A History-oriented Temporal SQL Extension

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
Fax: +39 (0)51 644.3540, Email: {*fgrandi,mrscalas,ptiberio*}*@deis.unibo.it*

Abstract

Dozens of temporal extension of the relational data model and of the query language SQL have appeared in recent years. Recently, a committee formed by researchers from the academic and the industrial worlds designed a consensual extension of the SQL-92 standard to include time, epitomized as TSQL2.

According to the fundamental concepts of *temporal grouping* and *temporal completeness* elaborated by Clifford, Croker and Tuzhilin, TSQL2 and the data model on which it relies (like most of their predecessors) can be shown to be *ungrouped* and, thus, temporally *incomplete*. This means that it has been shown that there may exist temporal data models and query languages formally more *expressive* than TSQL2, and also more easy to use as they would embody a more natural view of objects evolving in time. According to a terminology in use, these temporal languages can also be called *history-oriented*.

The definition of a *history-oriented* temporal SQL extension is the contribution of this paper. We present the lines on which a such an extension should be developed and provide examples of its use. The core of the extension is the addition of special range variables: *history* variables, and *version* or *temporal* variables are the syntactic and semantic tools which enable the support of historical grouping at user-interface level.

1 Introduction

Temporal databases have become a consolidated research field for the last decade. In this period, numerous extensions of the relational model to incorporate time and more than a dozen temporal query languages have been proposed. Important events to ratify the scientific maturity of the field can be considered the issue of the first book on temporal databases [15] and the ARPA/NSF international workshop on an infrastructure for temporal databases held in Arlington (Texas), 1993 [8, 11].

Following the workshop efforts, a language design committee, gathering people from industry and academia, has been formed to develop a specification for a consensus extension of the SQL-92 standard. The result of this effort, the TSQL2 language specification [12, 16], was released in September 1994 (a first draft circulated in December 1993).

In parallel to those initiatives, important insights and new features of temporal languages came into light. In particular, the notion of a *grouped* versus *ungrouped* temporal data model and query language was introduced in [1]. In such a model, all the information concerning the same real world entity is to be considered as "grouped" together. In other words, grouped models enforce the concept of *history* — considered as a whole train of events concerning a database object — as a first-class object of discourse. Clifford, Croker and Tuzhilin showed that grouped models and languages are formally *more expressive* than ungrouped ones. Such property had formally been defined as *temporal completeness* in [1].

In the consenus glossary resulting from another joint effort following the workshop, the "groupedness" of a DBMS (termed "with temporal value integrity" in [1]) merged into the "history-orientedness" concept, originally introduced in [4]. According to the final glossary definition [6], a temporal DBMS is said to be *history-oriented* if:

1. It supports history unique identification (e.g. via time-invariant keys, surrogates or OIDs);
2. The integrity of histories as first-class objects is inherent in the model, in the sense that history-related integrity constraints might be expressed and enforced, and the data manipulation language provides a mechanism (e.g., history vari-

ables and quantification) for direct reference to *object-histories*.

On the other hand, we showed in [5] how a temporal query language provided with two kinds of range variables, namely *version variables* (named "tuple variables" in [5]) and *history variables*, may improve clarity and readability of temporal queries. The two kinds of variables were originally introduced for the Quel extension HoTQuel [3, 4] but [11] shows how they can be used to extend SQL. While history variables are used to denote whole histories, version variables are used to denote homogeneous and time-determined object versions within a given history. These two kinds of variables may improve the language expressiveness, in the sense that they facilitate the user in writing queries.

Furthermore, they can be used to provide the appropriate syntactic and semantic tools needed by a history-oriented DBMS at query language level. If such a language is used within a DBMS based on a logical data model supporting history integrity (e.g. a 1NF relational model with history identity enforced through internal identifiers, like TU_g in [1]), history-orientedness and temporal completeness can be guaranteed. Another possibility is to use *temporal varaibles* instead of version ones, as proposed for the language $\mathrm{SQL}_{\mathtt{hi}}$ [1]. Temporal variables bind to a time-point within the lifespan of the history over which they are declared and can be used to denote a determined version of that history. Although a formal completeness proof for the proposed SQL extension is far beyond the scope of this work, examples will be given in order to show language expressiveness and to show equivalence between the use of version and temporal variables. However, in [1] it has been defined a grouped relational algebra (A_G) on which a history-oriented SQL extension can be based. A sketch of the temporal completeness proof for A_G can also be found in [1].

On the other hand, also ungrouped models and languages, extended to incorporate the *grouping* semantics, can be used to *simulate* grouping and history-orientedness. This is the case, for instance, of TSQL2, as it is provided with a *surrogate* data type [7] which can be used at a certain extent to support history identity and *generalized range variables* [14] which can also play the role of version and history variables. The price to pay is that this sort of grouping — not inherent to the language and to the underlying data model — is not *transparent* to the user. Indeed, it is user's responsibility to make it into use. This may reveal itself to be a quite hard and hazardous task, in particular as to update operations.

A *history-oriented temporal* SQL extension, even if it came out as a *less general* approach than TSQL2, must support grouping in a *transparent* way. Ease of use of the language, in accordance with the natural intuition of objects' evolution over time, will be a direct consequence of such a transparence. Temporal next generation databases should conform to history-oriented data models and provide history-oriented query languages at user-interface level. The SQL extension presented in this work (HoT-SQL), and for which a prototype implementation is under development at the University of Bologna, is designed to meet these requirements. Moreover, it will be enriched by some useful features and construsts developed for TSQL2, from the use of temporal elements as timestamps to time manipulation functions and predicates, though a strong compatibility with TSQL2 is not an issue of this work.

1.1 A History-oriented Temporal Relational Data Model

The data model considered in this work is a relational model with tuple time-stamping, conforming to the Bitemporal Conceptual Data Model (BCDM) [9], although only valid time [6] will be considered here for simplicity. In this model, all the time-varying relations are extended with a temporal attribute whose values are temporal elements, that is unions of maximal disjoint intervals. Since a temporal element may contain any number of intervals, variable-length records must be supported by the system in order not to violate first normal form.

History identity is assumed to be maintained by means of hidden and system-managed identifiers, implemented as surrogates, whose value is the same in all the tuples being versions of the same object. Thus, the set of all the tuples with a common history identifier make up the history of an object (entity or relationship instance).

Primary keys can be defined as in the standard relational model for each snapshot of a given relation, that is for the temporal relation restricted to a given time-point. For the sake of generality, key changes are allowed along time, since identity is maintained along (snapshot) versions with possibly different key values by means of history identifiers. Histories can also have time-invariant (constant) attributes. Special history attributes, which have a unique value per history, can be defined by means of aggregate functions working on the versions composing the history.

The temporal integrity constraints which follow are assumed to hold in any legal database instance.

History integrity: In a given history, different versions cannot be valid at the same time (history *uniqueness*). Although key changes are allowed, no object version can be (even partially) assigned a 'Null' key value or be assigned the key value of another object valid at a different time.

History referential integrity: If K is the primary key of a base relation R matching foreign keys $K_1, \dots, K_n$ in base relations $R_1, \dots, R_n$, when a key update changes the K value in R from k' to k'' valid in period P, also the K_i value in R_i must be changed from k' to k'' valid in period P (for all $i = 1, \dots, n$) within the same transaction. The only change that may involve K_i in isolation is the assignment of a 'Null' value.

In other words, the external identification supported by foreign keys must be maintained throughout any key change.

2 A Relational Language with History, Version and Temporal Variables

Version variables are used in HoT-SQL to denote flat tuples representing time-determined object versions, while history variables properly denote sets of flat tuples composed of all the temporal versions of the same object. Temporal variables are used to denote time-points contained in the lifespan of a history. Each value assumed by such a variable will single out a version in that history. All kinds of variables can be declared in the FROM clause of HoT-SQL statements according to the following syntax:

<from clause> ::=
FROM <var declaration>{, <var declaration>}

<var declaration> ::=
<rel name> [[<h-var name>] :
<tv-var name>{<tv-var name>}]

If the <tv-var name> starts by "t," it is assumed to be a *temporal* variable name, else a *version* variable name. Default rules for optional parts are added for upward compatibility of non-temporal SQL-92, in order to provide a temporal semantics to legacy SQL applications in which histories replace object snapshots (tuples).

In its complete form, the variable declaration in the FROM clause has the following format:

```
FROM Rel H-Var:TV-Var-1...TV-Var-n
```

in which H-Var is a history variable declared over relation Rel and TV-Var-1,...,TV-Var-n are version or temporal variables all declared over history variable H-Var. In the body of the query which the FROM clause belongs to, H-Var denotes the history of an entity of the class the relation Rel represents, whereas TV-Var-1,...,TV-Var-n denote (possibly) different versions of such entity or different time-points in its lifespan.

If the history variable name is omitted, a history variable with the same name of the relation is assumed (allowed only once):

```
FROM Rel:TV-Var-1...TV-Var-n
```

is equivalent to:

```
FROM Rel Rel:TV-Var-1...TV-Var-n
```

If only the relation name appears in the variable declaration part, a history variable and one version variable with the same name of the relation are implicitly assumed:

```
FROM Rel
```

is equivalent to (also allowed only once):

```
FROM Rel Rel:Rel
```

Only history variables may appear in the target list of a SELECT statement, because always histories (or history portions) are retrieved by a HoT-SQL query.

If Attr is an attribute name in the schema of the relation on which version variable V-Var has been declared, then the notation V-Var.Attr denotes the Attr value in the tuple binding to variable V-Var. This is the usual way to reference attribute values within time-determined versions. The expression V-Var.V-Time represents the timestamp of the object version denoted by V-Var.

If t-Var is a temporal variable declared over the history variable H-Var, then the notation H-Var.Attr represents the whole history of the attribute Attr, while the notation H-Var(t-Var) represents the version, valid at the time-point binding to t-Var, of the history binding to H-Var. The equivalent expressions H-Var(t-Var).Attr and H-Var.Attr(t-Var) denote the value of attribute Attr in such a version.

Furthermore, also global history attribute may be defined, and referenced in the HoT-SQL WHERE clause. For instance, if `H-Var` is a history variable then `H-Var.V-Time` represents the *lifespan* of the object denoted by `H-Var`. Aggregate functions can also be used to derive history global properties as follows:

```
H-Var.MAX(Salary)

H-Var.AVG(Hours)

H-Var.FIRST(Job)
```

where the function `FIRST` (`LAST`) yields the value found in the first (last) version of the history denoted by `H-Var`.

In the rest of the paper we will show how history, version and temporal variables can conveniently be used to express queries and updates. For the sake of simplicity we consider two relations with schemas:

```
Employee(Name,Job,Salary|V-Time)

EmpPro(Employee,Project,Hours|V-Time)
```

A hidden attribute `HID`, representing the history identifier, belong to every relation and is completely managed by the system.

2.1 Retrieval statements

The query:

```
SELECT *
   FROM Employee Emp:Job1 Job2
   VALID IN [1990-1992]
   WHERE Job1.Job<>Job2.Job
     AND Job1.Salary=Job2.Salary
     AND Emp.FIRST(Salary)>850
```

retrieves the 1990–1992 history of the employees who changed job whitout a salary change and whose first salary was higher than 850. If we want to consider only consecutive changes, a temporal condition `Job1.V-Time MEETS Job2.V-Time` must be added to the `WHERE` clause to check if `Job1` and `Job2` bind to consecutive employee's versions. If temporal variables are to be used, the same query can be rewritten as:

```
SELECT *
   FROM Employee Emp:t1 t2
   VALID IN [1990-1992]
   WHERE Emp.Job(t1)<>Emp.Job(t2)
     AND Emp.Salary(t1) =
           Emp.Salary(t2)
     AND Emp.Salary(first)>850
```

The special literal "`first`" (`last`) is used as a system defined temporal variable which always binds to the first (last) time-point of the lifespan of the history which it refers to (the history denoted by `Emp` in this case). If we want to consider only consecutive changes, the temporal condition `t1 MEETS t2` must simply be added.

The query:

```
SELECT Emp1.Name,Emp1.Job
   FROM Employee Emp1:IsClerk,
        Employee Emp2:IsMgr
   VALID IN IsMgr.V-Time
   WHERE IsClerk.Job='Clerk'
        AND IsMgr.Name='Smith'
        AND IsMgr.Job='Manager'
```

retrieves the name and the job of any employee who ever was a clerk, during the period(s) in which Smith was a manager. With temporal variables it becomes:

```
SELECT Emp1.Name,Emp1.Job
   FROM Employee Emp1:tClerk,
        Employee Emp2:tMgr
   VALID IN tMgr
   WHERE Emp1.Job(tClerk)='Clerk'
        AND Emp2.Name(tMgr)='Smith'
        AND Emp2.Job(tMgr)='Manager'
```

The query:

```
SELECT Employee,AVG(Hours)
   FROM EmpPro EP:DBMS,
   VALID IN [1994]
   WHERE DBMS.Project='DBMS'
     AND DBMS.V-Time OVERLAPS [1993]
```

For all the employees who worked to the DBMS project in 1993, the query retrieves their names and the average hours they worked to the DBMS project in 1994 (the `EmpPro` relation contains histories of the *relationship* Employee-Project). By using a temporal variable, the query becomes: The query:

```
SELECT Employee,AVG(Hours)
   FROM EmpPro EP:tDBMS,
   VALID IN [1994]
   WHERE EmpPro.Project(tDBMS)='DBMS'
     AND tDBMS IN [1993]
```

It can finally be noticed that, owing to the default declaration rules, the queries:

```
SELECT *
   FROM Employee
```

```
SELECT *
   FROM Employee E, EmpPro EP
   WHERE E.Name=EP.Employee
```

which are also correct SQL queries, are HoT-SQL legal queries, and retrieve complete histories (including valid time). If we consider TSQL2 [16], since surrogates are used to support history identity and must be managed by the user, the join query must be expressed as:

```
SELECT E.*,EP.*
   FROM Employee(HID) AS HE,
        EmpPro(HID) AS HEP,
        HE(*) AS E, HEP(*) AS EP
   WHERE E.Name=EP.Employee
```

which is a bit less readable than HoT-SQL and also incompatible with plain SQL. Range variables HE and HEP are only used here to support history identity, while E and EP are used as tuple variables ranging within histories with a common surrogate value (in order to test the join condition and to build the result).

In HoT-SQL, the mechanism for "grouping" all the tuples with a common HID value to form a history is inherent to the language semantics, whereas, in TSQL2, it must be explicitly managed by means of the first variable declarations: Employee(HID) AS HE,EmpPro(HID) AS HEP.

2.2 Update statements

In snapshot databases, an atomic fact to be stored in a relation can be represented as a tuple (transaction tuple). Such a tuple can simply be appended to the relation in case of insertion or it substitutes *at most another* tuple (target tuple) previously stored in the relation in case of update: a new object version replaces the old one. In temporal databases, while an atomic fact to be stored can still be represented as a transaction tuple (with its own timenstamp representing the validity of the fact), we usually have *more than one* target tuples in case of update. As a matter of fact, all the tuples concerning the same object to be modified, whose validity overlaps the validity of the transaction tuple, are target tuples and must be affected by the update. The loss of unicity of the target tuple raises problems of data consistency which can be solved by means of a history-oriented approach as outlined in [4].

In the insertion statement:

```
INSERT INTO Employee
   VALUES('Ford','Engineer',1200)
   VALID FROM Now
```

the clause VALID FROM Now is optional, since it corresponds to the default update validity (on-time transaction). Since NAME is the key, if another employee whose name is Ford (valid in [Now..UC], where UC means *until changed*) is found in the relation Employee, then the insertion is rejected and the transaction aborts, else a new history (with a newly generated HID) is created. The tuple (Ford,Engineer,1200|[Now..UC]) becomes the first version of the new history.

The statement:

```
UPDATE Employee Emp:Current YearAgo
   SET Salary=Current.Salary +
        0.2*YearAgo.Salary
   VALID FROM 'Dec/1/1994'
   WHERE Emp.Name='Johnson'
     AND Current.V-Time
           OVERLAPS Now
     AND YearAgo.V-Time
           OVERLAPS Now-%1 Year%
```

involves a retroactive salary increase for the employee named Johnson. The increase equals 20% of the salary he earned one year ago.

The statement:

```
DELETE Employee Emp1
   WHERE Emp1.LAST(Salary)+50 >
     ( SELECT AVG(Salary)
        FROM Employee Emp2:SameJob
        VALID IN SameJob.V-Time
        WHERE SameJob.Job =
              Emp1.LAST(Job) )
```

can be used to fire all the employees who earn 50 more than the average salary of any employee with the same job. It can be noticed that the last version of the deleted employees may correspond to their present status or to a future status if proactive changes have been recorded.

Moreover, a new instruction, the RECORD statement originary proposed for HoTQuel [3, 4], is added to HoT-SQL for user-friendly manipulation of historical data. This instruction combines the history-oriented approach for updates with a unified modification semantics resulting in a generalized modification statement able to deal with insertions, updates and deletions in a transparent way and respecting all the integrity constraints required by a history-oriented manipulation language.

For instance, the RECORD statement below:

```
RECORD INTO Employee
   VALUES('Jones','Cashier',950)
```

automatically behaves like the insertion statement:

```
INSERT INTO Employee
   VALUES('Jones','Cashier',950)
```

if no employee with name 'Jones' is already present in the `Employee` relation, else like the statement:

```
UPDATE Employee
   SET Job='Cashier', Salary=950
   WHERE Name='Jones'
```

In any case, the transaction never aborts and the storage of the temporal information contained in the `VALUES` clause of the `RECORD` statement is guaranteed. This mechanism avoids the knowledge of the database content otherwise requested from the user for the choice of the standard `INSERT`/`UPDATE` statements.

3 Conclusions

In this paper we have presented a history-oriented temporal SQL extension, provided with history, version and temporal variables.

A protoype implementation of a subset of the language is under development at the University of Bologna. The system is based on the MULTICAL temporal database system of the University of Arizona [10], on top of which a HoT-SQL preprocessor has been built by means of standard Unix tools (Lexx and Yacc) and C programming. The prototype suffer from several limitations inherited from the MULTICAL host system (e.g. the number of joinable tables limits the number of HoT-SQL declarable variables). The preprocessor implements, in a transparent way, the simulation of grouping via surrogates by translating HoT-SQL queries involving history and version variables into MULTICAL queries with plain range variables and supplemental conditions involving hidden history identifiers. A more robust and complete implementation of the language was beyond the horizon of the present work. A dedicated system, based on an optimized support of the temporally grouped algebra A_G, would probably be the best solution to implement a full HoT-SQL prototype.

Future work will include implementation but also an effort to make HoT-SQL ideas into a concrete proposal of temporal extension (TSQL3) of SQL3, the next-generation standard of database query languages. This would also mean to make HoT-SQL the more compatible as possible with the TSQL2 proposal, provided that a future TSQL3 should also guarantee upward compatibility with TSQL2.

References

[1] J. Clifford, A. Croker, A. Tuzhilin, On Completeness of Historical Relational Query Languages, *ACM Transactions on Database Systems* 19:1, 1994.

[2] J. Clifford, A. Croker, F. Grandi, A. Tuzhilin, On Temporal Grouping, *submitted for publication*, 1995.

[3] F. Grandi, M.R. Scalas, HoTQuel: A History-oriented Temporal Query Language, *Proc. of IEEE European Computer Conference*, Bologna (Italy), 1991.

[4] F. Grandi, M.R. Scalas, P. Tiberio, A History-oriented Data View and Operation Semantics for Temporal Relational Databases, C.I.O.C.-C.N.R. Tech. Rep. No. 76, Bologna, 1991.

[5] F. Grandi, M.R. Scalas, P. Tiberio, History and Tuple Variables for Temporal Query Languages, in [11].

[6] C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes, S. Jajodia (eds.), C. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M.R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A. Tansel. P. Tiberio, G. Widerhold, A Consensus Glossary of Temporal Database Concepts, *ACM SIGMOD Record* 23:1, 1994.

[7] C.S. Jensen, R.T. Snodgrass, The Surrogate Data Type in TSQL2, A TSQL2 Commentary, in [16].

[8] N. Pissinou, R.T. Snodgrass, R. Elmasri, I.S. Mumick, M.T. Özsu, B. Pernici, A. Segev, B. Theodoulidis and U. Dayal, Towards an Infrastructure for temporal Databases: Report of an Invitational ARPA/NSF Workshop, *ACM SIGMOD Record* 23:1, 1994.

[9] C.S. Jensen, M.D. Soo, R.T. Snodgrass, Unifying Temporal Data Models via a Conceptual Model, *Information Systems* 19:7, 1994.

[10] MULTICAL System, release 1.0, University of Arizona, Tucson, November 1993 (code available via anonymous ftp from *cs.arizona.edu*).

[11] R.T. Snodgrass (ed.), *Proc. of the ARPA/NSF International Workshop on an Infrastructure for Temporal Databases*, Arlington (TX), 1993.

[12] R.T. Snodgrass, I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada, TSQL2 Language Specification, *ACM SIGMOD Record* 23:1, 1994.

[13] R.T. Snodgrass, I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada, A TSQL2 Tutorial, *ACM SIGMOD Record* 23:3, 1994.

[14] R.T. Snodgrass, C.S. Jensen, The From clause in TSQL2, A TSQL2 Commentary, in [16].

[15] A. Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, R.T. Snodgrass (eds), *Temporal Databases: Theory, Design and Implementation*, Benjamin-Cummings, 1993.

[16] *The TSQL2 Language Design Committee:* R.T. Snodgrass (chair), I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada, TSQL2 Language Specification, 1994 (available via anonymous ftp from *cs.arizona.edu*).