Extensions to SQL for Historical Databases

NANDLAL L. SARDA

Abstract—**A historical database management system (HDBMS) is a high-level facility which provides timing information and automatically maintains history (i.e., past) data. It also provides facilities for time-related queries. One such HDBMS is described in this paper. Our HDBMS uses an extended relational data model with state-oriented, instead of "cubic," conceptualization. Two types of *historical* relations, called state and event relations, are provided for modeling real-world objects. The popular query langauge SQL has been extended for definition, retrieval, and update of historical relations. The extended SQL, called HSQL, is a superset of SQL. We define a few primitive algebra operations for historical relations, and use them as a basis for extensions to SQL. By doing so, HSQL retains the elegant structural and algebraic framework of SQL. HSQL contains a few new clauses, many operations and built-in functions on time domain, and facilities for retrospective updates and time-rollback.**

Index Terms—**Data models, database management, historical database, modeling of time, query languages, relational algebra, relational model, SQL.**

I. Introduction

THE ubiquitous nature of time requires that a database management system provide facilities for modeling of time for the real-world applications. Consequently, considerable research activity is being directed to the study of time in databases (e.g., see the project summaries of these efforts in [13]). Various aspects of time modeling are being investigated in this research. They include extensions to data models (mostly the relational), new algebra operations and query languages, and efficient storage structures for the monotonically increasing history.

Snodgrass and Ahn [14] were the first to clearly distinguish between two measures of time, called *real-world time*, the time at which an event takes place in the real world, and the *system time*, the time at which it is registered with the computer system. They defined four types of databases depending on which time measures are supported by a DBMS: snapshot (conventional database without time), rollback (with only system time), historical (with only real-world time), and the temporal (with both time measures) databases.

A "cubic view" of database is commonly proposed to capture the time dimension [1], [4], [15]. Here, time is added as a third dimension to the two-dimensional (i.e., flat) tables of the relational data model. At the representational level, the relations are extended to include time attributes. The following four possible approaches emerge in the literature:

1) instant-stamping of tuples [1]: each tuple includes a time value at which the data in the tuple became current;
2) interval-stamping of tuples [7], [9], [14]: each tuple includes two time values that define a time interval during which the data were current;
3) instant-stamping of attributes [5]: a time value is associated with each attribute value, and
4) interval-stamping of attributes [5]: two time values defining an interval are associated with each attribute value in a tuple.

The primary reason for advocating attribute stamping is that values of attributes within a relation vary at different rates. In the attribute-stamping approaches, each tuple contains, in effect, a history for each attribute. Consequently, such relations are not even in the first normal form.

Depending on the representational approach chosen, the researchers have proposed different types of algebra for historical/temporal databases. McKenzie and Snodgrass [17] have carried out a detailed comparison of the various proposals. Primarily, however, time-slice and joins based on time emerge as useful additions/extensions to the standard relational algebra.

The popular query languages SQL and Quel have been considered for extensions to support time-oriented facilities. Snodgrass [15] has proposed TQuel, which extends Quel with "event" and "interval" type of relations, and includes new clauses for specifying predicates on time attributes, for time slicing, and for rollback to an earlier point in time. A large number of built-in functions have also been defined for TQuel [16]. TQuel (like Quel itself) has a rather complex framework for aggregate functions.

Extensions to SQL [6] have been proposed by Ariav [1] and Navathe and Ahmed [7]. TOSQL of Ariav uses a cubic view of data with instant-stamping of tuples. Such a model makes definition of otherwise-simple operations like select and project quite complex. TOSQL has some complex clauses that do not retain the structural framework of SQL. Moreover, it does not address the question of how to relate data across two or more relations.

The TSQL of Navathe and Ahmed [7] is based on interval-stamping of tuples. It does not differentiate between event and state type of objects. TSQL includes a WHEN clause for specifying predicates on time attri-

Manuscript received April 20, 1989; revised December 6, 1989. This work was supported in part by a grant from the Natural Science and Engineering Research Council of Canada when the author was at the University of New Brunswick, Saint John, Canada.

The author is with the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay 400 076, India.

IEEE Log Number 9035101.

butes, a TIME-SLICE clause, and a number of built-in functions for "temporal" ordering of data. It also includes a new operation called MOVING WINDOW, which may be of interest in some special applications.

At least two important issues are not addressed by these extensions to the query languages. First, they do not provide a *direct* way for relating *concurrent* data from two or more relations. We expect this to be a very basic requirement for users of historical databases. Second, the question of different time granularities within the same database is not addressed adequately.

In this paper, we present our approach [8]–[11] to historical databases, which is characterized by

1) state-oriented view of historical database: state of a database object is defined by values of its attributes, and its state prevails over an interval of time. An "event" is a special case, where state prevails over one time instant only,

2) interval stamping of tuples at the representation level,

3) support for real-world time measure only.

The proposed HDBMS is based on an extended relational data model. In fact, data are stored as conventional relations with two added attributes for time-stamping. The standard relational algebra operations are directly applicable without any change in their meaning. We define two new primitive operations for manipulating the time domain. We show that the more useful operations, such as time-slicing and time-based joins, can be defined as "high-level" operations using the new primitives and the standard operations. The extended algebra, a *strict* superset of the standard relational algebra, is used as a basis for extending the popular query language SQL.

The remainder of this paper is organized as follows. In Section II, we give an overview of our historical data model, outlining the main modeling concepts. The time domain is defined in Section III. The operations and functions on time defined here are directly incorporated in our extensions to SQL. In Section IV, we extend the relational algebra by two new primitive operations. The extensions to SQL are contained in Section V. It addresses all aspects of HSQL: data definition, data retrieval, data update, retrospective updates, use of nonhistorical relations, and time-rollback facility. Finally, in Section VI, we make concluding remarks about our historical data model and extensions to SQL.

II. The Historical Data Model: An Overview

Modeling: In our approach, to model the requirements of an application, the database designer considers only the "current perspective" of the requirements. In this perspective, the requirements will typically be stated using the "current" tense, ignoring the need to store history data and timing information. It is as if the designer were to design a "snapshot" database, much as in the conventional database design. For example, we will consider an employee to have a rank and a salary rather than explicitly considering an employee to have had a history of ranks and salaries over a period of time.

In the conventional database design, the designer identifies entities, relationships, their identifiers, and attributes. Alternatively, the designer identifies objects (or, object types, to be specific) for modeling them as relations. Each object has a unique identifier and a set of functionally-dependent attributes. An effort is made to identify "minimal" objects, often using the theory of normalization.

The modeling approach in our historical data model essentially remains the same. The values of attributes of an object define its *state*. A change of value for any of its attributes represents a change of state. A state prevails over an interval of time, during which none of the attributes change their values.

This concept of state defined by a set of attributes might need an additional step for "time normalization" [7] in the design process. A database object identified earlier may need decomposition if its attributes vary at drastically different rates. This is the classical file segmentation problem [19], where the designer tries to balance the storage and access costs. For instance, the "employee" object has attributes *E#* (unique), RANK and SALARY. These attributes may be accessed together most of the times. Although RANK and SALARY may change at different times and at different rates, the changes are relatively infrequent. Therefore, we let RANK and SALARY together define *state* of an employee object. An alternative would have been to decompose it into two objects, one defined by *E#* and RANK, the other by *E#* and SALARY.

In our model, we also provide for a special kind of object, called *event*. It is an object which prevails for only one time unit. The designer would identify such objects during the requirements analysis. Since every object of event type exists for one instant, the concept of history does not apply to it. Alternatively, one could say that an event becomes history as soon as it occurs. The concept of update does not apply to event-type objects.

Historical Relations and their Representation: The objects of state and event type are defined as *historical* relations. For such relations, the HDBMS will maintain timing and history data automatically. A historical relation is defined by listing its "visible" attributes, which do *not* include the timing attributes that will be automatically added by HDBMS. The designer also needs to specify the required time *granularity*. HDBMS provides a hierarchy of time units for this purpose, ranging from very coarse to very fine.

The granularities should be carefully chosen as they affect relative ordering of events recorded in the database. For example, if DATE is the granularity for SHIPMENT object, two shipments received on the same day (but at different times of the day) will be treated to have occurred at the same time.

To clearly understand the basic functions of proposed HDBMS, consider the following definition of a relation

(given in extended SQL):

```
CREATE STATE TABLE EMP
  (ENO   CHAR(10) NOT NULL   /*  employee
                              number */
   PROJ  CHAR(10),           /* project */
   SAL   DECIMAL(5)          /* salary */
   UNIQUE (ENO))
WITH TIME GRANULARITY DATE.
```

Here, EMP is a *historical* relation of state type. HDBMS will add two more attributes to this relation, named FROM and TO, whose values are in TIME domain, and which together define a (closed) nonnull interval of time. In the proposed HDBMS, EMP will actually be stored as two segments defined by the following schemes:

CURRENT-EMP (ENO, PROJ, SAL, FROM, TO)
HISTORY-EMP (ENO, PROJ, SAL, FROM, TO).

The former contains tuples pertaining to the current states only, and the latter contains tuples representing history data. The value of attribute TO in CURRENT-EMP will always be the current time value, represented by keyword NOW. Note that the two segments (to be themselves referred as relations hereafter) are union-compatible.

No such segmentation is defined automatically for event type of historical relations. Also, for these, the values of FROM and TO attributes within a tuple will be equal since an event prevails for only one time unit. The alias AT can be more meaningfully used instead of FROM for such relations.

Let us next consider some operations on EMP. At time $t1$, we wish to insert the following data for a new employee:

<'SMITH', 'LOTUS', 30000>

These data will be added to the current segment as follows:

<'SMITH', 'LOTUS', 30000, $t1$, NOW> (1)

At time $t2$, we wish to change SMITH's salary to 40000. The tuple (1) above in current segment is replaced by

<'SMITH', 'LOTUS', 40000, $t2$, NOW> (2)

and the following tuple is added to the history segment:

<'SMITH', 'LOTUS', 30000, $t1$, $t2-1$> (3)

where $t2-1$ is one instant (on same granularity level) before $t2$. Finally, at time $t3$, we wish to delete SMITH from EMP. The tuple (2) above is deleted from current segment and added to the history segment as follows:

<'SMITH', 'LOTUS', 40000, $t2$, $t3-1$> (4)

It should be noted that generation of history tuples (3) and (4) is a byproduct of the update and delete operations on EMP. HDBMS generates them automatically.

Key of a Historical Relation: The concept of *key* can now be defined for historical relations. The attribute K of R is a key iff at any time instant t, we do *not* have

r, s in R and
$r.K = s.K$ and
intervals in r and s both contain t.

This definition of key basically indicates that its values are "time-unique" (and not tuple-unique as in the relational model).

Integrity Constraints: HDBMS enforces the following integrity constraints.

1) A tuple with null interval is not stored in a relation.
2) A historical relation does not contain tuples with same visible attribute values but overlapping or consecutive time intervals. Such tuples are automatically "coalesced" by merging their time intervals. The coalesce operation is defined more formally in Section IV.
3) A new tuple can be added only if the current segment does not contain a tuple with same key value.
4) Only the tuples from current segment can (normally) be updated or deleted.

Remarks:

1) HDBMS supplies values for attributes FROM and TO from some internal clock. Thus, they represent what has been termed as system or transaction time [14]. In a real-time or on-line environment, we expect the system time to be generally the same as the actual or effective time. It is possible that some transactions are not recorded immediately, or stored data must be corrected retrospectively. In such cases, HDBMS must be supplied with effective time (to override its internal clock time). HDBMS provides facilities for this purpose. Thus, effectively, the time values stored in historical relations are real-world times.

2) A designer should associate clear application-oriented meaning to the time attributes. For example, if ORDER is a state-oriented object, the attribute FROM could mean the date on which the order was received (as against date sent by customer) and TO could mean the date on which it was completely filled and paid (upon which time the order would be deleted).

3) Additional time attributes may be defined to capture other time measures (e.g., proposed time). These would be visible attributes to be defined and maintained explicitly. We extend SQL to include TIME as an attribute type.

III. The Time Domain: Representation and Operations

Time can be imagined to be measured using a clock of suitable granularity. Every "tick" of the clock represents a time *instant*. The value of an instant is the number of ticks from the start of the clock. Thus, as in [5], time is isomorphic to the natural numbers, and the set of all times is a linear order, i.e., given instants $t1$ and $t2$, we either have $t1 = t2$, or $t1 < t2$, or $t1 > t2$.

The *current* time refers to the latest clock tick, and is denoted by NOW. Thus, NOW can be thought as a moving time variable as in [4].

An *interval* is a sequence of consecutive time instants. It is represented as $t1..t2$, where $t1 <= t2$. It includes all time instants from $t1$ to $t2$, *both* inclusive. The interval $t1..t1$ contains the single instant $t1$. The *null interval* does not include any time instant. The interval $t1..t2$, where $t2 < t1$, is also considered to be null.

HDBMS provides real-world units for measuring time. The following units, which form a hierarchy from coarse to fine granularity, are provided:

YEAR YEAR:MONTH
DATE (or, YEAR:MONTH:DAY)
DATE:HOUR DATE:HOUR:MIN
DATE:HOUR:MIN:SEC

We define many operations and built-in functions on instants and intervals. Many more can be added to this list (e.g., see [20]). They are described below. In the following, t's represent instants and p's represent intervals.

A. Instant Comparisons

These are the usual comparison operations:

<, >, =, <=, >=, /= (not equal))

They produce a Boolean result.

B. Interval Comparisons

The following infix operations are included in HSQL:

t in p = true, if t is included in interval p.
$p1 = p2$ = true, if both $p1$ and $p2$ include the same set of time instants, false otherwise.
$p1$ *overlap* $p2$ = true, if $p1$ and $p2$ include at least one common instant, false otherwise.
$p1$ *contains* $p2$ = true, if all instants of $p2$ are also contained in $p1$, false otherwise.
$p1$ *meets* $p2$ = true, when $p1$ is $t1..t2$ and $p2$ is $t2+1..t3$, false, otherwise.
$p1$ *adjacent* $p2$ = ($p1$ *meets* $p2$) *or* ($p2$ *meets* $p1$).
$p1$ *precedes* $p2$ = true, when $p1$ is $t1..t2$ and $p2$ is $t3..t4$ and $t2 < t3$, false otherwise.

C. Interval Operations

The following infix operators are provided:

"..": make interval, given two time instants.
"+": concatenate (i.e., merge) two overlapping or consecutive intervals.
"*": extract common part of two overlapping intervals.
Note: ".." is not commutative; "+" is not associative, and "*" is both commutative and associative.

D. Functions (built-in)

We will use @ as a wildcard in function names to represent one of the following time units: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS.

a) *Extracting time/intervals* from tuples: if t is a tuple variable, the t.INTERVAL, t.FROM (or, t.AT), t.TO can be used to refer the time instants/interval in tuple t.

b) *Segment selection*: Given a historical state relation R, CURRENT (R) and HISTORY (R) can be used to refer to current and history tuples, respectively, in R. The reference R by itself refers to the union of both segments. It would contain both current and history tuples.

c) *Elapsed time*: given two instants $t1$ and $t2$ of the same granularity, the function ELAPSED@($t1$, $t2$) measures elapsed time in units given by @. The granularity level of $t1$ and $t2$ must be equal or finer than @. For example,

ELAPSEDYEARS (1984, 1987) gives 4 years
ELAPSEDMONTHS (1985:02:15, 1985:08:06) gives 5 months
ELAPSEDMONTHS (1985:02:01, 1985:08:06) gives 6 months

d) *Predecessor/successor of instants*: given $t1$, PRED($t1$) gives instant preceding $t1$ and SUCC($t1$) gives instant succeeding $t1$ on same granularity level.

E. Comparing Instants of Different Granularities

The issue of comparing instants and intervals of different granularity must be resolved with care. There are two ways to convert a time value of finer granularity to a coarser granularity (much like truncation and roundoff in going from real to integer). On the other hand, an instant represents an interval on a finer granularity level.

HSQL permits direct comparisons of instants of different granularities in, what seems to us, a natural way. Consider one example of a comparison:

1985:06 < 1985:07:27.

We first convert coarser instant 1985:06 into interval 1985:06:01 .. 1985:06:30. Next, each instant in that interval is compared for < with 1985:07:27. The comparison should hold for each of them. Effectively, this comparison reduces to

1985:06:01 < 1985:07:27 *and* 1985:06:30 < 1985:07:27.

If two intervals involved in an operation have different granularities, the coarser interval is converted into the finer one. As an example, the interval 1987:05 .. 1988:02 is same as interval 1987:05:01..1988:02:29 at the granularity level of DATE.

HSQL also includes functions for conversion of granularities. The function UPTO@ is used to convert a time value to a coarser granularity by truncating components at finer levels. For example, UPTOMONTHS(1986:08:25) gives 1986:08. Another function, called INTERVAL@, could be used to convert an instant into an interval at finer granularity level given by @. For example, INTERVALMONTHS(1984) gives 1984:01..1984:12.

IV. Historical Relational Algebra

The algebraic operations form a convenient basis for defining, understanding, translation, optimization, and execution of query languages. The standard relational model [18] defines as primitives the operations: Union (U), Set difference ($-$), Cartesian product (X), Projections (π), and Selection (σ). Additional, and more useful, operations, such as join, can be expressed in terms of these. Moreover, the above primitive operations form a basis for defining completeness criteria for query languages [18].

Standard Operations: The underlying data model for the proposed historical data model is the relational model itself. A historical relation R with X as a visible set of attributes is represented by a relation of the standard relational model as

R (X, FROM, TO).

To maintain consistency with the relational model, the relational operators can be applied to R with their usual meaning. However, the result of some of these operations may or may not be a historical relation. Specifically,

1) $\pi_Y(R)$: result may not be a historical relation. It would be a historical relation only if Y includes the time attributes FROM and TO.

2) $\sigma_F(R)$: result is a historical relation.

3) R X S: (where R, S are historical relations): result is not a historical relation as each tuple in result contains two time intervals.

Completeness: Although the five primitive operators listed earlier are deemed complete, they fall short for the historical data model. The basic reason for their inadequacy is that the tuples in historical relations have been time-stamped with intervals, while it may be necessary to retrieve and manipulate data based on instants. We need new operations for converting data in time intervals to data at time instants and vice-versa. We extend the set of primitive operators by two new operators given below. The new operators cannot be expressed in terms of other relational operator. Hence, they are deemed primitive, to be included as basis for defining completeness of query languages.

New Primitive Operators:

1) Expand (e): $R1 = e(R2)$.

e is a unary operator whose operand and result are historical relations. The interval-stamped tuples in $R2$ are converted into instant-stamped tuples by replicating them for each instant included in their intervals. Specifically, if $R2$ contains the tuple

$<x, t1, t2>$

it will produce the following in $R1$

$<x, t1, t1>$
$<x, t1+1, t1+1>$
.....
$<x, t2, t2>$

where attribute values are same, but intervals are of single-instant duration.

There are two variations of expand when the result relation is to have coarser granularity than the source. The *expand-anytime* ($e1$) produces a result tuple for each instant covered partially or fully by the source tuple interval, and *expand-alltime* ($e2$) produces result tuples for only fully covered instants. To illustrate, consider the following source tuple:

<SMITH, LOTUS, 30000, 1985:06:10, 1986:02:18>.

Then, expand-alltime *by month* will produce the following seven tuples (one tuple for each month from July 1985 to Jan. 1986):

<SMITH, LOTUS, 30000, 1985:07, 1985:07>
<SMITH, LOTUS, 30000, 1985:08, 1985:08>
...
<SMITH, LOTUS, 30000, 1986:01, 1986:01>.

Expand-anytime *by month* will give nine tuples, from June 1985 to Feb. 1986. The specification expand-anytime *by year* will give the following 2 tuples:

<SMITH, LOTUS, 30000, 1985, 1985>
<SMITH, LOTUS, 30000, 1986, 1986>.

However, expand-alltime by year will not produce any tuple since the interval in source does not cover any full year.

2) *Coalesce* (c): $R1 = c(R2)$.

c is also a unary operator whose operand and result are historical relations. It basically performs the inverse function of e. It combines those tuples of $R2$ which have same (visible) attribute values but consecutive or overlapping time intervals into a single tuple in $R1$ with interval that includes intervals of combined tuples. Thus, if

$s, t \in R2$ and

$s.X = t.X$ and

$x.p$ *overlap* $t.p$ or $s.p$ *adjacent* $t.p$

then, s and t are combined in $R1$ as

$$<s.X, s.p + t.p>.$$

(Note: $s.p$ refers to interval in s.)

Proposition 1: If $R1$ is a historical relation satisfying the integrity constraints (specifically that no two tuples of $R1$ have same visible attribute values but adjacent and overlapping intervals) then

$$R1 = c\,(e\,(R1)\,).$$

New Useful Operators: The new primitive operators, expand and coalesce, along with the standard relational operators, can be used to define some highly useful operators for historical relations. We mainly define only one operation here, called *concurrent product*, using which the other operations commonly found in literature, such as concurrent join and time-slice, can be defined easily.

1) *Concurrent Product (Xt)*: $R3 = R1\ Xt\ R2$.

Xt is a binary operation whose operands as well as result are historical relations. It differs from the Cartesian product in that it pairs only those tuples of $R1$ and $R2$ which have overlapping time intervals. The interval in the result gives the extent of overlap. Thus, if Y and Z are visible attributes of $R1$ and $R2$, then

$$R3 = R1\ Xt\ R2$$
$$= \{ < t.Y, s.Z, t.p * s.p > \mid t \in R1 \text{ and } s \in R2 \}.$$

Recall the integrity constraint that tuples with null intervals are automatically removed from historical relations.

Proposition 2: Xt is commutative (note that interval operation * is also commutative).

Proposition 3: If Y and Z are attributes of $R1$ and $R2$, respectively, then

$$R1\ Xt\ R2 = c\left(\pi_A\left(\sigma_F\left(e\left(R1\right) X\, e\left(R2\right)\right)\right)\right)$$
where $A = \{ Y, Z, R1.\text{FROM}, R1.\text{TO} \}$ and
F is $R1.\text{FROM} = R2.\text{FROM}$.

Note that X is the Cartesian product.

Proposition 4: Given a singleton historical relation $\{<t1, t2>\}$ with no user-defined attributes, the time-slice operation τ suggested in literature [5], [17] can be defined as

$$\tau_{t1..t2}(R1) = R1\ Xt\ \{ <t1, t2> \}$$

V. HSQL: SQL Extended for Historical Databases

HSQL is a superset of the popular query language SQL [3], [6]. It provides facilities for definition, storage, retrieval, and update of historical relations. The extensions preserve the simple framework of SQL, also retaining its structural and syntactic simplicity. The extensions have a sound basis in the historical relational algebra described in Section IV.

HSQL provides the TIME domain for defining implicit (viz., FROM and TO) attributes as well as explicit time attributes. It directly supports the operations and functions on time described in details in Section III. Also, clauses have been added to support the historical algebra operations defined in Section IV.

The new facilities in HSQL will be described in the following sequence:

1) facilities for definition of historical relations and their time granularities (Section V-A),
2) facilities for retrieval of data (Section V-B)
3) facilities for data updates (Section V-C) and
4) facilities for retroactive updates (Section V-D)
5) other facilities (Sections V-E and V-F).

In defining the syntax of new features, we will refer the SQL grammar in [6], and give syntax definitions only for modified or new clauses.

A. Data Definition

A historical relation may be either of state or event type. A suitable time granularity is associated with it. A historical relation is defined by the CREATE TABLE statement, which gives it a suitable name and lists its visible attributes. The modified and new syntactic definitions are as follows:

base-table-def
= CREATE [STATE | EVENT] TABLE-name
(base-table-element-commalist)
granularity-def
granularity-def
= YEAR [:MONTH[:DAY[:HOUR[:MIN[:SEC]]]]]
= DATE [:HOUR[:MIN[:SEC]]].

Note that DATE is an abbreviation for YEAR:MONTH:DAY. HSQL permits user-defined columns to be of type TIME, defined as follows:

column-def = column-name TIME granularity-def.

Fig. 1 gives a schema for a historical database which will be used for illustration in the rest of the paper.

B. Data Retrieval

In standard SQL [6], a retrieval statement may contain the SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. In fact, all these may be present together in a query:

SELECT	scaler-exp-commalist
FROM	table-ref-commalist
WHERE	search-condition
GROUP BY	column-ref-commalist
HAVING	search-condition.

SQL has a simple basis (in terms of relational algebra) for understanding execution of such a query: it consists of the following steps.

1) Take Cartesian product of tables listed in FROM (let the result be table $T1$),
2) Select (σ) tuples of $T1$ which satisfy the search-condition in WHERE (let the result be $T2$),
3) Partition $T2$ into groups where the tuples in each group have same values for the columns listed in the GROUP BY clause,
4) Select those groups of $T2$ which satisfy the search-condition given in HAVING,
5) Project (π) required attributes (or, functions and expressions thereof) from the groups of step 4). When GROUP BY and HAVING clauses are absent, the result of step 2) is used for this step.

We have attempted to retain this simple framework in HSQL. In fact, the standard clauses of SQL have identical meanings in HSQL. New facilities have been added using additional keywords and through two new clauses. The important extensions are as follows:

1) The SELECT clause may contain the keyword COALESCED to coalesce (c) the result *before* applying the

```
CREATE STATE TABLE PROJECT
      (PID CHAR(8) NOT NULL
       LAB CHAR(10),
       LOC CHAR(10),
       MGR CHAR(10),
       UNIQUE (PID))
   WITH TIME GRANULARITY OF YEAR:MONTH;

CREATE EVENT TABLE SHIPMENT
      (PID CHAR(8) NOT NULL
       ITEM CHAR(10) NOT NULL
       QTY DECIMAL(3),
       DATESENT TIME DATE,
       UNIQUE (PID, ITEM))
   WITH TIME GRANULARITY OF DATE;

CREATE STATE TABLE EMP
      (ENAME CHAR(10) NOT NULL,
       PID CHAR(8)
      SAL DECIMAL(5),
      UNIQUE (ENAME))
   WITH TIME GRANULARITY OF YEAR:MONTH
```

Fig. 1. A schema for example historical database.

final projection. A preliminary projection on attributes used in SELECT is performed before applying coalesce. This facility cannot be used with grouping.

2) The FROM clause may contain the keyword CONCURRENT to indicate concurrent product (Xt) instead of Cartesian product of relations in the FROM clause.

3) A table-ref in FROM may be CURRENT(R) or HISTORY(R) or, simply, R where R is a historical relation.

4) When FROM CONCURRENT is specified, the condition in WHERE clause may refer to time attributes in the result of concurrent product simply as *.FROM, *.TO or *.INTERVAL.

5) New group-oriented functions have been added for use in SELECT and HAVING.

6) A new clause FROMTIME ... TOTIME ... for taking time-slices of historical relations has been included.

7) A new clause, called EXPAND BY, has been added, using which a historical relation (which may be the result of the concurrent product) can be expanded to convert intervals into instants (as in expand operator e) of indicated granularity.

A general query in HSQL may have all the following clauses (in that order):

```
FROMTIME ... TOTIME ...
SELECT [COALESCED] ...
FROM [CONCURRENT] ...
WHERE ...
EXPAND BY ...
GROUP BY ...
HAVING ...
```

The execution of such a query can be understood in terms of the following sequence of operations (parentheses indicate the corresponding clause):

1) Time-slice the relations in the FROM list (FROMTIME)
2) Apply X or Xt to those relations (FROM)
3) Apply σ (selection operation) (WHERE)
4) Apply e as per indicated granularity (EXPAND BY)
5) perform grouping (GROUP BY)
6) apply selection on groups (HAVING)
7) apply coalesce (c), if applicable (COALESCED)
8) perform final projections (SELECT).

The neat execution framework of a HSQL query thus retains the structural simplicity of SQL. When the extensions are not used, an HSQL query is equivalent to a SQL query, and when a clause or an extension is not used, the corresponding step in the execution is omitted. This simple framework is also desirable for formally defining semantics of HSQL (e.g., by using the historical relational algebra).

We now describe the syntax of new features in HSQL, again with reference to the SQL grammar in [6], and explain their action in more details, if necessary.

1) *Time-slicing*

```
FROMTIME time1 [TOTIME time2].
```

It is used to take time-slice (τtime1 .. time2) of historical relations listed in the FROM clause. time2 is taken as NOW by default.

2) *Selection* (in SELECT)

This specification gives the columns, or expressions or functions on columns, to be retrieved by the query. The syntax and facilities of SQL are directly applicable, including the built-in functions like SUM, COUNT, etc. In HSQL, the timing attributes may also be specified for selection using their names (FROM, TO, AT, or INTERVAL) with * or range-variable as qualifiers.

3) *from-clause*

```
from-clause = FROM [CONCURRENT] table-
                  ref-commalist
table-ref = table [range-variable]
          | segment-ref (table) [range-variable]
segment-ref = CURRENT | HISTORY.
```

As mentioned earlier, the word CONCURRENT is used to take concurrent product of tables in FROM. The words CURRENT and HISTORY allow us to choose a specific segment of a historical relation.

4) *where-clause*

The search-condition in where-clause may include conditions on time attributes using operations and functions defined in Section III.

5) *expand-by-clause*

```
expand-by-clause
   = EXPAND BY [ALLTIME | ANYTIME]
     level-spec
level-spec
   = YEAR | MONTH | DAY | HOUR | MIN | SEC.
```

The level-spec gives granularity upto which source tuples are to be expanded. The granularity of source tuples must be equal or finer than level-spec. The choice of ALLTIME (default) and ANYTIME indicates whether existence throughout an instant (recall that an instant is

an interval at finer level) or part of an instant is to be considered for expansion. Note that this clause may change granularity of the result.

6) *group-by-clause*

The syntax and meaning of this clause remains the same as in SQL, except that time attributes may be used for grouping.

7) *having-clause*

The syntax and meaning of this clause remains same as in SQL. However, a few new functions have been defined for use in the search condition. These functions can also be used in the SELECT clause. The new built-in functions are as follows:

FIRST or LAST(column-name): The value of specified attributes is extracted from that tuple of a group which has the earliest or latest time interval compared to other tuples in the group.

GROUP-UNION or GROUP-COMMON(* | interval-attribute): gives a time interval which is the concatenated or common interval (using the "+" or "*" operation) of the intervals of all tuples in a group after arranging those intervals in their chronological sequence, if necessary.

We now consider some examples using the database in Fig. 1 to illustrate the new features of HSQL.

Examples:

Query 1: When did SMITH join LOTUS project and at what salary:

```
SELECT FIRST(FROM), FIRST(SAL)
FROM EMP
WHERE ENAME = 'SMITH' and
      PID = 'LOTUS'.
```

Query 2: How many projects were in progress during 1985 at New York location (note: some of those may still be going on!)

```
FROMTIME 1985:01 TOTIME 1985:12
SELECT COUNT(DISTINCT PID)
FROM PROJECT X
WHERE X.LAB = 'NEW YORK'.
```

Query 3: List projects completed during managership of ROBERT. A project is assumed completed if it is not current. For a project to be selected, ROBERT should have been manager in its last state:

```
SELECT PID
FROM HISTORY(PROJECT) X
WHERE X.PID NOT IN
          (SELECT PID
           FROM CURRENT(PROJECT))
GROUP BY PID
HAVING LAST(MGR) = 'ROBERT'.
```

An alternative way to formulate the query is

```
SELECT PID
FROM PROJECT
GROUP BY PID
HAVING LAST(MGR) = 'ROBERT' and
       LAST(TO) /= NOW.
```

Query 4: Obtain number of shipments in each month of ROBERT's managership of projects during 1985.

```
FROMTIME 1985:01 TOTIME 1985:12
SELECT X.PID, *.FROM, COUNT(*)
FROM CONCURRENT PROJECT X,
                SHIPMENT Y
WHERE X.PID = Y.PID AND
      X.MGR = 'ROBERT'
EXPAND BY MONTH
GROUP BY X.PID, *.FROM.
```

Note that granularity of SHIPMENT and PROJECT are different. The time granularity of their concurrent product will be DATE, the finer of the two. Moreover, since SHIPMENT is an event relation, the concurrent product is also an event relation. *.FROM refers to the corresponding time attribute in the result.

Note: A concurrent product of two historical relations $R1$ and $R2$ can also be obtained as follows:

```
SELECT R1.ALL, R2.ALL, R1.INTERVAL*
  R2. INTERVAL
FROM R1, R2.
```

Thus, using * and +, it is possible to define what have been called in literature as concurrent and union joins of two historical relations.

C. Data Manipulation

In this section, we consider updates to historical relations performed at real-world times. We call these "normal" updates; here, the transaction time, supplied automatically by HDBMS, is the same as the real-world time.

The updates include insertion, deletion, and change operations. The data manipulation statements of HSQL are the *same* as SQL. Their execution by HDBMS is, however, different in that HDBMS manipulates time and enforces the integrity constraints. The normal update operations were illustrated by an example in Section II, indicating how tuples are added to the current segment and how history tuples are generated.

D. Retrospective Updates

It may occasionally be necessary to update data in a historical relation with retrospective effect. We expect that this need would be rare, because real-world implications of such updates are complex. For instance, if the price of an item is changed retrospectively, we have to decide the course of action for orders already completed.

In general, a retrospective update for state-oriented historical relation may affect only the current state of an object, one or more history states of an object, or current as well as one or more history states of an object. For event-oriented relations, there is one tuple per event. These may also be changed retrospectively. We provide different syntax for these two types of historical relations.

State-Oriented Relations: The syntax for retroactive update of a state relation is

FROMTIME $t1$ [TOTIME $t2$]
Update-statement | insert-statement | delete-statement

By default, time-2 is taken as NOW.

To explain the semantics of retrospective manipulation, we first define two interval operations. Let $p1$ and $p2$ be two overlapping intervals. Then,

$p3 = p1$ *before* $p2$

is that portion of $p1$ which precedes $p2$, and

$p4 = p1$ *after* $p2$

is that portion of $p1$ which follows $p2$. Both $p3$ and $p4$ would be null if $p1$ and $p2$ do not overlap.

The retrospective update statement may affect tuples in both current and history segments. Let $<s, t3, t4>$ be a tuple (in current if $t4$ = NOW, else in history) whose *visible* attributes are to be changed to s'. HDBMS replaces the tuple by the following if $t3 \ldots t4$ overlaps with $t1 \ldots t2$ (given in FROMTIME):

$<s$, (t3 . . t4) *before* $(t1 \ldots t2)>$
$<s$, (t3 . . t4) *after* $(t1 \ldots t2)>$
$<s'$,t1 . . t2> .

Recall the integrity constraint that tuples with null intervals are not stored at all. All tuples modified in this fashion are coalesced and stored in appropriate segments.

The retrospective delete statement may affect zero or more tuples in either segment. If $<s, t3, t4>$ is one such tuple, HDBMS replaces it by the following if $t3 \ldots t4$ overlaps with $t1 \ldots t2$:

$<s$, $(t3 \ldots t4)$ *before* $(t1 \ldots t2)>$
$<s$, $(t3 \ldots t4)$ *after* $(t1 \ldots t2)>$.

The retrospective insert statement may add zero or more tuples in either segment. Let s be a tuple to be added. HDBMS then considers $<s, t1, t2>$ for insertion. It first checks the integrity requirement that no states in this interval already exist for the involved key; i.e., if k is the key value in s, then the historical relation (in either segment) must not contain a tuple with k and interval that overlaps $t1 \ldots t2$.

Event-Oriented Relations: For retrospective updates to event-oriented historical relations, the data manipulation statement could be

delete-statement

with same syntax as in SQL, or

AT time
insert-statement | update-statement.

The delete statement may delete zero or more tuples. The time given in the AT clause is used as event time for tuples being inserted or existing tuples being updated.

Updates without Time Change: It may be necessary at times (e.g., for error correction) to change visible attribute values while keeping values of time attributes the same as before. To facilitate this, the time specification in the FROMTIME, TOTIME, and AT clauses could be simply given as "?." For instance, the specification

FROMTIME ? TOTIME ?
update-statement

would only change visible attributes (indicated by SET clause of the UPDATE statement) of the selected tuple.

Examples: The following retrospective data manipulation examples are with respect to the historical database defined in Fig. 1.

1) All the laboratories in Boston were moved to Cambridge from Jan. 1986:

```
FROMTIME 1986:01
UPDATE PROJECT
  SET LOC = 'CAMBRIDGE'
  WHERE LOC = 'BOSTON'
```

2) Jane was manager of the project OS5 from March 1984 to August 1984:

```
FROMTIME 1984:03 TOTIME 1984:08
UPDATE PROJECT
  SET MGR = 'JANE'
  WHERE PID = 'OS5'
```

Assume that PROJECT contains the following two tuples for OS5 with intervals overlapping the intervals in the above UPDATE:

<OS5, SOFTWARE, BOSTON, DICK, 1983:04, 1984:05>
<OS5, SOFTWARE, BOSTON, HARRY, 1984:06, 1985:09>

From the first, HDBMS produces

<OS5, SOFTWARE, BOSTON, DICK, 1983:04, 1984:02>
<OS5, SOFTWARE, BOSTON, JANE, 1984:03, 1984:05>

and the following from the second:

<OS5, SOFTWARE, BOSTON, JANE, 1984:06, 1984:08>
<OS5, SOFTWARE, BOSTON, HARRY, 1984:09, 1985:09> .

Coalescing these four produces

<OS5, SOFTWARE, BOSTON, DICK, 1983:04, 1984:02>
<OS5, SOFTWARE, BOSTON, JANE, 1984:03, 1984:08>
<OS5, SOFTWARE, BOSTON, HARRY, 1984:09, 1984:05> .

These three tuples will replace the original two tuples affected by this statement.

3) There was a shipment for project ADA involving part HW125 in quantity 3 on Dec. 3, 1984:

```
AT 1984:12:03
INSERT INTO SHIPMENT
  VALUES ('ADA', 'HW125', 3, NULL).
```

This statement adds the following tuple to SHIPMENT:

<ADA, HW125, 3, NULL, 1984:12:03, 1984:12:03>.

E. Using Nonhistorical Relations in HSQL

A nonhistorical relation R is the conventional relation of the standard relational data model. It has neither history nor timing information in its tuples. However, it may be used together with historical relations in a database. It then becomes important to define its meaning in the context of time. There are two possible interpretations for a nonhistorical relation:

1) *as a snapshot relation:* a nonhistorical relation may be interpreted as containing data that are valid only at NOW. Taking a time-slice of such a relation at $t <$ NOW produces an empty relation. Also, taking concurrent product of the snapshot relation R with a historical relation S (having history and current segments Sh and Sc) gives us

$$R\ Xt\ S = R\ Xt\ Sc$$
$$= R\ X\ (\tau_{NOW..NOW}(Sc))$$

Note that the result tuples will contain intervals of one instant NOW. Thus, timing data contained in the historical relation hardly play any role when used together with a snapshot relation.

2) *as a constant relation:* a nonhistorical relation may be interpreted as a state type relation whose tuples are effective from $-\infty$ to NOW. Such a relation might contain data which are truly time-invariant (e.g., the constant data about books containing ISBN, title, authors, and publisher). Alternatively, such a relation might represent objects whose history is of no interest (e.g., data about suppliers containing name and address, with no need to maintain past addresses). If R is such a relation (and S, as before, is a historical relation), then

$$\tau_p(R) = R$$
$$R\ Xt\ s = R\ X\ S.$$

Note that Cartesian product and concurrent product give the same result in this interpretation. In fact, the result retains the time data of the historical relation.

HSQL, by default, treats a nonhistorical relation R as a constant relation. It can be directly used in FROM [CONCURRENT] with this interpretation. To obtain the first interpretation, one can take a time-slice of historical relations at NOW and use them with R in an HSQL query. Thus, both interpretations are possible in HSQL.

F. Facilities for Time Rollback

As seen earlier, HDBMS provides facilities both for normal and retrospective updates. We refer to retrospective updates as "correction" transactions. These transactions may modify both current and history segments. Once processed, their effect is permanent on the database. Thus, if we wish to take a time-slice after a correction has been applied, the time-slicing is performed on the modified database.

A *time-rollback* (simply, *rollback*) differs from time-slice in a very important way. A rollback to time t refers to the state of database as it was known at time t. Thus, here, we must disregard (i.e., undo) updates and correction transactions processed after time t. The time rollback was proposed by Snodgrass [14], [15]. It may, however, be observed that its use would probably be very rare in real-world applications.

To be able to apply time-rollback on a historical relation R, we must include the clause

WITH ROLLBACK FACILITY

in the DEFINE TABLE statement for relation R. HDBMS then sets up a "correction log" for R to store all relevant data about corrections applied to R. These data include not only the time at which the correction takes effect, but also the time when the correction was made. The latter is used for deciding whether the correction needs to be undone during rollback. Algorithms for time-rollback using logs are given in [11].

The HSQL statement for performing rollback is

```
ROLLBACK table-name
AS OF time-instant.
```

The result of a rollback is a historical relation, which may be saved and used in further operations.

VI. Conclusions

Modeling time and reasoning about it has been of concern to philosophers, logicians, linguists, and, recently, to computer scientists working in the areas of artificial intelligence [12] and information processing [2]. Logicians have proposed "temporal logic" as a tool to capture the meaning of temporal statements. Alternative considerations for the following issues have led to many different solutions [12]:

- discrete or continuous, linear or branching time,
- interpretation over time points or time intervals,
- transport of truth over an interval to its subinterval, and
- logical form of associating time with an assertion.

Shoham [12] has proposed interval-based temporal logic and has defined propositional and first-order versions of it. It is based on discrete time, assertions over intervals defined by an ordered pair of points, and "<" (i.e., before) relationship between points. Shoham shows that his model can capture objects proposed by others, such as properties, events, and processes by categorizing temporal propositions.

The historical data model of this paper uses the same foundation as Shoham's temporal logic. Hence, an HSQL query can be interpreted as a formula in the first-order

interval predicate calculus. Also, the proposed HDBMS and HSQL have many unique and innovative features. In the following, we emphasize these features and compare them to other proposals in the literature.

1) We consider our state- and event-oriented view of the real world more natural than the "cubic" view (of [1], [4], [15]), as the cubic view is too "regular" to model different database objects that are added, updated, and deleted at different times.

2) The state-oriented view naturally leads to interval-stamping of tuples. The database designer can use the theory of normalization [18] and also apply time-normalization [7] to arrive at "minimal objects" that do not exhibit any anomalies.

3) Since HDBMS maintains time and history data, the database designer can use the current perspective of user requirements during logical schema design. Consequently, the database schema is simpler than when history is explicitly included in schema design.

4) HDBMS supports only the real-world time. In a real-time environment, on most occasions, the transaction time would be equal to real-world time. HDBMS provides facilities for (occasional) retrospective updates. There are two advantages to our approach: storage-efficiency, since system times are not stored in every tuple, and efficient query processing as we do not need a default operation ("As of NOW" as in [15]).

5) HSQL retains the simple structural and algebraic framework of SQL. The extensions are well-defined in that they relate directly to specific algebra operations, which makes HSQL queries amenable to efficient translation, optimization, and execution [10]. HSQL provides a facility for directly joining concurrent data from different relations. It also adequately addresses the issue of time granularities.

We have made some progress with an experimental implementation of the proposed HDBMS and HSQL. The HSQL run-time system, which can interpret a query program consisting of historical and relational algebra operations (as also arithmetic and logical operations involved in predicates), has been designed and implemented. The HSQL query processor, consisting of query-tree generator, optimizer, and query-program generator as its major components, is under implementation. We have studied the algebraic properties of new operators for their use as query optimizing transformations [10]. A practical HDBMS will have to address the efficiency issue more thoroughly than our experimental implementation. For example, the issue of efficient storage structures for handling monotonically increasing history needs further research.

References

[1] G. Ariav, "A temporal oriented data model," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 499-527, Dec. 1986.

[2] A. Bolour, T. L. Anderson, L. J. Dekeyser, and H. K. T. Wong, "The role of time in information processing: A survey," *SIGMOD Rec.*, vol. 12, no. 3, pp. 27-50, Apr. 1982.

[3] D. D. Chamberlain *et al.*, "SEQUEL 2: A unified approach to data definition, manipulation and control," *IBM J. Res. Develop.*, vol. 20, no. 6, pp. 560-575, 1976.

[4] J. Clifford and D. S. Warren, "Formal semantics for time in databases," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 214-254, 1983.

[5] J. Clifford aand A. V. Tansel, "On an algebra for historical relational databases: Two views," in *Proc. ACM SIGMOD*, 1985, pp. 247-265.

[6] C. J. Date, *A Guide to the SQL Standard.* Reading, MA: Addison-Wesley, 1987.

[7] S. B. Navathe and R. Ahmed, "A temporal relational model and query language," *Int. J. Inform. Sci.*, Sept. 1988.

[8] N. L. Sarda, "Modelling of time and history data in database systems," in *Proc. CIPS Congress '87*, May 1987, pp. 15-20.

[9] ——, "Algebra and query language for a historical data model," *Comput. J.*, to be published.

[10] ——, "Design of a historical database management system," *Comput. J.*, submitted for publication.

[11] —— "An algorithm for time-rollback on historical relations," Tech. Rep., May 1988.

[12] Y. Shoham, *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence.* Cambridge, MA: MIT Press, 1988.

[13] R. Snodgrass, Ed., "Research concerning time in databases: Project summaries," *SIGMOD Rec.*, vol. 15, no. 4, pp. 19-39, Dec. 1986.

[14] R. Snodgrass and I. Ahn, "Temporal databases," *IEEE Comput. Mag.*, pp. 35-42, Sept. 1986.

[15] R. Snodgrass, "The temporal query language TQuel," *ACM Trans. Database Syst.*, vol. 12, no. 2, pp. 247-298, June 1987.

[16] R. Snodgrass, S. Gomez, and E. McKenzie, "Aggregates in the temporal query language TQuel," TEMPIS Document 16, Univ. of North Carolina, July 27, 1987.

[17] E. McKenzie and R. Snodgrass, "An evaluation of historical algebras," TR87-020, Univ. of North Carolina, Oct. 1987.

[18] J. D. Ullman, *Principles of Database Systems.* Rockville, MD: Computer Science Press, 1984.

[19] S. T. March and D. G. Severance, "Determination of efficient record segmentation and blocking factors for shared data files," *ACM Trans. Database Syst.*, vol. 2, no. 2, pp. 279-296, 1977.

[20] C. J. Date, "Defining data types in a database language (Alternative Title: A proposal for adding date and time support to SQL)," *SIGMOD Rec.*, vol. 17, no. 2, pp. 53-76, June 1988.

Nandlal L. Sarda received the B.E. degree in electrical engineering from Nagpur University, and the M.Tech. and Ph.D. degrees in computer science from the Indian Institute of Technology, Bombay.

Since 1972, he has been on the faculty of the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, where, at present, he holds the position of Associate Professor. He also worked as Associate Professor of Computer Science at the University of New Brunswick, Saint John, Canada, from September 1986 to October 1988. His main research interests are in the areas of database systems and programming languages. He is presently coordinating the Government of India sponsored project on Informatics at IIT Bombay with the twin objectives of manpower training, and research and development in various aspects of Informatics.