition of historical relational databases is given in [38].

We now define instances of a relation, say R, over the interval $[l, u)$ and at the time point t. Let $r_{[l,u)}$ and r_t denote these instances, respectively. Let s be a tuple of r, $s[A]_{[l,u)}$ is the tuple component which corresponds to attribute A. If A is an atomic attribute, $s[A]_{[l,u)}$ is the same as $s[A]$. Similarly, $s[*A]_{[l,u)}$ is the same as $s[*A]$ because these attribute types are independent of time. On the other hand, for the triplet-valued attribute $\$A$, $s[\$A]_{[l,u)}$ is equal to a triplet $<[l', u'), a>$ where $[l', u') = [l, u)$ **inter** $[m, n)$, **inter** denotes set intersection and $[m, n)$ is the time interval of $s[\$A]$. When the intersection of intervals is null, $s[\$A]_{[l,u)}$ is also null and does not participate in the result. For the set-triplet-valued attribute $*\$A$,

$$s[*\$A]_{[l,u)} = \left\{ <[l', u'), x_v> \,\middle|\, \$x \in s[*\$A] \textbf{ and } [l', u') = ([l, u) \textbf{ inter } [x_l, x_u)) \right\}$$

where **inter** denotes intersection of two time intervals. The tuple $s_{[l,u)}$ is then defined as

$$s_{[l,u)} = (s[A_1]_{[l,u)} \, o \, s[A_2]_{[l,u)} o \cdots o \, s[A_n]_{[l,u)})$$

where $A_i \in \mathrm{Atr}(R)$, $1 \le i \le \deg(R)$, and o denotes concatenation. Thus,

$$r_{[l,u)} = \{ s_{[l,u)} \mid s \in r \}.$$

The time point t can be visualized as the interval $[t, t+1)$. So, r_t is equivalent to $r_{[t,t+1)}$.

III. Historical Relational Algebra Operators

This section summarizes the algebra of time-carrying (historical) relations [36], [38]. The basic HRA operators are project, Cartesian product, set union, set difference, selection, aggregate formation, pack, unpack, triplet decomposition, and triplet formation. There are also two useful operators, slice and drop time, that are expressible in terms of the basic set of operators. The drop-time operation discards the time component of a triplet-/set-triplet-valued attribute and creates a simple-/set-valued attribute. Project (Π) and Cartesian product ($\times$) directly apply to historical relations. For the union ($\cup$) and the set difference ($-$), the types of corresponding attributes must be compatible, and for the value-equivalent tuples, overlapping time intervals should be coalesced. Selection (σ) and join ($\bowtie$) can be extended to historical relations with minor modifications.

Let t_1 and t_2 be two tuples having components for a set of attributes X. Then $t_1[X] = t_2[X]$ denotes $t_1[X_i] = t_2[X_i]$ for all $X_i \in X$. D denotes the set of relations in the database.

Project (Π_X): Let $R \in D$, $X \subseteq \mathrm{Atr}(R)$ and $|X| = k$. Then

$$\Pi_X(R) = \{ t[X] \mid t \in R \}.$$

The X component of each tuple of R is retained and other attributes are discarded.[1]

Selection (σ): Let $R \in D$, and let F be a formula involving atoms $x\theta y$, $x\theta v$ and connectives **and**, **or**, and **not**. x and y are tuple components, and v is a constant. θ is either a simple comparison operator (i.e., $\theta \in \{<, >, =, \ge, \le, \ne\}$), or a set operator (i.e., $\theta \in \{\subset, \subseteq, \supset, \supseteq, =\}$), or the set membership operator (i.e., $\theta \in \{\in, \notin\}$). θ must be consistent with the type of operands. Then

$$\sigma_F(R) = \{ t \mid t \in R \textbf{ and } F \text{ holds for } t \}.$$

Pack (P): Let $R \in D$, $|\mathrm{Atr}(R)| = n$, $A \in \mathrm{Atr}(R)$, and $C_A = \mathrm{Atr}(R) - \{A\}$. For each $(n-1)$ tuple $g \in \Pi_{C_A}(R)$, we define an n tuple W_g:

$$W_g[C_A] = g$$

$$W_g[A] = \begin{cases} \{ t[A] \mid t \in R \textbf{ and } t[C_A] = g \} \\ \quad \text{if } A \text{ is simple valued or triplet valued} \\ \{ x \mid (\exists t)(t \in R \textbf{ and } t[C_A] = g \textbf{ and } \\ \quad x \in t[A]) \} \\ \quad \text{otherwise;} \end{cases}$$

then

$$P_A(R) = \left\{ W_g \mid g \in \prod_{C_A} (R) \right\}.$$

The pack operator $P_A(R)$ maps (packs) sets of tuples in R, whose $(n-1)$ components for C_A are the same, into a single tuple. With the exception of triplet- and set-triplet-valued attributes, this operation is identical to the pack operation of [21].

Unpack (UN): Let $R \in D$, $A \in \mathrm{Atr}(R)$, $C_A = \mathrm{Atr}(R) - \{A\}$. For each tuple $t \in R$, we define a set of tuples

[1]Projecting out the time-dependent attributes of a historical relation causes the loss of the time dimension. This problem can be solved by storing the time over which a tuple is valid in an implicit time attribute, which can also indicate the time tuple is deleted (see [8], [38]).

$$UN_A(\{t\}) = \begin{cases} \{t\} & \text{if } A \text{ is simple valued or triplet valued} \\ \{t' \mid t'[A] \in t[A] \textbf{ and } t'[C_A] = t[C_A]\} & \\ & \text{otherwise;} \end{cases}$$

then

$$UN_A(R) = \bigcup_{t \in R} (UN_A(\{t\})).$$

If A is simple valued, then $UN_A(R) = R$; otherwise $UN_A(R)$ maps each tuple t in R into a set of tuples such that each element in $t[A]$ becomes the A value of one resulting tuple. With the exception of triplet- and set-triplet-valued attributes, this operation is identical to the unpack operation of [21].

Aggregate Formation [18]: Let $R \in D$, $X \subseteq \text{Atr}(R)$, $|X| = k$. Let f be an aggregate function, and A, $A \in \text{Atr}(R)$ be simple valued. Then $R<X, f_A>$ is a relation with degree $k + 1$, and is defined as

$$R<X, f_A> = \{t[X] \circ y \mid t \in R \textbf{ and } y = f_A(\{t' \mid t' \in R \textbf{ and } t'[X] = t[X]\})\}$$

where "$\circ$" denotes concatenation.

The aggregate formation operator first *partitions* tuples of relation R such that tuples having the same X component are in the same partition. Then the function f is applied to the A component of tuples in each partition, and the X value and the associated aggregate value are output for each partition.

Triplet Decomposition: Triplet-decomposition operation breaks a triplet-valued attribute into its three components. It adds two new attributes to the relation for representing the time interval: one for the lower bound, one for the upper bound. The value component replaces the original triplet-valued attribute.

Let $R \in D$, $C_A = \text{Atr}(R) - \{\$A\}$, and $\$A \in \text{Atr}(R)$. Triplet decomposition creates a new relation whose degree is $\deg(R) + 2$ and is defined as

$$T - DEC_{\$A}(R) = \{t \circ v \circ l \circ u \mid (\exists t')(t' \in R \textbf{ and } t[C_A] = t'[C_A] \textbf{ and } v = t'[A_t] \textbf{ and } l = t'[A_l] \textbf{ and } u = t'[A_u])\}$$

where "$\circ$" denotes concatenation.

Triplet Formation: Let $R \in D$, A, L, $U \in \text{Atr}(R)$, and $C_A = \text{Atr}(R) - \{A, L, U\}$. Let R' be the result relation whose degree is $\deg(R) - 2$, obtained by projecting out attributes A, L, and U from R and appending a triplet-valued attribute $\$A$ such that $C_{\$A} = \text{Atr}(R') - \{\$A\} = C_A$. Then

$$T - FORM_{A,L,U}(R) = \{t \mid (\exists t')(t' \in R \textbf{ and } t[C_A] = t'[C_A] \textbf{ and } t[A_t] = t'[A] \textbf{ and } t[A_l] = t'[L] \textbf{ and } t[A_u] = t'[U])\}.$$

Triplet-formation operation creates a triplet-valued attribute $\$A$ from the three attributes A, L, and U which correspond to the data value, lower bound, and upper bound components of triplets, respectively.

Slice: Slice operation aligns the interval components of a triplet-valued attribute according to the interval components of another triplet-valued attribute. Let $R \in D$ and $\$A$, $\$B \in \text{Atr}(R)$. Slice operation is defined as

$$SLICE_{\$A,\$B}(R) = \{t \mid (\exists t')(t' \in R \textbf{ and } t[C_{\$A}] = t'[C_{\$A}] \textbf{ and } t[A_t] = t'[A_t] \textbf{ and } t[A_l] \geq t'[B_l] \textbf{ and } t[A_u] \leq t'[B_u] \textbf{ and } t[A_l] \geq t'[A_l] \textbf{ and } t[A_u] \leq t'[A_u] \textbf{ and } (t[A_l] = t'[A_l] \textbf{ or } t[A_l] = t'[B_l]) \textbf{ and } (t[A_u] = t'[A_u] \textbf{ or } t[A_u] = t'[B_u]))\}.$$

The interval of the first attribute $\$A$ is sliced according to the interval of the second attribute $\$B$. Let t be a tuple of R. Intersection of the time interval components of $t[\$A]$ and $t[\$B]$ is assigned to the time interval component of the new triplet for $\$A$. The new triplet receives its data value from $\$A$. If the two time intervals do not overlap, the tuple is discarded.

IV. TBE Queries

This section describes TBE, and gives example queries. A complete, BNF-like syntax of TBE is given in the Appendix. We use the following database in the examples throughout the paper.

DIVISION (dvname, *$manager, location)
DEPARTMENT (dname, division, *$manager, *$budget)
EMPLOYEE (ename, *$salary, *$department).

An example TBE query is shown in Fig. 2. The query returns Tom's salary history.

A. Query Specification

In TBE, the user specifies a query by placing example elements on the empty relation skeletons provided on the screen. The query is arranged as a hierarchy of windows (subqueries). A window consists of a name, an output, one or more relation skeletons, an optional condition box, and an optional range box. The user communicates with a screen manager using the cursor and function keys on the keyboard to build empty constructs on the screen and to open windows. The subquery at the topmost level is called the ROOT. A TBE query consists of one ROOT and its descendent subqueries. Many TBE queries can be expressed in one window (i.e., the ROOT), whereas quer-

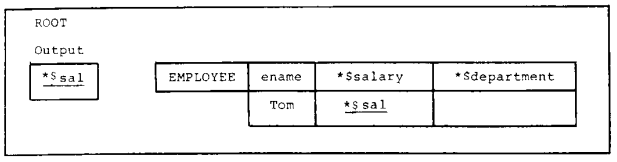


Fig. 2. An example TBE query: salary history of Tom.

ies involving set difference and aggregate operations usually require additional windows (subqueries).

1) Relation Skeleton and Variables: A relation skeleton represents a relation scheme. It has columns for each attribute of the relation and a header row with the relation name and the attribute names. It also has several empty rows for the specification of the example tuples of a query. The tuples are formed according to elementary items. Elementary items are constants (numbers, strings, triplets, or a set of numbers, strings, or triplets) and free or fixed variables. Variables can be simple valued, triplet valued, set valued, or set triplet valued. As a notation, free variables are underlined, and fixed variables are doubly underlined. Triplet-valued variables are prefixed with "$," set-valued variables are prefixed with "*," and set-triplet-valued variables are prefixed with "*$."

A free variable $\underline{x}$ is any arbitrary identifier which appears in a subquery. A variable $\underline{\underline{x}}$ is a fixed variable if it appears in a window W and $\underline{x}$ appears in another window which references W. The value of a fixed variable is bound to the value of the corresponding free variable. Fixed variables are similar to procedure parameters in a programming language.

A triplet-valued variable has three components, which correspond to the lower and upper bounds of the time interval, and the attribute value which is valid over this interval. Let $\$\underline{x}$ be a triplet-valued variable $\$\underline{x}.l$, $\$\underline{x}.u$, and $\$\underline{x}.v$ denote the lower bound of the time interval, the upper bound of the time interval, and the value component, respectively. $\$\underline{x}.T$ is an abbreviation representing the time interval of $\$\underline{x}$; it makes interval specification in TBE queries easier. A triplet-valued variable or any of its components may be used as part of example elements. Triplet-valued example elements can also be specified explicitly by citing each component. For instance, $< x, y, z >$ defines a triplet, the lower bound, the upper bound, and the value of which are x, y, and z, respectively. x, y, and z may be constants or variables. In conformance with the conventions of other high-level languages, a missing component is indicated by a comma. Similarly, a triplet definition may take its time or value component from another triplet-valued variable, e.g., $< \$\underline{x}.T, y >$.

Example elements may appear in relation skeletons as $\theta\ x$ where θ is a comparison operator consistent with the type of x; x can be a variable or a constant; and $\theta\ x$ is a shorthand version of an expression which can be specified in the condition box. For instance, the example element >3 is a relation skeleton indicates that a free variable, say $\underline{x}$, appears in this column, and the expression $\underline{x} > 3$ appears in the condition box.

2) Output Box: The output of the ROOT or a subquery is a relation skeleton without the header row. The output skeleton specifies the format and the contents of the query's result. Each box in the output skeleton can contain an elementary item, an aggregation operation, an arithmetic expression with arithmetic operators ($+$, $-$, $*$, $/$) and elementary items, or a subquery name (the output of which is a single-column relation). The aggregation operation specifies the name of a window along with the number of column on which the aggregation operation is to be applied.

3) Condition Box: The condition box is a skeleton with one large column and several rows. Each row contains a Boolean expression which consists of terms connected with the logical connectives *not*, *and*, *or*. Each time in the form of $t_1 \theta t_2$ where t_1 and t_2 can be elementary items, aggregation operations applied to window names, subquery names with single output columns, or intervals combined with set operations (i.e., **intersection**, **union**, and **difference**). Depending on the types of t_1 and t_2, the operator θ can be a comparison operator (such as $<$, $>$, $=$), a set comparison operator (such as $\subset$, $\supset$, $\supseteq$), or the set membership operator ($\in$). Conditions at separate rows of condition box are AND'ed.

For user convenience, we introduce temporal operators as syntactic variations of set operations. These binary infix operators work on time intervals and are similar to the ones in [33]. They are *Overlap* (for set intersection), *Extend* (for set union), and *Without* (for set difference). Comparison operators are *Contains*, *Does-not-contain* (for set inclusion), *Is-in*, *Is-not-in* (for set membership), *Is* and *Is-not* (for set equality). Predicates *Intersects* and *Precedes* are introduced to state whether two intervals intersect or one precedes the other.[2] Note that the former operators return intervals, whereas the latter ones return a Boolean truth value as a result. *Overlap* returns a single interval as the result. On the other hand, *Extend* and *Without* may return two separate intervals because intervals are not closed under these operations.

4) Range Box: The range box is a skeleton with two boxes and several rows, and is used to dynamically define relations over which variables range. The first box contains variables separated by commas, and enclosed within parentheses. The second box contains OR'red window or relation names, and defines the union of relations over which the variables in the first box range. Another function of the range box is on the spot definition of single column relations over which simple- or set-valued variables range. This function of the range box is useful when values of aggregates are calculated at different time points. A set of values are listed following the designation *Actual*. A sequence of integers can be specified with the first and last values and dots in between. Fig. 3 contains a TBE query illustrating the use of the range box and the condition box. The query returns, for each year from 1975 to 1985, the division Tom worked for.

[2]Others can also be added, e.g., *Succeeds*, *Before*, *After*, etc.

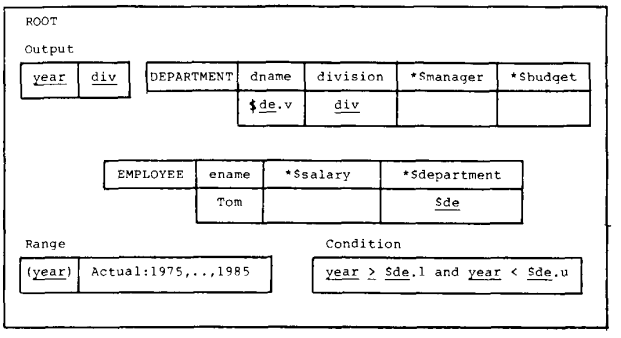


Fig. 3. An example TBE query: the divisions Tom worked for.

B. Query Interpretation

Example elements are used to match and retrieve data in the database, and to bind relations in different windows. The answer to a query is a relation. Each tuple of the output relation for a window is produced by

1) matching constants and free variables appearing in the window with the corresponding tuple components of relation instances in the database.

2) binding and matching fixed variables in the window.

3) satisfying conditions specified in the condition box, and

4) retrieving the output items from the database.

The following rules are used in the instantiation of different example elements.

1) A constant matches only to itself.

2) A free variable in a relation skeleton matches to any value in that column of the relation instance provided that both the variable and the attribute are simple valued or set valued. If a simple-valued variable appears in the column of a set-valued attribute, the variable matches to any member of any set value of that attribute. If a variable occurs at several columns in a window, all occurrences match to the same value. This is one way of relating data in different columns or relations. Similar matching rules apply to triplet-valued and set-triplet-valued variables. A triplet-valued variable matches to any one of the triplets in the column it appears. A set-triplet-valued variable matches to any one of the sets in the corresponding column. If a triplet-valued variable is specified in a set-triplet-valued column, it matches to any one of the triplets in the set of triplets which are the values of the attribute. When a variable and an attribute (column) in which the variable appears as an example element have the same type (i.e., simple valued, set valued, triplet valued, or set triplet valued), then we say that the variable is *compatible* with the attribute; otherwise, it is *incompatible*. The matching rules for compatible/incompatible variables are given in Table I.

A fixed variable $\underline{\underline{x}}$ in window A is *bound* to the free variable $\underline{x}$ in window B where B references A. $\underline{\underline{x}}$ matches either the current value of the binding variable $\underline{x}$ or any element of the set which is the current value of $*\underline{x}$. $*\underline{\underline{x}}$ matches the current value of the binding variable $*\underline{x}$. Similarly, $\$\underline{\underline{x}}$ is bound to $\$\underline{x}$ (or $*\$\underline{x}$), and matches a triplet which is the same as the value of $\$\underline{x}$ or one of the triplets in $*\$\underline{x}$. Variable $*\$\underline{\underline{x}}$ matches a set of triplets in the database which is also the current value of $*\$\underline{x}$. The components of a triplet-valued variable, say $\$\underline{\underline{x}}$, can also be used as a fixed variable, e.g., $\underline{\underline{x}}.T$, $\underline{\underline{x}}.l$, $\underline{\underline{x}}.u$, or $\underline{\underline{x}}.v$. A fixed variable $\$\underline{\underline{x}}$ ($\underline{\underline{x}}$) is inconsistently defined when it is bound to $*\$\underline{x}$ ($*\underline{x}$). Fig. 4 explains the matching and binding rules for free and fixed variables.

The empty column of a relation skeleton matches to any value in the corresponding column of the relation instance. Hence, the row of a relation skeleton matches a row of the relation instance when a successful match occurs for its nonempty columns. Data in different skeletons can be related by using the same variable in appropriate columns. Data among subqueries are related through the correspondance of free and fixed variables. Subqueries are invoked either by the aggregage functions specified in the output or condition box of a subquery or by the subquery names appearing in the output box of another subquery. When a query Q is invoked, it is evaluated by using the matching and binding rules, and by satisfying the conditions. Then, aggregation functions (if any) are applied to the result of Q. If Q is invoked by citing its name in the output box of another subquery, the result of Q is placed as a set-valued (set-triplet-valued) attribute value in that box. In this case, however, Q should have a single-column output. Such subqueries are used in forming relations with set-valued or set-triplet-valued attributes.

After all the conditions are satisfied, the result of the query is produced by extracting the variables and constants specified in the output box from the matching tuples in the relevant relation instances. An arithmetic expression, specified in the output box, is evaluated and the result is assigned to the corresponding column. The same process of matching tuples, evaluating the conditions and the aggregates in an arbitrary order, is repeated until no new output tuples can be produced.

C. Example Queries

In this section, we present some TBE example queries.

Example 1: What were the divisions Tom worked for when he earned $20K? The TBE query is given in Fig. 5.

The predicate in the condition box ensures that the intersection of the times of salary and department triplets is not empty. There may be several division triplets if Tom worked for more than one department when his salary was $20K.

Example 2: What are the names of employees who worked for the Toy, Shoe, and Fur departments, and have made $50K or more since leaving these departments? The TBE query is given in Fig. 6.

This example illustrates the use of example elements as time and value components of triplets. The range box defines a single-column relation with three tuples. Note that $< \underline{sl}, \underline{su}, \underline{s} >$ can be specified as $< \underline{sl}, , \underline{s} >$ to avoid the unused variable $\underline{su}$.

Example 3: What are the names of employees who

TABLE I
MATCHING RULES FOR COMPATIBLE/INCOMPATIBLE VARIABLES

Matching	Simple Variable $\underline{x}$	Set-Valued Variable $\underline{*x}$	Triplet-Valued Variable $\underline{\$x}$	Set Triplet-Valued Variable $\underline{*\$x}$
Simple-Valued Attribute A	$\underline{x}$ can take any value in column A	not allowed	$\underline{\$x}.l$, $\underline{\$x}.u$, $\underline{\$x}.v$, or $\underline{\$x}.T$ can be used and match any value in column A	not allowed
Set-Valued Attribute $*A$	$\underline{x}$ can match any member of the set which is any value in column $*A$	$\underline{*x}$ matches any value in column $*A$ of the relation	$\underline{\$x}.l$, $\underline{\$x}.u$, or $\underline{\$x}.v$ can be used and match any member of the set which is any value in column $*A$	not allowed
Triplet-Valued Attribute $\$A$	$\underline{x}$ can be used as a component of explicit triplet definition and matches to any triplet having it as a component	not allowed	$\underline{\$x}$ can take any value (triplet) in the column $\$A$ of the relation	not allowed
Set Triplet-Valued Attribute $*\$A$	$\underline{x}$ matches to any triplet, having it as a component, in the set which is a value of $*\$A$	not allowed	$\underline{\$x}$ can take any triplet which is a member of the set value in columns $*\$A$	$\underline{*\$x}$ can take any value (set) in column $*\$A$

MATCHING & BINDING	x in B or *B	*x in *B	$\underline{\underline{\$x}}$ in $B or *$B	$\underline{\underline{*\$x}}$ in *$x
x in A or *A	x matches to the current value of x	n. a.	$\underline{\underline{\$x}}$ matches to a triplet whose specified component is the same as x	n. a.
*x in *A	x matches to any member of the current value of *x	$\underline{\underline{*x}}$ matches to the current value of *x	$\underline{\underline{\$x}}$ matches to a triplet whose specified component is a member of the current value of *x	n. a.
$x in $A or *$A	x takes its value from the specified component of the triplet which is the current value of $x	n. a.	$\underline{\underline{\$x}}$ takes the current value of $x	n. a.
*$x in *$A	x takes its value from the specified component of the triplet which is a member of of the current value of *$x	n. a.	$\underline{\underline{\$x}}$ matches to any member of the current value of *$x	$\underline{\underline{*\$x}}$ takes the current value of *$x

(i)

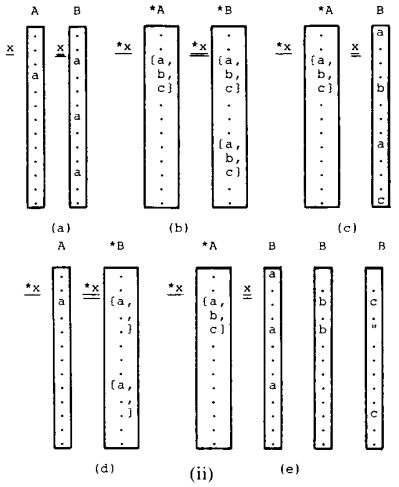


Fig. 4. Binding rules for fixed variables. (i) Binding rules for fixed variables. (ii) Illustrations.

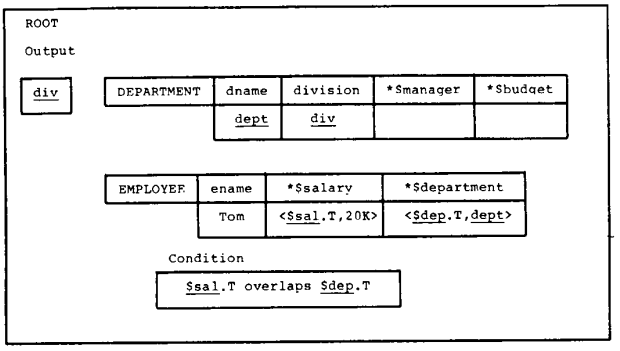


Fig. 5. TBE query of Example 1.

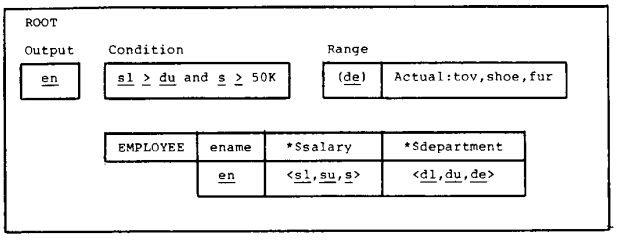


Fig. 6. TBE query of Example 2.

make (made) more than their managers? The TBE query is given in Fig. 7.

This query illustrates the utilization of time intervals. The first part of the expression in the condition box ensures that an employee's salary, the department he works in, the department's manager, and the manager's salary have intersecting time intervals. In other words, the employee makes a salary ($\underline{\$sal}.v$) during the time he has a manager (***man***) who makes less than $\underline{\$sal}.v$.

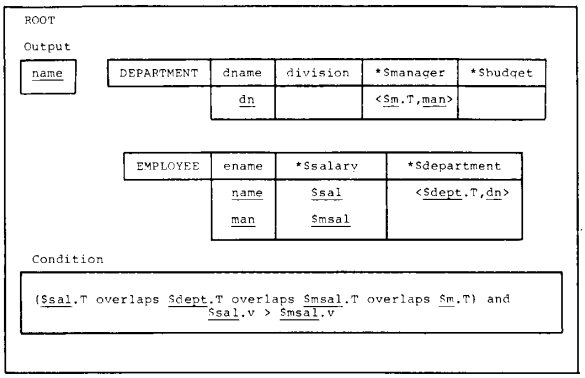


Fig. 7. TBE query of Example 3.

Example 4: When and who have worked in the Toy department, and made more than \$30K? The TBE query is given in Fig. 8.

This query illustrates the formation of a triplet-valued attribute in the output box. *Overlap* creates a single interval as the result. On the other hand, *Extend* and *Without* may create one or two intervals depending on the operand intervals. So several triplets, one or more, can be formed when these operators are specified in the output box as triplet components. Naturally, the resulting attribute becomes set triplet valued. A set formator subquery, having these operators in the output box, also returns a set of triplets (but not a set of sets of triplets; please see Example 6).

Example 5: Get the list of present employees in each department which is ever managed by Tom. Output the employee names as a set along with the department name. The TBE query is given in Fig. 9.

This example illustrates the formation of a set-valued attribute. Subquery *A* returns the names of employees currently working in the department ***dn***. Note that ***dn*** in *A* is bound to ***dn*** in the ROOT. The predicate in the condition box of *A* specifies that the upper bound of triplet's interval is *now*. Hence, this is a current triplet.

Example 6: For each department, get the names of employees and the times they worked in that department when the department's buget was greater than \$100K, and the employee's salary was \$20K. The TBE query is given in Fig. 10.

This query illustrates the creation of set-triplet-valued attributes. For each department, the subquery EMP returns the names, along with the time, of the employees who worked for that department.

Example 7: Find the number of employees in the Toy department for each year from 1975 to 1985. The TBE query is given in Fig. 11.

This query illustrates the aggregate formation operation. The range box declares a single-column relation whose tuples are 1975, 1976, $\cdots$, 1985. ***year*** matches any one of them. The subquery *A* returns the names of employees who were in the Toy department during ***year***. COUNT(*A*) returns the count of *A*. Note that for this query, we assume that the database is modeled by a time unit of years.

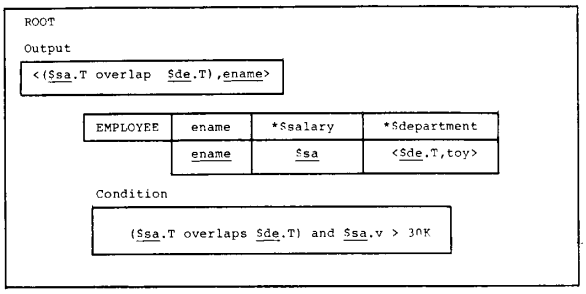


Fig. 8. TBE query of Example 4.

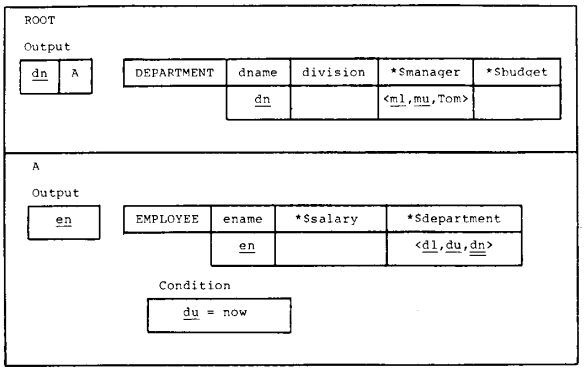


Fig. 9. TBE query of Example 5.

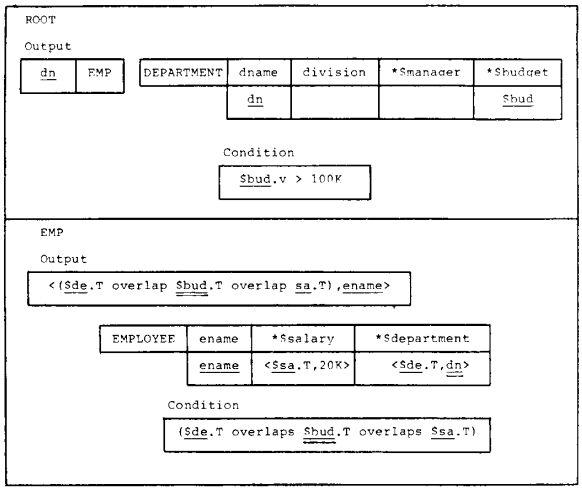


Fig. 10. TBE query of Example 6.

Example 8: Find the names of employees who worked in the Toy department only during the tenure of Tom as a department/division manager. The TBE query is given in Fig. 12.

This query illustrates the use of set theoretic expressions involving time intervals in the condition box. Note

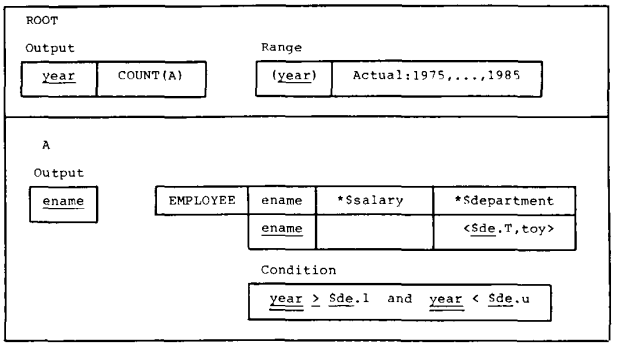


Fig. 11. TBE query of Example 7.

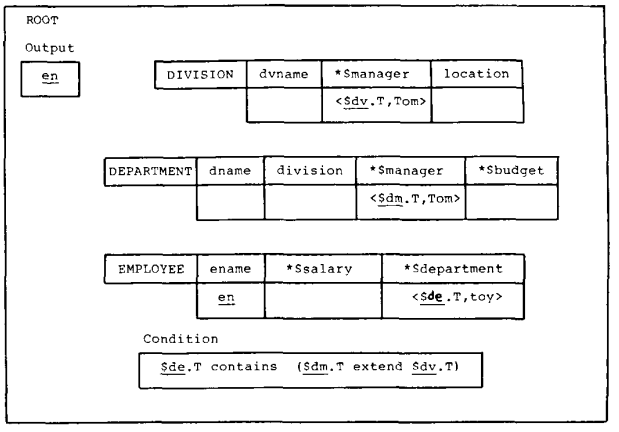


Fig. 12. TBE query of Example 8.

that the subset comparison operation ensures that only those employees whose tenures in the Toy department were (are) completely included within the tenure of Tom as a manager are selected for output.

Example 9: Find the counts of departments in each location. Output the locations and the corresponding counts. The TBE query is given in Fig. 13.

This query illustrates the use of external root queries in the range box. Subquery *B* calculates the counts of departments in each division. The SUM aggregate function in the ROOT computes the sum of these counts in each different location.

Clearly, with four windows, this query looks complicated. There are two reasons for this complexity. First, for uniformity, we always construct a window for any "derived set" over which a simple summary statistics is to be taken. If this requirement were relaxed for simple derived sets used in aggregations (by using an alternative syntax), windows *A* and *C* would be elminated. Second, this query requires two aggregations; and because of the example database design (that location is an attribute of DIVISION relation, not DEPARTMENT relation), these aggregations must be performed in consecutive steps; hence, we have the need for two windows. In many other database languages, this query is expressible as two separate queries leading to explicitly stored intermediate results.

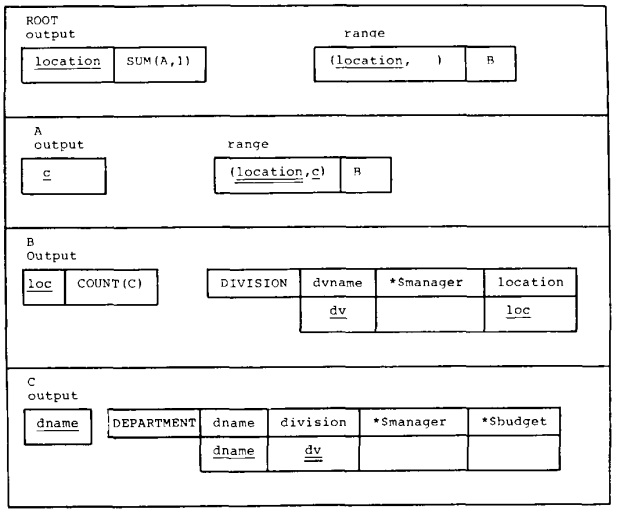


Fig. 13. TBE query of Example 9.

V. Parsing TBE Queries

Translation of TBE queries into equivalent historical relational algebra (HRA) expressions is similar to the translation methods used in STBE [24], [26]. Therefore, we describe very briefly the parsing method developed by Ozsoyoglu *et al.* [26], and discuss the utilization of slice operator for handling time-carrying attributes of TBE. After translation, the resulting parse tree represents an HRA expression which breaks the nested (hierarchical) structure of a TBE query, and hence, allows for the optimization of the query.

Hierarchical structure of TBE queries can be represented by a labeled directed acyclic graph (DAG) where the windows are the vertices of the DAG, and the edges correspond to the invocation of windows. The edge labels of the DAG indicate the specific utilization of the descendent subquery which is either a window name mentioned in an aggregate formation operation in the output or condition box or a subquery name (whose output is a single column) appearing in the output box. The former is called an *aggegate subquery* which performs an aggregation operation f on the result of the subquery. The label f_C is placed on the edge $A \rightarrow B$ when the aggregate function f is applied to the attribute C (if any) in B. The latter is called a *set formation subquery*. Similarly, if C is the column to be obtained from B for set formation, label S_C appears on the edge $A \rightarrow B$. Fig. 14 gives the DAG's for Examples 6 and 7.

A. The Basic Trees

For each window, there is a basic HRA expression which represents the locally processable operations in that window. The parse tree for the basic HRA expression is called the *basic tree*. The leaves of the basic tree are relations or column names, and the nonterminal nodes are HRA operations. The basic HRA expression for a window has the following components.

Fig. 14. The DAG's of TBE queries in Examples 6 and 7.

1) Each relation in the window containing an incompatible (free or fixed) variable is unpacked along the attribute where the variable is defined.

2) All the selections which do not require the evaluation of other subqueries are performed.

3) Joins of relations having the same variable are performed. The resulting relations (if more than one) are combined by a cartesian product operation.

4) Attributes for those triplet-valued variables whose time components appear as arguments in intersection (i.e., Overlap) operations are time sliced by each other. Slicing aligns the time of the attributes. Similarly, attributes of variables whose time components are used as arguments in union and difference operations are aligned by appropriate slicing operations.[3]

5) If the query consists of only the root window and one of the components of a triplet-valued variable is specified in the output box, then a triplet-decomposition operation is applied on the attribute where the triplet-valued variable is specified.

6) If the query consists of only the root window, then for the intervals explicitly specified in the output box, a triplet-formation operation is introduced.

7) All attributes whose column skeletons are used to define free and fixed variables are projected. The basic trees for the example queries 6 and 7 are given in Fig. 15(a) and (b), respectively.

B. Transformation on the Basic Trees

The subqueries are first transformed into a form so that each subquery can be processed without any reference to its ascendant subqueries in the DAG. Then the transformed subqueries are recursively attached starting from those nodes with no outgoing edges towards the ROOT node to construct the parse tree for the query. There are two possible transformations, *f transformations* and *s transformations*. *f* transformations apply to aggregate labels, and *s* transformations apply to set formation labels in the DAG. Assume the subquery Q_1 is mentioned in the subquery Q_0. *f* transformation modifies Q_1, and applies an aggregate operation to the result of Q_1. This operation creates a new root for the basic tree of Q_1. Inconsistently defined variables (i.e., *$\underline{x}$ in column *A of Q_0 and $\underline{\underline{x}}$ in column *B of Q_1) require additional operations. First, the column *A is projected and the Cartesian product of *A and the basic tree of Q_1 is formed. Then, a selection operation is applied to retain the values of *B which are members of the sets in *A. Hence, the basic tree of Q_1 becomes ready for the aggregation operation. If *$\underline{\underline{x}}$ appears in several columns of Q_1, the intersection of the projected columns is used in the Cartesian product operation.

The *s* transformation is applied to a set formation subquery which creates a set of elements taken from the attribute A, *A, A, or *A of a relation. Two additional operations are added to the basic tree of Q_1. First, the relevant attributes are projected. Then, a pack operation is applied to the simple or triplet-valued attribute which appears in the output box of Q_1. Also, the presence of inconsistently defined variables requires the same modifications mentioned above.

If a triplet-valued variable's time component appears as a fixed variable in an intersection expression of Q_1, a slice operation is needed between the corresponding attributes in Q_0 and Q_1. Assume $\underline{x}$ appears in column A of the relation skeleton R in Q_0, and $\underline{\underline{x}}.T$ is specified in an intersection operation along with the time component of the triplet-valued variable used in column B of the relation skeleton S in Q_1. Before applying an *f* or *s* transformation, attribute B should be time sliced by attribute A. For this purpose, the Cartesian product of the basic tree of Q_1 and $\Pi_{\$A}(R)$ is formed, and a slice operation, from $R.\$A$ to $S.\$B$, is applied to the result. Triplet-formation and triplet-decomposition operations are also introduced for the explicitly specified time intervals in the output box after an *f* or *s* transformation. The case where a fixed variable (not involving time) appears in the condition box of Q_1 only is similarly handled. Of course, in this case, the predicate in the condition box is enforced using a selection operation instead of the slice operation. Example of *f* and *s* transformations are given in Fig. 15(a) and (b), respectively.

The next step is the attachment of transformed basic trees recursively starting from the leaves of the DAG. A depth-first-search in the DAG determines the sequence of attachments. Let the subquery Q_1 be an immediate descendant of the subquery Q_0. If Q_1 is mentioned in the output box of Q_0, the basic tree of Q_1 is combined with the basic tree of Q_0 by a join operation. The join condition is the equality of columns in which free (from Q_0) and fixed variables (from Q_1) appear. If the subquery Q_1 appears as part of a predicate in the condition box of Q_0, then, basic trees of subqueries involved in the predicate (including the basic tree for Q_1) are combined by a Cartesian product operation, and a selection operation is applied to the result ensuring the equality of fixed variables with the corresponding free variables and the predicate as well. Parse trees for the example queries 6 and 7 are given in Fig. 15(a) and (b), respectively.[4]

To summarize, the basic tree created for the root TBE query and its transformations is a complete translation to HRA. Set formation can be considered as a special case

[3]For this purpose, the versions of the slice operation implementing the set union and the set difference of time intervals of attributes are analoguously defined [38]. However, these operations may return two noncontiguous time intervals associated with the same value component for a triplet, i.e., the resulting attribute becomes a set-triplet-valued attribute.

[4]Note that the join in Fig. 15(b) should in fact be a "directional join" if the employee counts of zero for some years are to be retained in the output. For details, see [26].

of the aggregation operation where a set of values is returned instead of calculating a single value for them. Aggregation in TBE is implemented in the form of subqueries which utilize the correspondence of free and fixed variables. Thus, the corresponding attributes in the subquery act as an agent of partitioning. However, when fixed variable are inconsistently specified or when a fixed variable in the subquery appears only in the condition box, partitioning cannot be done within the subquery. Because a set of values must define the aggregation set in the subquery, the relevant attribute from the ascendent subquery is projected and introduced to the basic tree of the subquery. Now, partitioning can be done within the subquery without the need for the ascendent subquery. Hence, transformations on a subquery tree complete the subquery's translation to HRA. There are two ways for referencing subqueries, and the attachment of subqueries considers these two possibilities separately.

VI. Expressive Power of TBE

Time-by-Example is a direct extension of a relational calculus, called RC/S*, that replaces the universal quantifiers and the negation of the relational calculus of Codd [39] by the set comparison operators [27]. RC/S* queries are always safe, i.e., they always produce finite set of output tuples in finite time [27]. The conditions that make RC/S* queries safer are built into the syntax of TBE.

Time-by-Example and the historical relational algebra of Section III have the same expressive power. That is, any query expressible in TBE is expressible in HRA and vice versa. From Sections V-A and V-B, we have TBE $\subset$ HRA. Thus, expressing the HRA operators in TBE and the fact that the subquery concept and the range box together effectively serve as a scoping construct (i.e., parentheses) are sufficient to show that HRA $\subset$ TBE. Thus, TBE and HRA have the same expressive power. Below we describe how HRA operators can be expressed in TBE.

Expressing projection and Cartesian product is straightforward. For the selection operation, selection conditions can be placed in the attribute columns of relation skeletons (i.e., θ v) or in the condition box. Union of two relations can be specified in the range box. One can simulate the set difference by the aggregate function count. To calculate $R - S$, it suffices to introduce a subquery which returns a tuple of S if it also exists in R and a condition, specified in the root, to make sure that the count of the output of the subquery is zero. The pack operation is similar to an aggregation operation where the relation is partitioned according to all the remaining attributes except the packed attribute. A subquery returns the packed attribute value, and the root applies a set of formator on this result. The unpack operation is expressed by specifying an inconsistent variable in a set-(set triplet-) valued attribute. Expressing triplet decomposition and triplet formation is simple. Specifying a triplet explicitly in the output box would suffice to create a triplet-valued attribute. For a triplet decomposition, on the other hand, two additional boxes are appended to the output box for the lower and upper bounds of the interval. The value part of the triplet replaces the original attribute. These are the basic operations of HRA and the remaining ones can be expressed in terms of the basic operations.

VII. Comparison to Other Languages

Several temporal query languages, based on extensions to the relational model, have been reported. LEGOL 2.0 is a relational algebra based language developed for querying time-varying legal data [15]. Ben-Zvi [6] adds a time view construct to SQL for extracting a static relation to be operated on by an SQL query. Another extension to SQL proposed by Ariav [3] includes the constructs AT, WHILE, DURING, BEFORE, AFTER, and AS OF. Only one historical relation can be referenced in a query. TSQL [20] augments SQL with constructs such as temporal ordering, temporal group by, time slice, moving window, and when. These clauses add powerful time processing capabilities to the language. These query languages are based on the relational data model where tuple time stamping is used. In a recent work, Segev and Shoshani [28], [29] develop a temporal data model which is independent of any specific data model, define temporal data structure that support sequences of temporal values, and finally describe operators (using a SQL-like syntax) that manipulate temporal structures.

We compare TBE to three other temporal query languages, TQUEL, HTQUEL, and HQUEL in detail. TQUEL [33] is an extension to the query language QUEL of INGRES [34]. TQUEL introduces three new clauses, *valid*, *when*, and *as of*, and allows temporal operators *overlap*, *extend*, and *precede*. Overlap and extend are the same as our operators. Valid clause specifies the time of the result. When clause augments the where clause for temporal qualification criteria, an as of clause allows for rollback to a different database state. Following is the Example 4 in TQUEL. Underlying database involves two relations ES(NAME, SALARY) and ED(NAME, DEPT). Each relation also contains implicit time attributes.

```
Range of s is ES
Range of d is ED
Retrieve (s.name)
    valid from start of (s overlap d) to end of (s overlap
    d)
    where s.name = d.name and s.salary > 30K and
    d.dept = "Toy" and
    when s overlap d
    as of now.
```

As is seen, a join operation is needed between the two relations. TQUEL allows difference of time intervals in append and delete statements. However, a valid clause cannot specify the difference and union of time intervals when they are not contiguous. Unique and cumulative versions of aggregate functions are defined in [33]. Unique aggregates are needed because of the redundancy in the data model. TBE can handle these situations with standard aggregate functions. Optimization of query process-

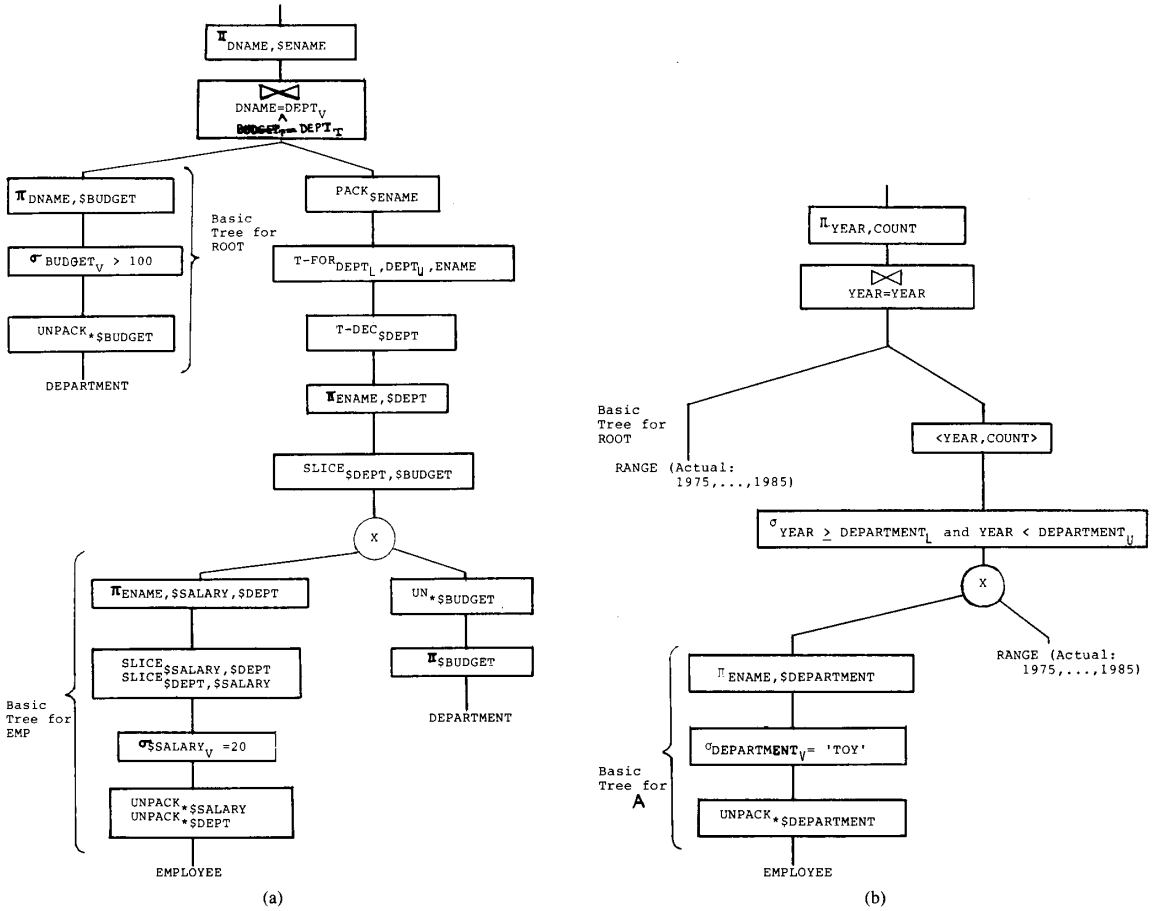


Fig. 15. The parse trees for Examples 6 and 7.

ing in TQUEL is limited because there is no corresponding relational algebra, and QUEL applies tuple substitution which does not lend itself easily to query optimization. This is a common problem with other extensions to QUEL as well. A prototype of TQUEL has been implemented.

HTQUEL is a substantial extension to QUEL [11]. HTQUEL facilitates extraction of time domains which are later used as temporal predicates in the when clause to specify the time reference of a snapshot. Powerful functions to visit time points (intervals) in temporal domains are also included for temporal navigation. HTQUEL has an assignment statement for operations on temporal domains. The underlying data model is similar to TBE's. Following is the HTQUEL expression for Example 4:

```
Range of e is EMPLOYEE
Retrieve into vs (tdome(e.salary))
  where e.salarly > 30K
Retrieve into vd (tdom (e.dept))
  where e.dept = "Toy"
v ← vs ∩ vd
Retrieve (e.name) when v.
```

This query can be written in a different way as well. We chose this one because it illustrates the salient features of the language. HTQUEL handles intersection, union, and difference of time domains elegantly. The time specification of the result is implicit unlike TQUEL and TBE. Aggregate operations are not defined for HTQUEL yet. A relational algebra is defined and completeness of HTQUEL is proved.

HQUEL is another extension to QUEL [37] and is based on the model described in Section II. It allows set formation and set comparison operators for time intervals and references to the components of triplets. $\$A(V)$, $\$A(L)$, $\$A(U)$, and $\$A(T)$ represent the value, lower bound, upper bound, and time interval components of the attribute $\$A$, respectively. Example 4 in HQUEL looks like

```
Range of e is EMPLOYEE
Retrieve ( <e.$salary(T) ∩ e.$dept(T), e.ename> )
  where e.$salary(V) > 30K and e.$dept(V) =
  "Toy" and e.$salary(T) ∩ e.dept(T) ≠ φ.
```

HQUEL handles set intersection, union, and difference in the where clause as new additional operators. However,

TABLE II
COMPARISONS INVOLVING TBE AND QUEL-BASED TEMPORAL LANGUAGES

Languages	HQUEL	HTQUEL	TQUEL	TBE
Temporal Operators	√	√	√	√
Aggregate Operations	√	—	√	√
Query Optimization	—	—	—	√
Noncontiguous Intervals	√	√	√	√
Implementation	—	—	√	—

set difference and set union operations create set triplet valued attributes when they are specified in the target list. Aggregate operations have been incorporated to HQUEL. HQUEL is based on the HRA of Section III, and has the same expressive power as HRA. We summarize our comparisons in Table II.

VIII. CONCLUSIONS

The relational query language, Time-by-Example, and its query processing methodology are presented. The underlying data model is a modified relational model which uses attribute time stamping and nonfirst normal form relations, and allows different attribute types (simple, set, triplet, and set triplet) to coexist in historical relations to represent time-dependent and time-independent attributes. The three-dimensional view of relations (time cubes) is also maintained at the user level. The model captures the semantics of time elegantly, and avoids the redundancies and complexities inherent in other proposals which are based on tuple time stamping.

In this paper, we have also described a methodology for translating TBE queries into HRA expressions and for creating a parse tree, the internal nodes and leaves of which are HRA operations and relations, respectively. This methodology lends itself to the optimization of query processing. New access procedures, however, are needed for the triplet-formation, triplet-decomposition, and slice operations. These operations can be carried out simultaneously with join operations. The access paths defined for the pack and unpack operations in STBE [28] can also be used in TBE. However, the cost model of STBE should be revised to accommodate the triplet-valued (set-triplet-valued) attributes. Efficient storage techniques are also needed for historical relations. They are beyond the scope of this paper, and we plan to explore them in a separate study.

Clearly, these are several features of TBE which make it quite complex for novice users. These are

1) the concept of ROOT query and subqueries,
2) matching variables to values,
3) parameter passing between windows using free and fixed variables, i.e., binding rules,
4) simple, set-valued, triplet-valued, and set-triplet-valued variables and attributes.

While it is true that these features make the language more complex than, say, QBE, they also make the language more powerful. (TBE and STBE are relationally complete, whereas QBE is not). This issue is discussed in [27]. We now comment on these features.

We have recently finished a study [1] about the "user-friendliness tests" of the features listed above. Fixed variables look complicated at first glance, but they actually are not. After all, they are the values taken from the present window down to another window. This is identical to passing parameters in procedure calls using the "call by value" technique. Thus, we think that somebody with the knowledge of a high-level programming language can grasp the concept of fixed variables. As far as subqueries are concerned, they are closely associated with windows, and windowing is extensively used in commercial software packages.

There is a simple and straightforward logic behind the matching rules of variables (Table I): there are two types of attribute values, 1) atomic (triplet) values, 2) sets of atomic (triplet) values. For variable instantiations, if the user wants to pick a tuple component [a single value for (1), a set for (2)], (s)he specifies a simple (triplet) or a set-valued (set triplet) variable. This is the case for compatible specification. On the other hand, if the user wants to pick one of the values in a set (set triplet), (s)he specifies a simple (triplet) variable on the corresponding set (set triplet) valued tuple component. This is the case of incompatibility.

As for binding rules, fixed variables are used for passing parameters from one window to another. They can be considered as constants in the passed window, received from the parent window. Either a single value (triplet) or a set of values (triplets) are passed down to a window (subquery). In the subquery, tuples matching these values are examined. There are two possibilities. Either a tuple component matches a single passed value (triplet or set) or a tuple component matches to one of the values (triplets) in the set of passed values (triplets). This forms the logic behind the binding rules table in Fig. 4.

Triplets and sets of triplets are the fundamental concepts of TBE and the underlying data model. They have a simple structure, and their intuitive meaning is simple.

We should also point out that our discussion of the TBE in this paper is meant to be a software specification not really the true syntax of a commercial implementation. (However, we have written an object-oriented display manager for STBE which relies on the symbols *, $, etc. to distinguish set-/simple-valued attributes, free/fixed variables, etc.) The addition of color to a TBE display manager would probably enhance the query specification process greatly.

APPENDIX
BNF SPECIFICATION FOR TBE

```
<TBE query spec>  ::= <root window> { <subquery> }
<root window>     ::= ROOT <window>
<subquery>        ::= <subq name> <window>
<subq name>       ::= <ucase ident>
<window>          ::= <output box> { <relation skeleton> }
                      <range box>
                      <condition box>
```

```
<output box>        ::=output { <out column> }
<out column>        ::=<var> | <arith exp> | <out triplet>
<arith exp>         ::=<arith exp> <arith op> <arith exp> |
                       ( <arith exp> ) | <simple var> |
                       <integer> |
                       <aggreg> | <$var comp>
<aggreg>            ::=<aggfunction> (<subq name>) |
                       COUNT (<subq name>) |
                       <aggfunction> (subq name> "|"
                       <integer> )
<aggfunction>       ::=AVE | SUM | MAX | MIN | WAVE
<arith op>          ::=+ | - | * | /
<out triplet>       ::="<"<interval part>,
                       <value part>">" |
                       "<"<out tr spec>,<out tr spec>,<out
                       tr spec>">"
<value part>        ::=<out tr spec>
<out tr spec>       ::=<simple var> | <$var comp>
<interval part>     ::=<interval part> <temp set op>
                       <interval part> |
                       (<interval part>) | <triplet interval>
<temp set op>       ::=overlap | extend | without | intersection |
                       union | difference
<triplet intval>    ::=<$var interval> | <interval const>
<rel sklton>        ::=ϵ | <rel name> { <attr spec> }
<rel name>          ::=<ucase ident>
<attr spec>         ::=<attr name> | <simple attr name>
                       <simple spec> |
                       <*attr name> <*spec> | <$attr name>
                       <$spec> |
                       <*$attr name> <*$spec>
<simple attr name>  ::=<identifeier>
<*attr name>        ::=*<identifier>
<$attr name>        ::=$<identifier>
<*$attr name>       ::=*$<identifier>
<simple spec>       ::=<simple var> | <$var comp> |
                       <comp> <const>
<comp>              ::=ϵ | <comp op>
<*spec>             ::=<*var> | <simple var> | <$var
                       comp>
<$spec>             ::=<$var> | "<"<tr comp>,<tr
                       comp>,<tr comp>">" |
                       "<"<$var interval>,<tr comp>">"
<*$spec>            ::=<*$var> | <$var>
<tr comp>           ::=ϵ | <simple var> | <$var comp> |
                       <const>
<simple free var>   ::=_<identifier>
<*free var>         ::=_*<identifier>
<$free var>         ::=_$<identifier>
<*$free var>        ::=_*$<identifier>
<simple bound var>  ::=—<identifier>
<*bound var>        ::=—*<identifier>
<$bound var>        ::=—$<identifier>
<*$bound var>       ::=—*$<identifier>
<simple var>        ::=<simple free var> | <simple bound
                       var>
<*var>              ::=<*free var> | <*bound var>
<$var>              ::=<$free var> | <$bound var>
<*$var>             ::=<*$free var> | <*$bound var>
<$var comp>         ::=<$var>.l | <$var>.u | <$var>.v
<$var interval>     ::=<$var>.T
<condition box>     ::=ϵ | condition <logic exp>
<logic exp>         ::=<logic exp> AND <logic exp> |
                       <logic exp> OR <logic exp> |
                       NOT <logic exp> |
                       (<logic exp>)| <simple comp exp> |
                       <set comp exp> | <temp exp>
<simple copm exp>   ::=<operand> <comp op> <operand>
<operand>           ::=<simple var> | <$var comp> |
                       <aggreg> | <const>
<comp op>           ::=> | >= | < | <= | = | =
<set comp exp>      ::=<*set operand> <set comp op> <*set
                       operand> |
                       <*$set operand> <set comp op>
                       <*$set operand> |
                       <$operand> <set memb op>
                       <*$var> |
                       <operand> <set memb op> <*var> |
                       <temp exp>
<*set operand>      ::=<*var> | <set const>
<set comp op>       ::=] | ]= | [ | [= | ≈ | /
<set memb op>       ::=is in | is not in
<*$set operand>     ::=<*$var> | <$set const>
<set const>         ::="{"<value list>"}"
<value list>        ::=<const> {,<const>}
<$set const>        ::="{"<$list>"}"
<$list>             ::=<$const> {,<$const>}
<$const>            ::="<"<iconst>,<iconst>,<const>">"
<temp exp>          ::=<interval part> <temp comp op>
                       <interval part> |
                       <interval part> <temp comp op>
                       EMPTY SET
<temp comp op>      ::=intersects | preceeds | contains | does not
                       contain |
                       is | is not | <set comp op>
<range box>         ::=ϵ | range { <range list> }
<range list>        ::=(<simple var>) actual : <const list> |
                       (<simple var>) actual : <iconst>,
                       . . . ,<iconst> |
                       (*var) incremental :
                       (<iconst>,<iconst>,<iconst>) |
                       (<list of var>) <union of rel> |
                       <range list>
<const list>        ::=<const> {,<const> }
<const>             ::=<integer> | <string>
<iconst>            ::=<digit> { <digit> }
<list of var>       ::=<variable> {,<variable> }
<variable>          ::=<simple var> | <*var> | <$var> |
                       <*$var>
<union of rel>      ::=<relation> { OR <relation> }
<relation>          ::=<subq name> | <rel name>
<identifier>        ::=<lcase let> { <lcase let> | <digit> }
<ucase ident>       ::=<ucase let> { <ucase let> | <digit> }
<integer>           ::=<digit> { <digit> }
<digit>             ::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
<lcase let>         ::=a | b | c | . . . | x | y | z
<ucase let>         ::=A | B | C | . . . | X | Y | Z
<var>               ::=<simple var> | <*var>
                       <*$var> | <*$var> | <$var comp>
<string>            ::=<first let> { <lcase let> | <ucase let>
                       | <digit> }
<first let>         ::=<lcase let> | <ucase let>
<interval const>    ::=[<integer>,<integer>)
```

ACKNOWLEDGMENT

We would like to thank the reviewers for their extensive comments.

REFERENCES

[1] W. Abdul-Qader and G. Ozsoyoglu, "A comparative human factors study of QBE and STBE," submitted for publication, 1987.
[2] T. L. Anderson, "The database semantics of time," Ph.D. dissertation, Univ. Washington, Seattle, 1981.
[3] G. Ariav, "Preserving the time dimension in information systems," Ph.D. dissertation, Dep. Decision Sci., Univ. of Pennsylvania, Philadelphia, 1984.

[4] E. Breutmann, E. Falkenberg, and R. Maurer, "CSL: A language for defining conceptual schema," in *Data Base Architecture*, G. M. Nijjsen, Eds. Amsterdam: North-Holland, 1979.
[5] J. A. Bubenko, "The temporal dimension in information processing," in *Architecture and Models in Database Management*, G. M. Nijjsen, Ed. Amsterdam: North-Holland, 1977.
[6] J. Ben-Zvi, "The time relational model, " Ph.D. dissertation, Univ. California, Los Angeles, 1982.
[7] J. Clifford and D. S. Warren, "Formal semantics for time in databases," *ACM Trans. Database Syst.*, vol. 6, no. 2, 1983.
[8] J. Clifford and A. U. Tansel, "On an algebra for historical relational databases: Two views," in *Proc. ACM SIGMOD Conf.*, 1985.
[9] E. F. Codd, "A relational model of data for large shared databanks," *Commun. ACM*, vol. 13, June 1970.
[10] P. Dadam, V. Lum, and H. D. Werner, "Integration of time versions to relational database systems," in *Proc. VLDB Conf.*, 1984.
[11] S. K. Gadia and J. H. Vaishnav, "A query language for a homogeneous temporal database," in *Proc. ACM PODS Conf.*, 1985.
[12] S. K. Gadia, "A homogeneous relational model and query languages for temporal databases," *ACM Trans. Database Syst.*, vol. 13, no. 4, Dec. 1988.
[13] S. Ginsberg and K. Tanaka, "Interval queries of object histories," in *Proc. VLDB Conf.*, 1984.
[14] G. Jaeschke and H. Schek, "Remarks on the algebra of non first normal form relations," in *Proc. ACM PODS Conf.*, 1982.
[15] S. Jones, P. Mason, and R. Stamper, "LEGOL 2.0: A relational specification language for complex rules," *Inform. Syst.*, vol. 4, no. 4, 1979.
[16] M. R. Klopprogge and P. C. Lockermann, "Modelling information preserving databases: Consequences of the concept of time," in *Proc. VLDV Conf.*, 1983.
[17] A. Klug, "Abe—A query language for constructing aggregates-by-example," in *Proc. 1st LBL Int. Workshop Statistical Database Management Syst.*, 1981.
[18] ——, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *J. ACM*, July 1982.
[19] V. Lum *et al.*, "Designing DBMS support for the temporal dimension," in *Proc. ACM SIGMOD Conf.*, 1984.
[20] S. B. Navathe and R. Ahmed, "TSQL—A language interface for history databases," in *Proc. Temporal Aspects in Inform. Syst. Conf.*, 1987.
[21] Z. M. Ozsoyoglu and G. Ozsoyoglu, "An extension of relational algebra for summary tables," in *Proc. 2nd LBL Int. Workshop Statistical Database Management*, 1983.
[22] Z. M. Ozsoyoglu and G. Ozsoyoglu, "A query language for statistical databases," in *Query Processing in Database Systems*, W. Kim, D. S. Reiner, and D. Batory, Ed. New York: Springer-Verlag, 1985.
[23] Z. M. Ozsoyoglu and G. Ozsoyoglu, "STBE—A database query language for manipulating summary data," in *Proc. IEEE COMPDEC Conf.*, 1984.
[24] G. Ozsoyoglu and V. Matos, "On optimizing summary-table-by-example queries," in *Proc. ACM PODS Conf.*, 1985.
[25] G. Ozsoyoglu, M. Z. Ozsoyoglu, and V. Matos, "Extending relational algebra and relational calculus with set-valued attributes and aggregate functions," *ACM Trans. Database Syst.*, vol. 12, no. 4, Dec. 1987.
[26] G. Ozsoyoglu, V. Matos, and Z. M. Ozsoyoglu, "Query processing techniques in the summary-table-by-example database query language," submitted for publication, 1985.
[27] G. Ozsoyoglu, and H.-Q. Wang, "A relational calculus with set operators, its safety, and equivalent graphical languages," submitted for publication, 1987.
[28] A. Segev and A. Shoshani, "Modeling temporal semantics," in *Proc. Temporal Aspects in Inform. Syst. Conf.*, 1987.
[29] ——, "Logical modeling of temporal data," in *Proc. ACM SIGMOD Conf.*, 1987.
[30] A. Shoshani and K. Kawagoe, "Temporal data management," in *Proc. VLDV*, 1986.
[31] R. Snodgrass, "The temporal query language TQuel," in *Proc. ACM PODS Conf.*, 1984.
[32] R. Snodgrass and I Ahn, "A taxonomy of time in databases," in *Proc. ACM SIGMOD Conf.*, 1985.
[33] R. Snodgrass, "A temporal query language," *ACM Trans. Database Syst.*, vol. 12, no. 2, June 1987.
[34] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, Sept. 1976.
[35] B. Sundgren, *Theory of Databases*. New York: Petrocelli/Charter, 1975.
[36] A. U. Tansel and J. Clifford, "On an algebra for historical relational databases: Two views," in *Proc. ACM SIGMOD Conf.*, 1985.
[37] A. U. Tansel and M. E. Arkun, "HQUEL: A historical query language," in *Proc. 3rd Int. Workshop Statistical Sci. Database Management*, Luxembourg, 1986.
[38] A. U. Tansel, "Adding time dimension to relational model and extending relational algebra," *Inform. Syst.*, vol. 13, no. 4, 1986.
[39] J. Ullman, *Principles of Database Systems*. Arlington, VA: Computer Science Press, 1982.
[40] G. Wiederhold, J. F. Fries, and S. Weyl, "Structured organization of clinical databases," in *Proc. AFIPS*, 1975.
[41] M. M. Zloof, "Query-by-example: A database language," *IBM Syst. J.*, vol. 16, no. 4, 1977.

Abdullah U. Tansel received the B.S. degree in management from the Middle East Technical University, Ankara, Turkey, in 1972, and the M.S. and Ph.D. degrees in computer engineering from the same university in 1974 and 1981, respectively. He also received the M.B.A. degree from the University of Southern California, Los Angeles.

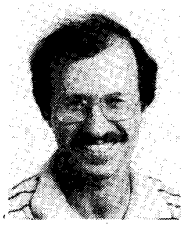
His research interests include temporal databases, distributed databases, and management information systems. He currently is an Associate Professor of Computer Information Systems at the City University of New York. Before joining CUNY, he taught in the Middle East Technical University.

Dr. Tansel is a member of IEEE Computer Society and the Association for Computing Machinery.

M. Erol Arkun received the B.S.E.E. degree from Robert College in 1963 and the M.S.E.E. and Ph.D. degrees from the Polytechnic Institute of Brooklyn, Brooklyn, NY, in 1965 and 1972, respectively.

He was an Assistant, then an Associate Professor of Computer Engineering at the Middle East Technical University, Ankara, Turkey, until 1980. From 1980 to 1986 he was an Associate Professor at the Bernard M. Baruch College of the City University of New York. Since 1986 he has been a Professor and the Director of the Computer Center in the newly found Bilkent University, Ankara. His interests include data management, information systems, and database systems.

Dr. Arkun is a member of the Association for Computing Machinery and an affiliate of the IEEE Computer Society.



Gultekin Ozsoyoglu received the B.Sc. degree in electrical engineering and the M.Sc. degree in computer science from the Middle East Technical University, Ankara, Turkey, in 1972 and 1974, respectively, and the Ph.D. degree in computer science from the University of Alberta, Edmonton, Alta., Canada, in 1980.

He is presently an Associate Professor of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH. From 1980 to 1983 he was with the Cleveland State University. His research interests include databases and security.