From Proceedings of 17th VLDB Conference, Barcelona
Pages 423-430, September, 1991

Temporal Logic & Historical Databases

Dov Gabbay & Peter McBrien
Imperial College
Dept. of Computing
180 Queen's Gate
London SW7 2BZ

email: pjm@doc.ic.ac.uk

Abstract

We review attempts at defining a general extension to the relational algebra to include temporal semantics, and define two temporal operators to achieve a temporal relational algebra with a close correspondence to temporal logic using *since* and *until*. We then demonstrate how this temporal relational algebra may to a limited extent be encoded in standard relational algebra, and in turn show how an extended temporal SQL may be encoded in standard SQL.

Keywords: Historical Databases, Relational Algebra, Temporal Logic, Relational Databases, SQL.

Introduction

In {Bubenko, 1977} a model for storing historical information in a relational manner was proposed, since which various alternative approaches to *historical databases* have been advanced, a small selection of which are {Clifford & Tansel, 1985; Tansel, 1987; Snodgrass, 1987; Navathe & Ahmed, 1988; Gadia, 1988}. These all propose some temporal relational model, and then either a relational algebra or query language over the model. Only recently {Tuzhilin & Clifford, 1990} have there been some moves towards unifying the various models proposed by extending the relational model in a general manner to provide a *temporal structure*, and to enhance the relational calculus and algebra to provide a small set of temporal operators which provide fully expressive queries over the temporal structure, akin to the five primitive relational algebra operators over a standard relational structure.

Independently of this work, research into various *temporal logic* programming languages (a few examples being {Abadi & Manna, 1987; Barringer *et al*, 1990; Gabbay, 1989}) has been conducted, invariably assuming a fully deductive temporal database is available in which information may be stored, with an associated Prolog-like query language. It has been shown that *US logic* is fully expressive for a historical data model in the same sense that first-order logic is for a non-temporal data model {Gabbay, 1989}. US logic is comprised of first-order classical logic with the addition of the modal operators *until* and *since*. Assuming a model of time where we have a start time 0 and end time *n* (there are variants of the logic for different models of time) we may loosely define *since* and *until* by the following table:

Operator	*Semantics*
A until B	*A* must hold at all times until the time *B* holds. At time *n* the result is false.
A since B	*A* must have held at all times since *B* held. At time 0 the result is false.

Whilst apparently possessing rather impoverished temporal semantics, these two operators allow for the definition of a wide range of other temporal operators. For instance we may derive the following modal operators:

Operator	*Semantics*
$\blacklozenge A$	*A* held at sometime in the past.
$\bullet A$	*A* held at the previous moment.
$\blacksquare A$	*A* held at all times in the past.
$\lozenge A$	*A* holds at sometime in the future.
$\bigcirc A$	*A* holds at the next moment.
$\square A$	*A* holds at all times in the future.

where the equivalence using until and since of each of these operators will be given later when we give a more formal definition of US logic.

An as yet under developed theme for joining work in historical databases and temporal logic is to restrict the functionality of US logic languages to include only those operators and constructs which are equivalent to those in some temporal extension of the relational model and algebra, in a manner similar to Datalog for the standard relational model {Ullman, 1988}. We would then be able to formulate and manipulate queries on a historical database in a logic programming language based on US logic. Towards this aim, this paper sets out to make a link between relational databases and temporal logic, considering both theoretical and practical aspects. Our two themes will be:

- To introduce a new *temporal relational algebra* (TRA), comprising of the five operators of the relational algebra, namely select σ, project π, Cartesian product $\times$, set difference $-$ and union $\cup$, together with the temporal operators *since-product* and *until-product*.
- To define a Temporal-SQL which implements the new operators of the relational algebra, and demonstrate an encoding of the extended language in standard SQL.

We limit our work at present to consider only the query part of the language. SQL updates will be considered as part of future work.

The provision of a TRA will allow for work in formalizing a temporal Datalog language to be performed. By showing the relationship between the TRA and the relational algebra, we make apparent the changes that are needed in current database technology to fully implement the TRA. This provides the dual benefits of making the results of work in the field of executable temporal logic available to the commercial user in the well established style of a relational database management systems (RDMS), and of allowing existing commercial RDMS to be adapted to the new paradigm with the minimum of alteration.

1 Temporal Structure

In order to describe a temporal algebra in a manner independent of the actual representation of temporal data used, we use the *temporal structure* introduced in {Tuzhilin & Clifford, 1990}, which describes any discrete linear bounded temporal model. A historical database $\mathbf{D_T}$ is considered as a series of relational databases $\mathbf{D} \in \{ \mathbf{D}_t \mid 0 \le t \le n \}$ where the subscript t denotes the time associated with the a particular database. Each $\mathbf{D}_t$ has the same schema as $\mathbf{D_T}$, a restriction natural in the context of databases since in practice we would hope to encode the various instances of a relation $\mathbf{R}_t$ into a single relation $\mathbf{R}'$. By restricting t to some finite subset of the natural numbers 0 … n we achieve a bounded model, and using the usual ordering on the natural numbers achieve a linear model. As a shorthand, we will term each instance of t as a *tick*.

We extend the structure to always include in $\mathbf{D_T}$ a relation *time*, the single attribute of which represents the value of t in any particular $\mathbf{D}_t$. This allows us to represent the explicit naming and comparison of time values found in such languages as TQuel {Snodgrass, 1987} or TSQL {Navathe & Ahmed, 1988}.

Example 1.1

For the rest of the paper we shall use the following temporal structure as an example. $\mathbf{D_T}$ consists of the relations *time(db_time)*, *employee(name,salary)* and *works_for(name,manager)*. The table below shows tuples of the relations at the first few values of t

database	*time*	*employee*	*works_for*
$\mathbf{D_0}$	(0)	(Peter,100) (Richard,90)	(Peter,Ed) (Richard,Ed)
$\mathbf{D_1}$	(1)	(Peter,110) (Richard,90)	(Peter,Ed) (Richard,Dov)
$\mathbf{D_2}$	(2)	(Peter,110) (Richard,140)	(Peter,Dov) (Richard,Dov)
$\mathbf{D_3}$	(3)	(Peter,140) (Richard,140)	(Peter,Dov) (Richard,Dov)

1.1 Schema Evolution

The temporal structure allows for schema evolution in a simple manner. We regard $\mathbf{D_T}$ as including all relations over the lifespan of the historical database. Thus if the schema of example 1.1 were changed at time 4 to include the employee's age, we say that $\mathbf{D_T}$ includes both *employee(name,salary)* and *employee(name,salary,age)* (in a similar manner to Prolog, the number of attributes to the relation must be regarded as part of its name for this to be correct). When the change of relationship is made at time 4, it is a matter of system administration as to whether we copy all records from the old format of *employee* to the new, or leave the two types of relations in the database and query them separately.

1.2 US Logic

A US logic may be defined for a variety of models of time, a detailed example being {Gabbay, 1989}. So as to make the link between the TRA and US logic clear, we define in Table 1.1 a US logic working over the temporal structure. In Table 1.1 A and B are formulae of the logic possibly including other *since* and *until* operators. A query ϕ is made relative to some tick q (and hence $\mathbf{D}_q$), writing the query as *at* q : ϕ. In a similar manner to Datalog, predicates are assumed to correspond either to relations in the database $\mathbf{D}_q$ or to be defined by logical rules.

Operator	*Semantics*
A *since* B	B holds for $\mathbf{D}_s$ and A holds for all $\mathbf{D}_t$ where $s \le t < q$
A *until* B	B holds for $\mathbf{D}_s$ and A holds for all $\mathbf{D}_t$ where $q < t \le s$
$A \wedge B$	A holds for $\mathbf{D}_q$ and B holds for $\mathbf{D}_q$
$A \vee B$	A holds for $\mathbf{D}_q$ or B holds for $\mathbf{D}_q$
$\neg A$	A does not hold for $\mathbf{D}_q$

Table 1.1 : Definition of US Logic

Clearly, at tick 0 there are no previous $\mathbf{D}_t$, and therefore *since* never holds for tick 0. Similarly, at tick n there are no further $\mathbf{D}_t$ and so *until* will never hold at tick n. There is a certain arbitrariness in these definitions, for if we take A *since* B as an example, whether A can be in the same database as B or not ($s \le t$ *or* $s < t$) can be left as a matter of convenience and taste, since the two alternatives can be defined in terms of each other.

Operator	*Equivalent*	*Semantics*
$\blacklozenge A$	*true since A*	A holds for some $\mathbf{D}_t$ where $t < q$
$\bullet A$	*A since true*	A holds for $\mathbf{D}_t$ where $t = q - 1$
$\blacksquare A$	*A since* $\neg \bullet$*true*	A holds for all $\mathbf{D}_t$ where $t < q$
$\Diamond A$	*true until A*	A holds for some $\mathbf{D}_t$ where $t > q$
$\bigcirc A$	*A until true*	A holds for $\mathbf{D}_t$ where $t = q + 1$
$\square A$	*A until* $\neg \bigcirc$*true*	A holds for all $\mathbf{D}_t$ where $t > q$

Table 1.2 : Derived Modal Operators of US Logic

Whilst apparently possessing rather impoverished temporal semantics, these two operators allow for the definition of a wide range of other temporal operators. A set of operators commonly found in the literature is given in Table 1.2, along with the equivalent using just *since* and *until.*

The inclusion of the *time* relation in the temporal structure allows for the time at which a query holds to be found; without such a relation a higher-order function would be required. The time relation will always contain the tick of the database in scope of a temporal operator, for example the following are always true:

- *at t* : *time(x)* holds for $x=t$,
- *at t* : $\blacklozenge$*time(x)* holds for $x \in \{t-1, t-2, ..., 0\}$,
- *at t* : $\bullet$*time(x)* holds for $x=t-1$ iff $t>0$.

We may then find the interval [*s,e*] over which *A* holds by the query:

$$at\ n : \blacklozenge(time(e) \wedge \neg A)\ since\ (A\ since\ (time(s) \wedge \neg A))$$

Since we may obtain the interval over which any general term of the logic holds, we may obtain all of the interval operators in {Allen, 1983} by comparison of the start and end ticks of the intervals related to the terms involved in the operator.

2 Temporal Relational Algebra

The evaluation of a query in the *temporal relational algebra* (TRA) will always be made at a given database time *t*, and will be processed using the information of $\mathbf{D}_t$. In this situation it seems natural to consider the five operators of the relational algebra to query only the tuples of a particular $\mathbf{D}_t$, and to introduce new temporal operators to project the tuples from 'other' databases $\mathbf{D}_{t'}$ (where $t' \neq t$) into the 'present' database. The temporal operators *since* and *until* are first-order complete for a discrete historical database {Gabbay, 1989}, and their conciseness make them the natural choice for providing the extension of the relational algebra to handle time.

To make the relational algebra as expressive as US logic (in the same sense as the relational algebra is to classical logic), we must extend it with operators at least as expressive as *since* and *until*. In {Tuzhilin & Clifford, 1990} two *linear recursive operators* were introduced, the definition of which is as follows:

Past linear recursive operator $L_P(A,B)$

$$t=0 : L_{Pt}(A,B) = \varnothing$$
$$t>0 : L_{Pt}(A,B) = (A_{t-1} \cap L_{Pt-1}(A,B)) \cup B_{t-1}$$

Future linear recursive operator $L_F(A,B)$

$$t=n : L_{Ft}(A,B) = \varnothing$$
$$t<n : L_{Ft}(A,B) = (A_{t+1} \cap L_{Ft+1}(A,B)) \cup B_{t+1}$$

From the definitions it can be seen that *A* and *B* must be union compatible, and that L_P is thus a restricted version of the *since* operator, and L_F is a restricted version of the *until* operator, where the two relations must share the same attributes, and we may only search for pairings between identical tuples, (i.e. we may only ask *A(x) since B(x)* and *A(x) until B(x)*). Thus $L_P(A,B)$ finds the tuples which exist in *A* at all times before the present, but after the same tuple was in *B*.

To show that these operators are fully expressive, it was demonstrated in {Tuzhilin & Clifford, 1990} how to derive the general *since* and *until* operators, where there is no restrictions linking the tuples. For example, to derive *since*, we must make *A* and *B* union compatible by in following manner (where $\mathcal{E}$ is the universal set):

$$\blacklozenge B = L_P(\mathcal{E},B) \qquad \lozenge A = L_F(\mathcal{E},A)$$
$$A' = A \times \blacklozenge ✢ B \qquad B' = \lozenge A \times B$$
$$\text{Since}(A,B) = L_P(A',B')$$

2.1 Temporal Product

As an alternative to the definitions of {Tuzhilin & Clifford, 1990}, we propose to define the *since* and *until* operators of US logic slightly more directly as the products *since-product* and *until-product* given below. This has the advantage that the primitive operators of temporal logic (since and until) are defined directly as new operators of the temporal algebra. We justify the introduction of these new operators by demonstrating how they may be efficiently implemented in sections 3 and 5.

Since-Product $S_\times$

$$S_{\times t}(A,B) = S_{\times t}(A,B,\mathcal{E})$$
$$t=0 : S_{\times t}(A,B,C) = \varnothing$$
$$t>0 : S_{\times t}(A,B,C) = (C \times B_{t-1}) \cup S_{\times t-1}(A,B,(A_{t-1} \cap C))$$

Until-Product $U_\times$

$$U_{\times t}(A,B) = U_{\times t}(A,B,\mathcal{E})$$
$$t=n : U_{\times t}(A,B,C) = \varnothing$$
$$t<n : U_{\times t}(A,B,C) = (C \times B_{t+1}) \cup U_{\times t+1}(A,B,(A_{t+1} \cap C))$$

The addition of $\mathcal{E}$ to form a triple allows us to consider the operator as holding for all *A*, and then recurse backward/forward over time intersecting with the tuples which actually hold at the various times. Note that in these definitions, we consider $\mathcal{E} \times B$ to returned tuples only of the same arity as $A \times B$. In essence, these operators provide the equivalents in the relational algebra of the temporal operators *since* and *until* (as defined in Table 1.1), with no shared variables between the two operands. The formula $A(x)\ S_\times\ B(y)$ finds all pairings of *x* and *y* such that *x* was a member *A* at all times before the current time, since and including the time when *y* was a member of *B*.

Example 2.1 gives the resulting tuples from a since-product made on the temporal structure given in Example 1.1, showing for each database $\mathbf{D}_t$ the result of posing the query *employee* $S_\times$ *works_for* relative to time of tick *t*. As for the standard product operator, the tuples returned contain all attributes of the two relations involved in the operation.

Example 2.1

at t : *employee* $S_\times$ *works_for*

database	*employee* $S_\times$ *works_for*
D_0	∅
D_1	Peter,100,Peter,Ed Peter,100,Richard,Ed Richard,90,Peter,Ed Richard,90,Richard,Ed
D_2	Peter,110,Peter,Ed Peter,110,Richard,Dov Richard,90,Peter,Ed Richard,90,Richard,Dov Richard,90,Richard,Ed
D_3	Peter,110,Peter,Dov Peter,110,Richard,Dov Richard,140,Peter,Ed Richard,140,Richard,Dov Peter,110,Peter,Ed

2.2 Temporal Model Operators

In a similar manner to the definitions for US logic {Gabbay, 1989}, we may define the usual temporal modal operators for our relational calculus according to Table 2.1. We also give the equivalents allowing the additional use of the *time* relation, since this gives a more efficient encoding in practice. Note that ($\mathcal{E}\, S_\times\, \mathcal{E}$) will equal $\mathcal{E}$ at all times bar tick zero when it will be ∅, and thus ($\mathcal{E} - (\mathcal{E}\, S_\times\, \mathcal{E})$) gives the empty set at all times, bar tick zero when it gives $\mathcal{E}$. The time the query is to be evaluated with respect to is tick *t*, and *n* is the maximum tick in our bounded model. All the formulae are enclosed in a π_A operator, indicating that only the attributes of relation *A* should be returned.

Temporal Operator	*Equivalent Using* $S_\times$ & $U_\times$	*Equivalent Using* $S_\times$, $U_\times$ & *Time*
●*A*	$\pi_A(A\ S_\times\ \mathcal{E})$	$\pi_A(A\ S_\times\ \mathrm{time(t-1)})$
○*A*	$\pi_A(A\ U_\times\ \mathcal{E})$	$\pi_A(A\ U_\times\ \mathrm{time(t+1)})$
■*A*	$\pi_A(A\ S_\times\ (\mathcal{E} - (\mathcal{E}\ S_\times\ \mathcal{E})))$	$\pi_A(A\ S_\times\ \mathrm{time(0)})$
□*A*	$\pi_A(A\ U_\times\ (\mathcal{E} - (\mathcal{E}\ U_\times\ \mathcal{E})))$	$\pi_A(A\ U_\times\ \mathrm{time(n)})$
◆*A*	$\pi_A(\mathcal{E}\ S_\times\ A)$	
◇*A*	$\pi_A(\mathcal{E}\ U_\times\ A)$	

Table 2.1 : Derived TRA Modal Operators

2.3 Temporal Join

When variables are shared between the operands, we have a situation similar to the non-temporal join, where certain attributes are used to pair the operands. We thus might define a temporal S_{join} and U_{join} as follows:

Natural since-join on attribute C

$$A\ S_{\mathrm{join(C)}}\ B = \sigma_{\mathrm{A.C=B.C}}(A\ S_\times\ B)$$

Natural until-join on attribute C

$$A\ U_{\mathrm{join(C)}}\ B = \sigma_{\mathrm{A.C=B.C}}(A\ U_\times\ B)$$

The values taken for the *since-join* on the *employee* and *works_for* relations of our example schema and database would be as follows:

database	*employee* $S_{\mathrm{join(name)}}$ *works_for*
D_0	∅
D_1	Peter,100,Ed Richard,90,Ed
D_2	Peter,110,Ed Richard,90,Dov Richard,90,Ed
D_3	Peter,110,Dov Richard,140,Dov Peter,110,Ed

2.4 Explicit References to Time

By including a query on relation *r* in a product with *time,* we find the times at which *r* holds, in a similar manner to US logic. For example, on the structure from Example 1.1 the query

$$at\ 3 : \sigma_{name='Peter'} \blacklozenge (employee\ S_\times\ time)$$

will result in all tuples of the *employee* relation in the past relative to tick 3, appended with the tick with which they are associated:

(Peter,100,0)
(Peter,110,1)
(Peter,110,2)

3 Encoding TRA in the Relational Algebra

In this section we outline a possible mapping of the TRA defined in Section 2.1 to the relational algebra extended with arithmetic capability in select and project operations. We will use this mapping in Section 5 to demonstrate how queries in a *Temporal-SQL* (defined in Section 4) may be translated to queries in standard SQL {Date, 1989}, and thus executed on current RDMS.

3.1 Mapping the Temporal Structure to Relations

A naive approach might be to add to each relation an additional attribute stating the time that tuple holds at. However this would lead to a large amount of redundancy in the database, with a tuple holding over *t* ticks being represented by *t* instances of the tuple with different values in the time attribute. Such series are efficiently encoded using intervals, where we add to each relation *R(x)* a *start_time* and *end_time* attribute, to give an encoded relation *R'(x,start_time,end_time)*. Such an approach has be advocated in {Navathe & Ahmed, 1988}, which in addition describes the data being in a *temporal normal form* if all such intervals are maximal, i.e. for every relation *R* there are no pairs of tuples $R'(x,s_1,e_1)$ and $R'(x,s_2,e_2)$ where $s_2-1 \le e_1$ and $s_1 \le e_2+1$.

The temporal structure (example 1.1) will be encoded as the relations:

time(tick,start_time,end_time),

TRA	*Relational Algebra*
$\mathrm{top}(R \cup_t S)$	$\sigma_{start\_time \le t \le end\_time}\ \mathrm{top}(R) \cup \sigma_{start\_time \le t \le end\_time}\ \mathrm{top}(S)$
$\mathrm{top}(R -_t S)$	$\sigma_{start\_time \le t \le end\_time}\ \mathrm{top}(R) - \sigma_{start\_time \le t \le end\_time}\ \mathrm{top}(S)$
$\mathrm{top}(\sigma_{t\ C}\ R)$	$\sigma_C\ \sigma_{start\_time \le t \le end\_time}\ \mathrm{top}(R)$
$\mathrm{top}(\pi_{t\ C}\ R)$	$\pi_C\ \sigma_{start\_time \le t \le end\_time}\ \mathrm{top}(R)$
$\mathrm{top}(R)$	$\sigma_{start\_time \le t \le end\_time}\ e(R)$
$e(A_t)$	A'
$e(\sigma_{t\ C}\ R)$	$\sigma_C\ e(R)$
$e(\pi_{t\ C}\ R)$	$\pi_C\ e(R)$
$e(R -_t S)$	*not encodable*
$e(R \times_t S)$	$\pi_{R,S,max(R.start\_time,S.start\_time),min(R.end\_time,S.end\_time)}\ (\ \sigma_{overlap(R,S)}\ (\ e(R) \times e(S)\ )\ )$
$e(R \cup_t S)$	*not encodable*
$e(R\ S_{\times t}\ S)$	$\pi_{R,S,max(R.start\_time+1,S.start\_time+1),R.end\_time+1}\ (\ \sigma_{overlap(R,S)}\ (\ e(R) \times e(S)\ )\ )$
$e(R\ U_{\times t}\ S)$	$\pi_{R,S,R.start\_time-1,min(R.end\_time-1,S.end\_time-1)}\ (\ \sigma_{overlap(R,S)}\ (\ e(R) \times e(S)\ )\ )$

Table 3.1: Encoding the TRA in the Relational Algebra

employee(name,salary,start_time,end_time),
works_for(name,manager,start_time,end_time).

The values of the relations are given in the table below. In practice we need not store the time relation since by definition the value of *time* for any time *t* would be (*t,t,t*).

time	*employee*	*works_for*
(0,0,0)	(Peter,100,0,0)	(Peter,Ed,0,1)
(1,1,1)	(Peter,110,1,2)	(Peter,Dov,2,3)
(2,2,2)	(Peter,140,3,3)	(Richard,Ed,0,0)
(3,3,3)	(Richard,90,0,1)	(Richard,Dov,1,3)
	(Richard,140,2,3)	

3.2 Mapping the TRA to Relational Algebra

Given the encoding of the times at which temporal relations hold as interval ranges in non-temporal relations, the TRA operators may be mapped to expressions in the relational algebra extended with arithmetic. The query is made with respect to some evaluation time *t*. Table 3.1 shows the encoding process necessary, by supplying rules for a function *top*, which converts the TRA expression to relational algebra. The process is broken down to a two tier system, where union and difference are only permitted at the 'top-level', i.e. not inside the scope of a product, since-product or until-product TRA operator. All TRA operators are subscripted by *t* to differentiate them from the relational algebra operators.

In Table 1.1 the predicate *overlap(A,B)* holds when $A.start\_time \le B.end\_time \wedge B.start\_time \le A.end\_time$ holds, the arithmetic function *max(A,B)* returns the higher of *A* and *B*, and *min(A,B)* returns the lower of *A* and *B*.

To demonstrate the difficulty in obtaining a general TRA union operator using the relational algebra union, consider the situation illustrated in Figure 3.1 where we want the union of a relation *P* which holds at all odd ticks, and *Q* which holds at all even ticks for the same tuples. Using a relational algebra union between *P* and *Q* can only merge two intervals, merging the intervals of $P \cup_t Q$ requires us to iteratively take the union *P* and *Q* to form intervals of length two ticks, then take the union of the union to make intervals of four ticks, and so on, requiring $n-1$ unions for an *n* tick temporal structure. For example, the union $P \cup_t Q$ in an eight tick structure would require seven unions, and be obtained by the formula $((P \cup Q) \cup (P \cup Q)) \cup ((P \cup Q) \cup (P \cup Q))$ omitting details of the interval intersections.

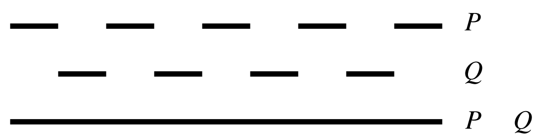

Figure 3.1

Similarly for a general TRA difference operator, consider the situation illustrated in Figure 3.2, where we want the union of a relation *P* which holds at all ticks, and *Q* which holds at all even ticks for the same tuples. Again, the relational algebra $P - Q$ may only subtract one interval of *Q* from *P,* and thus we must subtract *Q* from *P n/2* times to ensure the operation is complete. For example, $P -_t Q$ in an eight tick structure would require four subtractions, and be obtained by the formula $(((P - Q) - Q) - Q) - Q$ again omitting details of the interval intersections.

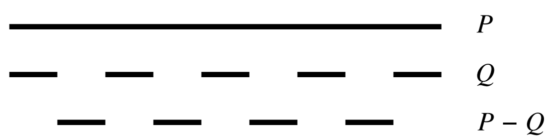

Figure 3.2

Whilst we can for any *n* tick structure give a general first-order encoding of a temporal union or difference in the relational algebra, it can not be regarded as a practical solution to the problem, and thus we may view these two operations as being the fundamental extensions to a system based on the relational algebra necessary to fully implement the TRA.

4 Temporal-SQL

Of particular importance when designing new relational operators is to consider the impact on SQL, and the enhancements necessary to model the new operators. We aim to provide operators which are in the same style as the present operators, both for reasons of acceptability to current users of SQL, and for practicability of the modification of current RDMS.

The since-product and until-product have obvious similarity with the relational algebra product operator, and thus it is natural to extend the **FROM** clause of SQL, where we at present have just the comma operator to indicate the product, to also allow the keywords **SINCE** and **UNTIL** to be used, indicating the corresponding temporal product. In addition, we must prefix any SQL query with an **AT** clause to indicate which time (i.e. which $\mathbf{D}_t$) should be used to evaluate the query against. Example 4.1 shows the equivalent of the TRA formula *employee(a,b)* $S_\times$ *works_for(c,d)* in such a modified SQL (as we would expect the result is identical to that given for $\mathbf{D}_3$ in Example 2.1).

Example 4.1

```
AT 3
SELECT *
FROM EMPLOYEE SINCE WORKS_FOR
```

yields:

NAME	SALARY	NAME	MANAGER
Peter	110	Peter	Dov
Peter	110	Richard	Dov
Richard	140	Peter	Ed
Richard	140	Richard	Dov
Peter	110	Peter	Ed

4.1 Temporal Natural Join

With this extension, we implement the temporal natural-join in a similar manner to the relational join. Thus to compute *employee(a,b)* $S_{join(name)}$ *works_for(a,c)* (i.e. to join on the *name* attribute) we simply add a **WHERE** clause to ensure the attribute is common to the two relations:

Example 4.2

```
AT 3
SELECT EMPLOYEE.NAME,SALARY,MANAGER
FROM EMPLOYEE SINCE WORKS_FOR
WHERE EMPLOYEE.NAME=WORKS_FOR.NAME
```

yields:

NAME	SALARY	MANAGER
Peter	110	Dov
Richard	140	Dov
Peter	110	Ed

4.2 Derived Modal Operators

Modal Operator	*Temporal-SQL Keyword*
●	**PREVIOUS**
○	**NEXT**
■	**ALWAYS_HAS**
□	**ALWAYS_WILL**
◆	**PAST**
◇	**FUTURE**

Table 4.1 : Temporal-SQL Derived Model Operators

Since the derived modal operators of Table 1.2 are shorthands for formulae involving the since- and until-products, we may introduce them as modifications to the **FROM** clause according to Table 4.1. For example, we may use the **PAST** operator to recall all the *managers* that *Peter* has worked for by the following query:

Example 4.3

```
AT 3
SELECT MANAGER
FROM   PAST WORKS_FOR
WHERE  NAME='Peter'
```

yields:

MANAGER
Dov
Ed

To select the *managers* that *Peter* has worked for, but no longer does, we need simply check that the *managers* being selected do not hold for the current (query) time:

Example 4.4

```
AT 3
SELECT MANAGER
FROM   PAST WORKS_FOR
WHERE  NAME='Peter'
   AND  MANAGER NOT IN
           (SELECT MANAGER
            FROM   WORKS_FOR
            WHERE  NAME='Peter')
```

yields:

MANAGER
Ed

5 Encoding Temporal-SQL in Standard SQL

The Temporal-SQL may be encoded in standard SQL {Date, 1989} using the obvious mapping of the encoding scheme used for translating the TRA to the relational algebra. The temporal structure is represented by using additional attributes (say **START** and **END**) on relations to describe the interval over which tuples hold, and we may hide this information by always naming the non-interval attributes in the **SELECT** clause. The **SINCE** and **UNTIL** operators are replaced by products (the SQL comma operator), and additional terms added to the **WHERE** clause to perform the interval intersections necessary in the encoding. The **AT t** clause is used to supply the value of **time**. Note that the restriction on encoding the temporal union operator fits well with SQL, which does not allow a **UNION** inside a **SELECT** clause.

The relational operators must then be translated to test these intervals. Thus we can write an equivalent query to in Example 4.1 (in terms of the tuples selected) as follows:

Example 5.1

```
SELECT EMPLOYEE.NAME,EMPLOYEE.SALARY,
       WORKS_FOR.NAME,WORKS_FOR.MANAGER
FROM   EMPLOYEE,WORKS_FOR
WHERE  EMPLOYEE.NAME=WORKS_FOR.NAME
  AND  time
           BETWEEN MAX(EMPLOYEE.START+1,
                       WORKS_FOR.START+1)
           AND EMPLOYEE.END+1
  AND  EMPLOYEE.START
           BETWEEN WORKS_FOR.START
           AND WORKS_FOR.END
```

The naming of attributes in the **SELECT** clause is necessary to avoid the listing of the 'hidden' extra attributes we added as part of our encoding. Although SQL does not have such a **MAX** function as we have used above, we can achieve the same function by checking that **time** is greater than both the function's arguments.

The temporal natural-join is achieved simply by adding the additional condition to the **WHERE** clause to check the **NAME** attribute of the relations are identical.

For the modal operators, we must define $\mathcal{E}$ in SQL as holding from 0 to n (the range of the bounded model introduced in section 1), and assume is has no attributes. Thus we obtain the query in Example 5.2 to represent the Temporal-SQL query in Example 4.3 . Note that we do not need to check the *overlap* function of the encoding, since everything will overlap $\mathcal{E}$.

Example 5.2

```
SELECT MANAGER
FROM   WORKS_FOR
WHERE  WORKS_FOR.NAME='PETER'
  AND   time
         BETWEEN MAX(1,WORKS_FOR.START+1)
             AND n+1
```

Adding the constraint of Example 4.4 that the manager selected must not belong to *works_for* at the current time gives Example 5.3.

Example 5.3

```
SELECT MANAGER
FROM   WORKS_FOR
WHERE  WORKS_FOR.NAME='PETER'
  AND  time BETWEEN
         MAX(1,WORKS_FOR.START+1) AND n+1
  AND  MANAGER NOT IN
            (SELECT MANAGER
             FROM   WORKS_FOR
             WHERE  time
                  BETWEEN WORKS_FOR.START
                      AND WORKS_FOR.END)
```

Discussion & Conclusions

We have described a temporal relational algebra (TRA) which provides for the representation of US logic formulae in an historical database environment. By describing a possible encoding of this TRA in the relational algebra (extended with arithmetic), we find that only extensions to the non-temporal union and difference relational operators would be required to obtain a fully first-order expressive historical database query language.

The results are used in the design of a query only Temporal-SQL language, and in turn its encoding in standard SQL. Limitations on the use of the union operator in SQL result in only the implementation of the temporal difference operator being a problem in practice.

The TRA may be used as a standard by which the expressive power of various languages proposed for for the querying of historical databases may be gauged. If a language can implement the *since-product* and *until-product* operators it can be said to be first-order expressive over the historical data model in the sense that the relational algebra is over the relational model.

An area yet to be investigated is the subject of efficiency of the operators, and the possibility of query optimization of TRA queries. As with the case of the relational model, a prime objective of such optimization would be reduction of products and joins present in a particular query.

Future work will investigate extending the Temporal-SQL language to include updates, so that the entire query/update semantics of executable temporal logic rules may be represented in the TRA. This would represent a considerable advance on the relational algebra, since we would have a algebraic description of updates as well as queries to the database.

Acknowledgements

The work reported in this paper was partly funded by the EC under Esprit project number 2469. The authors would like to thank all members of the project for their helpful involvement in numerous meetings, and in particular the contributions of A.Conti, M.Niézette, F.Schumacker and

P.Wolper of the University of Liège. We would also like to thank Richard Owens of Imperial College for much work in proof reading and correcting numerous errors present in draft versions of this paper, and the conference referees for helpful suggestions for improvements to the final version of the paper.

References

M.Abadi & Z.Manna, *Temporal Logic Programming*, IEEE Symposium on Logic Programming, 1987.

J.F.Allen, *Maintaining Knowledge about Temporal Intervals,* CACM Vol. 26, No. 11 pp 832-843, 1983.

H.Barringer, M.Fisher, D.Gabbay, G.Gough & R.Owens, *MetateM: A Framework for Programming in Temporal Logic*, Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, *Ed.* J.W.de Bakker, W.P.de Roever & G.Rozenberg, LNCS 430 pp 94-129, Springer-Verlag, 1990.

J.A.Bubenko, *The Temporal Dimension in Information Modelling*, Architecture and Models in Data Base Management Systems, North-Holland, 1977.

J.Clifford & A.U.Tansel, *On An Algebra for Historical Relational Databases: Two Views*, Proc. ACM SIGMOD Conference 1985.

C.J.Date, *A Guide to the SQL Standard*, Addison-Wesley, 1989.

D.Gabbay, *The Declarative Past and Imperative Future*, Temporal Logic in Specification: Altrincham Workshop 1987, LNCS 398 pp 409-448, Springer-Verlag, 1989.

S.K.Gadia, *A Homogeneous Relational Model and Query Languages for Temporal Databases*, ACM TODS, Vol. 13, No. 4 pp 418-448, 1988.

S.B.Navathe & R.Ahmed, *TSQL - A Language Interface for History Databases*, Temporal Aspects of Information Systems, pp 109-122, *Ed.* C.Rolland, F.Bodart & M.Leonard, North-Holland, 1988.

R.Snodgrass, *The Temporal Query Language TQuel,* ACM TODS Vol. 12, No. 2 pp 247-298, 1987.

A.U.Tansel, *Adding Time Dimension to Relational Model and Extending Relational Algebra*, Information Systems Vol 11, No. 4, pp 343-355, 1986.

A.Tuzhilin & J.Clifford*, A Temporal Relational Algebra as a Basis for Temporal Relational Completeness*, Proceedings of the 16th International Conference on Very Large Databases, Brisbane, 1990.

J.D.Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.