History and Tuple Variables
for Temporal Query Languages

Fabio Grandi    Maria Rita Scalas    Paolo Tiberio
C.I.O.C.-C.N.R. and Dipartimento di Elettronica Informatica e Sistemistica
Università di Bologna, Viale Risorgimento 2, I-40136 Bologna, Italy
E-mail: \{fgrandi,mrscalas,ptiberio\}@deis.unibo.it

Abstract

Standard relational query languages provide range variables whose values are tuples and which are used to denote objects (entities or relationships). Snapshot relational databases represent an object by means of a tuple, whereas temporal relational databases represent an object by means of a collection of time-stamped tuples with the same key value. Temporal extensions of query languages proposed so far are also provided with range variables whose values are still tuples and so can no longer denote objects, but time-stamped object versions.

In this paper we denote as history the set of all the tuples with the same key value in a relation and propose the introduction of two new kinds of range variables: history variables which have histories as values and denote objects and tuple variables which have tuples as values and denote object versions. We also illustrate temporal extensions to SQL with these two types of variables. Some examples show how history and tuple variables can improve clarity and ease of use of a temporal query language. Therefore we propose that the two types of variables should be part of a common infrastructure for a temporal SQL extension.

1 Introduction

In the relational model, a database is a collection of relations and relations are collections of tuples with the same attributes. If the term object is used to denote an entity or a relationship (as defined in the Entity-Relationship model [2]), then tuples represent objects and relations represent classes of objects.

Non-temporal relational languages SQL [3] and Quel [13] provide range variables to express queries. The tuples in a relation are one-to-one related with the objects of the class the relation represents and the key value in a tuple plays the role of identifier of the corresponding object. Therefore, we can acknowledge the following two properties of a range variable:

• \( P_1 \): From a semantic point of view, at any given time, the range variable denotes some object of the class the relation represents.

• \( P_2 \): From a functional point of view, at any given time, the value of the range variable is some tuple of the relation over which the variable ranges.
The property \( (P_1) \) is very important from an application point of view, since it provides a query language with a simple and powerful tool to directly denote instances of entities or relationships as defined in conceptual database design \([2]\).

In a temporal extension of the relational model, the two properties of the range variables cannot be both maintained, since the one-to-one correspondence between objects and tuples is lost. As a matter of fact, any object is represented in a temporal relation by the set of all the tuples having a common key value, which still acts as a time-invariant object identifier. In \([6, 7]\) we named \textit{history} such a set, since it represents the whole history of an object as it has been stored in the database. Any tuple has a definite time-pertinence (timestamp) and belongs uniquely to one history. Thus, tuples represent object versions, within an object history. We assume no tuples overlap (totally or partially) in time within the same history (\textit{history uniqueness}). This integrity constraint substitutes the \textit{key uniqueness} valid for snapshot relations.

In all the temporal languages proposed so far by other authors (e.g., TOSQL \([1]\), HTQuel \([5]\), TSQL \([9]\), HSQL \([10]\), TDM \([11]\), TQuel \([12]\), HQUEL \([14]\)), the values that range variables can assume are still tuples (property \( (P_2) \) maintained), since these languages do not extend SQL or Quel in this respect. If we also require some linguistic tool to be still available in the query language in order to denote objects in a temporal database (property \( (P_1) \) maintained), it follows that another kind of variables must be introduced such that their values are no longer single tuples but complete histories. This is why we propose to introduce in a temporal language two kinds of range variables:

- **history variables**, whose values are single histories in a relation;
- **tuple variables**, whose values are single tuples in a history.

From a semantic point of view, a history variable denotes an object whereas a tuple variable denotes a time-determined object version (see Fig. 1).

Experience demonstrates that there are two main goals for a database user formulating a query: the selection of objects and the retrieval of data concerning them. With a snapshot database, the former goal consists of tuple selection, whereas the latter is reduced to the choice of the target list for projection. With a temporal database, we would like the user to be allowed to pursue the same two goals. The former goal now consists of history rather than tuple selection, whereas the latter also includes the choice of a temporal span, which defines the time pertinence of the attribute values to be retrieved.

We name \textit{history-oriented} a temporal query language which allows these two goals to be preserved and provides history and tuple variables to this end. Several examples will be presented in order to clarify usage of history and tuple variables in a history-oriented temporal language.

In \([6, 7]\) we presented the complete definition (syntax and semantics) of the History-oriented Temporal Query Language HoTQuel. HoTQuel is an extension of Quel provided with history and tuple variables. In this work, we propose the availability of history and tuple variables as a general feature of temporal query languages. We show how their definition and usage can be added to an SQL temporal extension.

In the rest of the paper we refer to bitemporal relations \([8]\), which include valid and transaction time, with interval time-stamping of the tuples. We assume snapshot relations to have been augmented with four attributes, \textit{From}, \textit{To}, \textit{In}, \textit{Out}, representing the endpoints of the valid- and transaction-time intervals.
2 A History-oriented Temporal SQL Extension

In this Section we show how history and tuple variables can be embedded in a temporal extension of SQL. We also show how their use can provide the language with history-oriented features which are introduced in the following.

When dealing with a snapshot relational database, two main goals can be distinguished in data retrieval:

- \( G_1 \): identification of the objects of interest;
- \( G_2 \): retrieval of attribute values of the selected objects.

Notice that all the object attributes functionally depend on its key value.

In a temporal framework, since the attribute values also depend on time, \( G_2 \) also requires time coordinates to be specified. These allow the selection of the versions within the object histories from which the attribute values must be retrieved. The determination of the time coordinates can be included in \( G_1 \). Therefore, we can rewrite the two goals as follows:

- \( G'_1 \): identification of the objects of interest and specification of time spans;
- \( G'_2 \): retrieval of attribute values of the selected objects corresponding to the selected time spans.

We introduce now the history-oriented selection concept which formalizes the two goals of data retrieval discussed above. In the standard relational model, the values of the attributes of an object can be retrieved in a tuple, which can be referred to by its unique key value within the relation it belongs to. In a bitemporal relational model, the equivalent operation requires an
object identifier (key value), a transaction and a valid time span. From a logical point of view, the
data selection consists of the following phases [6]:

- \( S_{1,a} \) — **History Selection**: Selection of the histories of the objects of interest, according
to a selection predicate; (Output = a collection of key values);

- \( S_{1,b} \) — **Time Selection**: Selection of the transaction and valid time spans in which we want
to retrieve the data of interest, according to two time predicates;
  (Output = a collection of transaction- and valid-time intervals);

- \( S_2 \) — **Data Selection**: Selection of the tuples where the time intervals retrieved in phase
  \( S_{1,b} \) intersect the histories retrieved in phase \( S_{1,a} \), and projection of the selected tuples
  on the target list; (Output = a collection of attribute values).

In other words, during Phase \( S_2 \), the histories selected in Phase \( S_{1,a} \) are restricted over time
(within the temporal elements retrieved in Phase \( S_{1,b} \)) and projected onto a target list.

The history-oriented selection is the basic selection mechanism of the temporal SQL extension
we uniformly introduce in this paper. The general form of its **SELECT** statement is:

```
SELECT  target list  
FROM history and tuple variable declarations
WHERE  restriction predicates (on history and tuple variable attributes)
valid-time specification clause
transaction-time specification clause
```

The **target list** contains variable names denoting the histories to be selected (results of Phase \( S_{1,a} \))
and attribute names used for projection in Phase \( S_2 \). Also the keywords **VALIDITY** and **ACTUALITY**
can be included in the **target list** of queries retrieving the selected times (results of Phase \( S_{1,b} \)).
The **FROM** clause is devoted to the variable declarations, and allows the declaration of history
variables over relations and of tuple variables over history variables. The **WHERE** clause contains
all the predicates that are needed for history \( S_{1,a} \) and time \( S_{1,b} \) selection. Such predicates
involve temporal and non-temporal restrictions on the attributes of history and tuple variables.
The last two clauses are used to specify the results of valid- and transaction-time selection Phase
\( S_{1,b} \).

The variable declarations and the selection phases are exemplified in detail in the next subsections.

### 2.1 History and Tuple Variables

#### 2.1.1 Explicit declarations

In a snapshot relational language, a range variable ranges over a relation and the values it can
assume are tuples. Range variables can be optionally declared in the **FROM** clause of SQL’s **SELECT**
statement. The general syntax of the **FROM** clause is:

```
FROM Rel1 RangeVar1, Rel2 RangeVar2, ...
```
When a range variable name is omitted, a variable with the same name of the relation is implicitly assumed [4, Sec. 11.3.1]. In this case, relation names can directly be used as variables in the target list and in the WHERE clause of a statement. Also in Quel, in the absence of an explicit declaration, the language processor assumes an implicit range variable with the same name of the relation [4, Sec. 4.3.8]. However, explicit declaration of range variables is usually effected in Quel by means of the \texttt{RANGE} statement. SQL's range variables are local to the statement in which they are declared, whereas the scope is global in Quel; the binding between variable and relation is valid until the variable is redefined in another \texttt{RANGE} statement.

The SQL temporal extension we propose maintains the same syntax of the non-temporal range variable declarations for the \textit{history variable} declarations:

\begin{verbatim}
FROM Rel1 HistoryVar1, Rel2 HistoryVar2, ...
\end{verbatim}

History variables denote objects as the non-temporal range variables do, but have histories (sets of tuples with a common key value) as permitted values. As for SQL's range variables, history variable declarations are fully optional: if a history variable name is omitted then an implicit history variable with the same name of the relation is assumed.

In a temporal framework, a tuple variable ranges over a history and the values it can assume are versions (tuples within the same history). For instance, tuple variables could be declared in a temporal SQL extension as follows:

\begin{verbatim}
FROM Rel1 HistoryVar1: TupleVar11 ... TupleVar1m, (D_1)
  Rel2 HistoryVar2: TupleVar21 ... TupleVar2n,
  ...
\end{verbatim}

In this example, the tuple variables \texttt{TupleVar11..TupleVar1m} are declared over the history variable \texttt{HistoryVar1} to denote some versions of the object denoted by \texttt{HistoryVar1}. The tuple variables \texttt{TupleVar21..TupleVar2n} are declared over the history variable \texttt{HistoryVar2} to denote some versions of the object denoted by \texttt{HistoryVar2}. However, also alternative syntactic forms can be considered in order to make the declarations more readable. For instance, a different choice could be given by:

\begin{verbatim}
FROM Rel1 HistoryVar1 [TupleVar11, ..., TupleVar1m], (D_2)
  Rel2 HistoryVar2 [TupleVar21, ..., TupleVar2n],
  ...
\end{verbatim}

that is with tuple variable lists enclosed in square brackets (round brackets could be confusing with other SQL constructs and other kinds of brackets are not easily found on any terminal keyboard). In this case, commas can be used to separate variable names within the brackets. Anyway, the choice of a “standard” syntax for history and tuple variable declarations is a point fully amenable to consensus. In the rest of the paper, we adhere to the declaration syntax $D_1$.

\subsection{2.1.2 Implicit declarations}

Syntactic and semantic conventions on implicit variable declarations may also be adopted in order to make the temporal extension downward compatible with SQL as much as possible. For instance, we may also consider the colon in ($D_1$) as optional. If the colon is omitted, the relation name can be used as a history variable name and all the declared variables can be assumed to be tuple variables. In other words, the FROM clause:

\begin{verbatim}
FROM Rel1 HistoryVar1, Rel2 HistoryVar2, ...
\end{verbatim}
FROM Rel1 Var1 Var2, Rel2 Var3 Var4 Var5
can be interpreted as:

FROM Rel1 Rel1: Var1 Var2, Rel2 Rel2: Var3 Var4 Var5

Another possibility is to consider the first declared variable as a history variable and the others as tuple variables. This solution may be necessary when a relation name occurs more than once in the FROM clause (i.e. different objects must be referenced), as in:

FROM Rel1 Var1 Var2, Rel1 Var3 Var4 Var5

which can be interpreted as:

FROM Rel1 Var1: Var2, Rel1 Var3: Var4 Var5

Moreover, if only one variable name per relation is declared, that name can assumed to be the name of a history variable declared over the relation and it can also be assumed to be the name of one tuple variable. For instance, in the following clause:

FROM Rel1 Var1, Rel2

implicit declarations of a history variable and of a tuple variable, with the same name of the relation, can be assumed as if the clause were:

FROM Rel1 Var1: Var1, Rel2 Rel2: Rel2

Anyway, our proposal is to let the declaration format as free as possible, even if a regular use of the colon greatly improves readability of any query. The exact role of a declared variable should be in any case disambiguated by means of its use within the query. Only history variable names may appear in the target list (SELECT clause) of a query (see the next Section). History and tuple variables do not share attributes but those composing the key, which do not give rise to denotational ambiguity (key attribute values are repeated in every tuple of a history). Therefore, even if in the last example we have history and tuple variables with the same name (i.e. Var1 and Rel2), it can be evicted from the context when the name of a variable within the query indicates the history or the tuple variable. For instance, Var1 is used as a tuple variable name in Var1.VALIDITY and as a history variable name in Var1.HVALIDITY¹. Moreover, possible conflicts and unresolved ambiguities may be reported to the user by the query parser.

2.1.3 History and tuple attributes

In non-temporal query languages, range variables are used to express selection conditions on data by means of constraints on attribute values. Atomic attributes of objects denoted by range variables can be referenced in SQL as RangeVar.Attribute, where RangeVar is the variable name and Attribute is the attribute name.

In a temporal database, atomic attributes of an object version can be defined — and referenced in the queries as TupleVar.Attribute — as the attribute value in the tuples binding, one at a time, to the tuple variable denoting the object version.

¹Incidentally, this is the reason why we introduced the HVALIDITY and HACTUALITY keywords.
Atomic attributes of histories can be defined — and referenced in the queries as $\text{HistoryVar.Attribute}$ — as *global properties* of a history. According to its definition, the key is the only proper attribute of a history, thus, the key value can naturally be used as a history property. Also parts of aggregate keys — which are uniquely defined in a history though repeated in different histories — can be used as history properties. This allows *temporal join conditions involving foreign key matching* to be conveniently expressed between history attributes. Also time-invariant attributes can be used as history properties, since their value is constant in every tuple belonging to the history\(^2\). Moreover, since histories in a relation can be considered as tuples *grouped by* the key value (e.g., in SQL sense), also aggregate values could be used as history properties: the number of stored versions (computed as $\text{COUNT()}$ of the tuples), the whole lifespans of the denoted object (e.g., computed from the tuples as $[\text{MIN(From)}, \text{MAX(To)}]$ for valid time and as $[\text{MIN(In)}, \text{MAX(Out)}]$ for transaction time), the average value of some attribute and so on.

Moreover, the keywords $\text{VALIDITY}$ and $\text{ACTUALITY}$ can be used as selectors to denote, respectively, the valid- and transaction-time interval of a tuple variable in the form: $\text{TupleVar.VALIDITY}$ and $\text{TupleVar.ACTUALITY}$, that is as if they were atomic attribute names. Similarly, we can introduce the the keywords $\text{HVALIDITY}$ and $\text{HACTUALITY}$ to be used as selectors to denote, respectively, the valid- and transaction-time lifespans of a history variable in the form: $\text{HistoryVar.HVALIDITY}$ and $\text{HistoryVar.HACTUALITY}$. All these selectors allow interval-type expressions to be derived from the timestamps of object versions and from the lifespans of objects.

Obviously, if *more than one* tuple variable per history must be used to denote *different* versions of the same object and/or *more than one* history variable per relation must be used to denote *different* objects, explicit declarations cannot be omitted.

### 2.2 History selection

As in non-temporal languages, the *WHERE* clause and the variable definitions should be the linguistic tools devoted to the selection of objects. Conceptually, history selection can be effected in two ways:

- by means of some global property of the history itself;
- by means of some property of its component versions.

*A history-oriented temporal language should support global history properties as defined in Sec. 2.1.* Let us focus on a sample relation with schema $\text{Employee(Name, Job, Salary, From, To, In, Out)}$ where $\text{Name}$ is the primary key. Consider, for instance, the object selection implied by the clauses:

- $C_1$: the employee whose name is John (if the name is a unique key)
- $C_2$: the employees who entered the enterprise after 1/1/1985 (test on the lifespan)

*With a history-oriented temporal language, history variable attributes can be used to this purpose.* Clause $(C_1)$ can be expressed in the SQL extension as:

\(^2\)This is rigorously true only with respect to valid-time; different values along transaction-time can be found if error corrections took place. This may happen also for keys, if key changes are allowed.
SELECT *
FROM Employee Emp
WHERE Emp.Name="John"

whereas clause \((C_2)\) can be expressed as in:

```
SELECT *
FROM Employee
WHERE BEGINOF( Employee.HVALIDITY ) > 1/1/1985
```

In both examples, simple conditions on history attributes are tested and used to directly select objects. It can be noticed that the above queries retrieve the complete histories of the selected objects. In the second example, no history variables have been explicitly declared, and the relation name is assumed as an implicit history variable name.

Selection by properties of the component versions is probably the most common and the most useful in querying temporal data. Objects are often selected through conditions involving parts of their history (e.g., their present state). Furthermore, relationships between different historical states of the same object, or even of different objects, can be tested for the same purpose. Such relationships between object versions can be temporal or involve non-temporal attribute values. Examples are provided by the clauses:

- \(C_3\): the employees who were director in 1991
- \(C_4\): the employees who were engineers before becoming managers
- \(C_5\): the employees whose salary has increased by more than $5K since 1988
- \(C_6\): the employees whose job has changed but whose salary has not
- \(C_7\): the employees whose salary was $25K while Ann was director
- \(C_8\): the employees who ever earned more than $50K

*With a history-oriented temporal language, tuple variable attributes can be used to this purpose.* Clause \((C_3)\) involves a simple condition on the value of the attribute Job of the version valid in 1991 and can be expressed as:

```
SELECT *
FROM Employee: Who
WHERE Who IS VALID IN 1991
    AND Who.Job = "Director"
```

where two conditions are imposed to the tuple variable \(\text{Who}\): its Job attribute value must be Director and its valid-time interval must (totally or partially) overlap\(^3\) 1991. When one or

\(^3\)In this work we adopt the “overlaps” semantics defined for TQuel, that is \([a, b] \text{ OVERLAPS } [c, d] \iff a \leq d \land c \leq b\), which is close to the English meaning but differs from the homonym defined in Allen’s logic.
more versions of the same employee are valid in 1991, the employee is selected if at least one of
these versions has Job = "Director". In this example, also the tuple variable declaration can be
omitted, since only one object version is referred to and, thus, the relation name can be implicitly
used again:

SELECT *
FROM Employee
WHERE Employee IS VALID IN 1991
    AND Employee.Job = "Director"

Clause (C₄) involves a temporal relationship, “precedes”, between two versions of the same
object and can be easily expressed in the following query:

SELECT *
FROM Employee: JobEng JobMgr
WHERE JobEng.Job = "Engineer"
    AND JobMgr.Job = "Manager"
    AND JobEng.VALIDITY PRECEDES JobMgr.VALIDITY

Two tuple variables, JobEng and JobMgr, have been declared to range over a common history,
bound to the history variable Employee implicitly declared. Non-temporal local restrictions are
imposed to the tuple variables, whereas a temporal mutual restriction constraint their valid times.

Clause (C₅) is very similar to (C₄), but the role of the temporal and attribute value relationships
has been inverted, since local restrictions on tuple variable are temporal, whereas the mutual
restriction is non-temporal. It can be expressed as in the query:

SELECT *
FROM Employee: Emp88 EmpNow
WHERE Emp88 IS VALID IN 1988
    AND EmpNow IS VALID NOW
    AND EmpNow.Salary - Emp88.Salary > 5000

The two versions Emp88 and EmpNow are selected within the a common history by means of defined
by a simple condition concerning their validity and an additional attribute value relationship
between them is tested.

Clause (C₆) includes temporal and attribute value relationships between versions and can be
expressed as:

SELECT *
FROM Employee: EmpVer1 EmpVer2
WHERE EmpVer1.VALIDITY MEETS EmpVer2.VALIDITY
    AND EmpVer1.Job <> EmpVer2.Job
    AND EmpVer1.Salary = EmpVer2.Salary
The only constraint imposed by the definition of \texttt{EmpVer1} and \texttt{EmpVer2} is the fact that they belong to the same history (both are declared over the implicit variable \texttt{Employee}), which is coherent with the fact that they represent different versions of the same object.

Clause (\(C_7\)) involves properties of an independent object to select the objects of interest:

\begin{verbatim}
SELECT *
FROM Employee Emp1: Who, Employee Emp2: ThePeriod
WHERE Who.Salary = 25000
  AND Emp2.Name = "Ann"
  AND ThePeriod.Job = "Director"
  AND Who.VALIDITY IS DURING ThePeriod.VALIDITY
\end{verbatim}

In this case two history variables, \texttt{Emp1} and \texttt{Emp2}, are used to denote different objects. The tuple variables, \texttt{Who} and \texttt{ThePeriod}, respectively denote a version of these objects.

Clause (\(C_8\)) imposes a constraint on the salary value of a version of the objects to be selected. It can be simply expressed (using implicit declarations) as:

\begin{verbatim}
SELECT *
FROM Employee
WHERE Employee.Salary > 50000
\end{verbatim}

It can be noticed that this query retrieves all the data (complete histories) of all the employees who have at least one version with a salary value greater than $50K. In a “tuple-oriented” (vs history-oriented) temporal query language, only the qualifying versions (tuples with \texttt{Salary} > 50000) are retrieved. Moreover, this is fully consistent with the standard SQL, because in a snapshot database the histories are composed of exactly one tuple and, thus, the query above retrieves all the tuples with \texttt{Salary} > 50000.

\subsection{2.3 Time selection}

Time selection concerned in phase \((S_{1,b})\) of history-oriented data retrieval can also be effected in two ways:

- directly, by means of expressions containing only time constants;
- indirectly, by means of expressions containing references to the time-pertinence of some data.

In the second case, with a history-oriented temporal language, the time pertinence of history tuple variables can be used. The former pertinence represent an object lifespan and the latter an object version timestamp. For example, valid- and transaction-time intervals of any version of any object (not necessarily the same objects selected in phase \((S_{1,a})\)) can be used. Thus, time spans can be computed via expressions containing timestamps of object versions which can be determined through temporal or attribute-value relationships.

The final results of time selection must be specified in the queries in order to be used by the query processor during selection phase \((S_2)\). In a bitemporal language, two dedicated clauses must be used whereas only one is needed in a system supporting only one time dimension. The syntax of such clauses strictly depends on the particular language considered. The semantics of such clauses

OO-10
corresponds to the specification of time spans which are used to perform a final time-slice operation on data. Therefore, the time-slice operators already proposed for other temporal languages can be used in a history-oriented language for the specification of the results of time selection. Without loss of generality, we adopt for the SQL extension proposed in this work the same syntax we used for HoTQuel, which has a VALID and an ACTUAL clause for valid and transaction time selection:

```
SELECT ...
...
VALID time-specification
ACTUAL time-specification
```

Each `time-specification` may contain interval or time-point expressions in the form:

```
IN time-interval-expr
```

or

```
FROM time-point-expr TO time-point-expr
```

or

```
AT | SINCE | UNTIL time-point-expr
```

When one or both time clauses are omitted, default specifications (e.g. VALID ALWAYS, ACTUAL NOW) are assumed.

Let us now consider some examples. Time selection from the data concerning objects of the class “Employee” can be expressed via the clauses:

- \( C_9 \): the period in which the employee Ann was an engineer
- \( C_{10} \): the period in which the employees who had been engineers were managers
- \( C_{11} \): the periods in which the employees who ever earned \( \$ 40 \)K were present in the database (from their insertion to their deletion)

As in object selection \( (S_{1a}) \), the declarations of history variables and the WHERE clause are the linguistic tools devoted also to time selection.

Clause \( (C_9) \) can be expressed as in the query:

```
SELECT VALIDITY
FROM Employee Emp: Period
WHERE Emp.Name = "Ann"
    AND Period.Job = "Engineer"
VALID IN Period.VALIDITY
```

if the user knows that Name is the key attribute (or a unique history attribute); otherwise it can be expressed as:

```
SELECT VALIDITY
FROM Employee: Period
WHERE Period.Name = "Ann"
    AND Period.Job = "Engineer"
VALID IN Period.VALIDITY
```
In both cases, the selected timespan (one interval at a time), denoted as Period.VALIDITY, is retrieved. All the queries in this section are examples of time queries, that is queries with a time span as a result. In any case, the retrieved time span is determined as a property of objects of object versions (time pertinence). Notice that the output of such queries is in general a collection of time intervals which may also be contiguous. A post-processing of the retrieved data (e.g. tuple coalescence) must be applied to minimize their number. 

Clause (C_{10}) can be expressed as in the following query:

```
SELECT VALIDITY
FROM Employee Emp: WasEng MgrPeriod
WHERE WasEng.Job = "Engineer"
  AND MgrPeriod.Job = "Manager"
  AND WasEng.VALIDITY PRECEDES MgrPeriod.VALIDITY
VALID IN MgrPeriod.VALIDITY
```

where MgrPeriod.VALIDITY denotes the selected time period.

Time selection according to the clause (C_{11}) can be expressed as:

```
SELECT VALIDITY
FROM Employee: Earn40K
WHERE Earn40K.Salary = 40000
VALID IN Employee.HACTUALITY
```

The selected period, denoted by Employee.HACTUALITY, is computed as the transaction-time lifespan of all the qualifying histories. All the histories with a version whose salary value equals $40K qualify for the selection. Notice that the same time values can be retrieved via the query:

```
SELECT ACTUALITY
FROM Employee: Earn40K
WHERE Earn40K.Salary = 40000
ACTUAL IN Employee.HACTUALITY
```

The only difference is in the name of the columns in the resulting table (i.e. In, Out instead of From, To). The different role of VALID and ACTUAL clauses is played in further data selection (temporal restriction of histories) during phase (S_2) as evidenced in the next Section.

### 2.4 Data selection

As final examples, we now consider complete queries. During selection phase (S_2), tuples are selected from the histories already selected in phase (S_{1,a}) restricted to the time spans selected in phase (S_{1,b}) and projected according to the attribute names specified in the target list. If a history variable (with attribute selectors) is present in the target list, this means that some data about the objects denoted by the variable will be retrieved. If the name of a history variable used in the query is not present in the target list, no data about the denoted objects will be retrieved. Such variable has been declared for other purposes (i.e. object and/or time selection). Notice that the only variable which can be used in the target list are history variables, since queries retrieve history portions (e.g. time-determined attribute values of the objects). Relation names can be used in the target list as they can be assumed as implicit history variable names. Consider the queries:
• Q1: retrieve all 1990 salary values of the employees who earned less than $30K in 1985
• Q2: retrieve all salary values, during the period when the employee Mary was a manager,
of the employees who earned less than $30K in 1985
• Q3: retrieve John’s salary when he was a cashier, as of the time Tom’s data were inserted
  in the database
• Q4: find the present salary of the employees who ever received a wrong salary

To formulate query (Q1) in a history-oriented language, a history variable, say Emp1, must be
Declared over the relation Employee in order to denote the histories of the employees we are
Interested in. The selection (Si,a) of such histories needs the declaration of a tuple variable, say
Who. If, for example, the only tuple which satisfies the conditions imposed to Who has Name =
"John", then the only history qualifying for the query is John’s history (result of selection phase
(Si,a)). The whole query is the following:

\[
\text{SELECT Emp1.Salary, VALIDITY (Q1)}
\]
\[
\text{FROM Employee Emp1: Who}
\]
\[
\text{WHERE Who.Salary < 30000}
\]
\[
\text{AND Who IS VALID IN 1985}
\]
\[
\text{VALID IN 1990}
\]

The explicit valid time interval “1990” and the default transaction time point “now” are the
time coordinates (results of selection phase (Si,b)). In phase (S2) of data selection, the tuple(s)
of John’s history overlapping the selected time intervals are selected and projected on Salary,
according to the target list. The timestamps of the resulting tuple(s) are restricted to the time
intervals resulting from time selection (Si,b).

Query (Q2) can be expressed as:

\[
\text{SELECT Emp1.Salary, VALIDITY (Q2)}
\]
\[
\text{FROM Employee Emp1: Who, Employee Emp2: Period}
\]
\[
\text{WHERE Who.Salary < 30000}
\]
\[
\text{AND Who IS VALID IN 1985}
\]
\[
\text{AND Emp2.Name = "Mary"}
\]
\[
\text{AND Period.Job = "Manager"}
\]
\[
\text{VALID IN Period.VALIDITY}
\]

The use of the variables Emp1 and Who for object identification is the same as in the previous
query. The additional tuple variable Period is used for indirect specification of the valid time
coordinate for the time selection phase (Si,b); the transaction time coordinate assumes the default
value “now”. In this case, the values assumed by the tuple variable Period will be, one at a time,
all the tuples with Name = "Mary" and Job = "Manager". The validity interval of each of these
tuples will be used for John’s salary retrieval. The history variable Emp2, on which Period is
declared, is needed because Who and Period are generally versions of different objects.

Query (Q3) requires some data presumably about one version of the object “John”. There is no
difference between this query and queries requiring data about more versions, since the version(s)
of John with Job = "Cashier", used for object selection, are also used for valid time selection:
SELECT Emp1.Salary  
FROM Employee Emp1: WasCash, Employee Emp2  
WHERE WasCash.Job = "Cashier"  
  AND Emp1.Name = "John"  
  AND Emp2.Name = "Tom"  
VALID IN WasCash.VALIDITY  
ACTUAL AT BEGINOF( Emp2.HACTUALITY )

In this case, the ACTUAL clause is used to “roll back” John’s data at the insertion time of Tom’s data, which is computed as the beginning of the transaction-time lifespan of his history. History variables Emp1 and Emp2 are used to denote John’s and Tom’s histories, respectively.

Query (Q4) requires to select employees by means of two of their versions which have different values of the Salary attribute, are valid at the same time t, one is actual “now” (contains the correct value of the salary at t) and the other is actual at t (contains the salary value at t as it was known as of t, when it was payed). Using two tuple variables, say WrongSal and TrueSal, declared over the same history, the query can be formulated as:

SELECT Salary  
FROM Employee: WrongSal TrueSal  
WHERE TrueSal.VALIDITY OVERLAPS WrongSal.VALIDITY  
  AND WrongSal.ACTUALITY OVERLAPS WrongSal.VALIDITY  
  AND TrueSal IS ACTUAL NOW  
  AND WrongSal.Salary <> TrueSal.Salary  
VALID NOW

In the target list, Salary stands for Employee.Salary. The (implicit) history variable name can be omitted since data are retrieved from only one history.

3 Concluding Remarks

The introduction of history and tuple variables does not alter the “expressiveness” of a language in a logical and formal sense. As a matter of fact, it only hides from the user the fact that all the range variables used as tuple variables declared on a history, and the range variable used as the history variable itself, have the same key value. Therefore it saves the explicit writing of equijoin conditions on the key of many range variables, which must be explicated by the language processor according to the history-oriented selection method in order to effectively answer a query. For instance, query (Q2) can be pre-processed and rewritten as:

SELECT Emp1.Salary, Emp1.VALIDITY  
FROM Employee Emp1, Employee Who, Employee Emp2, Employee Period  
WHERE Who.Name = Emp1.Name  
  AND Who.Salary < 30000  
  AND Who.VALIDITY OVERLAPS 1985  
  AND Period.Name = Emp2.Name  
  AND Emp2.Name = "Mary"  
  AND Period.Job = "Manager"  
  AND Emp1.VALIDITY OVERLAPS Period.VALIDITY

OO-14
where Emp1, Who, Emp2 and Period are SQL-like range variables which can bind to tuples. A history-oriented language avoids this pre-processing job being done by the user, who must always also know which are the key attributes in order to perform it.

We feel that history and tuple variables improve clarity, readability and ease of use of a temporal language, introducing the concept of history as an interface between objects of the world and multiple tuples which represent them in a temporal database. However, consensus from database community and user feedback must be earned in order to put forward history-orientation as a standard feature of temporal query languages.

References