Design and Implementation of Extended Boolean and Comparison Operators for Time-Oriented Query Languages

MOSTAFA A. BASSIOUNI, AMAR MUKHERJEE AND MARK J. LLEWELLYN

Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA

Standard Boolean and comparison operators are based on the True/False logic and are therefore not quite suitable for handling events/activities defined over periods of time. In this paper, we generalize the Boolean and comparison operators by allowing their input and output operands to be sets of time intervals. Efficient algorithms for the implementation of the extended operators are presented. The generalized operators can be used to enhance the user interface of time-oriented query languages and allow application users to express their time-related qualifications more succinctly and elegantly. The paper is concluded by examples showing the flexibility of the extended operators in expressing time-related requirements.

Received September, 1993; revised June, 1994

1. INTRODUCTION

To motivate the issues discussed in this paper, we shall consider a historical relational database in which each relation is viewed as a collection of related attributes: some of the attributes are constant while the others are temporal attributes (i.e. their values are time varying). Historical/temporal databases (see Ahn, 1986; Clifford and Tansel, 1985; Gadia and Yeung, 1988; Navathe and Ahmed, 1989; Snodgrass and Ahn, 1986; Snodgrass, 1987) are designed to capture not only the present state of the modeled real world, but also its previous states. The following scheme is an example of a historical relation that contains information about the employees working in the different offices of a commercial company.

```
EMP (name,        /* assume unique names */
     Id,          /* employee number */
     sex,
     salary,
     manager,     /* name of manager */
     c_office,    /* city where office is located */
     c_home)      /* city of residence */
```

where attributes 'name', 'Id' and 'sex' are non-temporal, i.e. constant, attributes while attributes 'salary', 'manager', 'c_office' and 'c_home' are temporal, i.e. time-varying, attributes. For simplicity, we shall assume that names in relation EMP are unique. Thus both name and Id can serve as the time-independent key, TIK (see Navathe and Ahmed, 1989), of relation EMP.

We shall assume that the time element is incorporated into the database using attribute versioning with double timestamping (as in Clifford and Tansel, 1985; Gadia and Yeung, 1988, 1986; Bassiouni, 1988; Bassiouni and Llewellyn, 1991; Bassiouni *et al.*, 1993). This means that a double timestamp (i.e. an interval) is attached to the value of each temporal attribute. Throughout this paper, a double timestamping scheme similar to that of Tansel's model described in Clifford and Tansel (1985) will be used for each time varying attribute. This means that time varying attributes within tuples of a relation will be represented as sets of value/time-interval triples. Each triple will have the form $(v, \langle t_1, t_2 \rangle)$ and will be used to indicate that the value v has been valid from time t_1 to time t_2 (inclusive). The interval $\langle t_1, t_2 \rangle$ is said to be syntactically valid if $t_1 \leqslant t_2$. As in many previous temporal models, we shall assume that time is represented by (encoded as) non-negative integers and that the symbol 'NOW' (or interchangeably the symbol 'Ω') is used to represent the current integer value of time (this assumption does not impose any real restriction since the mapping of calendar times into integer values is possible and is usually used for the actual coding of calendar times). Table 1 gives a simple example of relation EMP based on attribute versioning with double timestamping.

2. GENERALIZED BOOLEAN OPERATORS

The notations ANDS[s], OR[s] and NOT[s] will be used to denote the time-sensitive Boolean operators. To understand the difference between these operators and their standard counterparts, consider the two Boolean expressions, E1 and E2, given below:

Expression E1:

```
c_office = 'Tampa' AND c_home = 'Tampa'
```

Expression E2:

```
c_office = 'Tampa' AND[s] c_home = 'Tampa'
```

The standard expression E1 requires that the (selected) employee has worked in the office located in the city of Tampa and has lived in a residence located in Tampa. Because expression E1 uses the standard AND operator, these two events (conditions) are not required to have occurred simultaneously. This means that expression E1

TABLE 1. Example of relation EMP using attribute versioning (the symbol Ω denotes NOW)

Name	*ID*	*sex*	*salary*	*manager*	*c_office*	*c_home*
Mark	5238	M	{(25000, ⟨1,3⟩), (29000,⟨4,5⟩), (34000,⟨6,Ω⟩)}	{(Fred,⟨1,3⟩), (Jack, ⟨4,5⟩), (John, ⟨6,Ω⟩)}	{(Bithlo,⟨1,3⟩), (Orlando,⟨4,Ω⟩)}	{(Bithlo,⟨1,Ω⟩)}
Fred	3671	M	{(28000,⟨1,3⟩), (29000,⟨4,5⟩), (35000,⟨6,Ω⟩)}	{(Dan,⟨1,4⟩), (Bob,⟨5,Ω⟩)}	{(Bithlo, ⟨1,4⟩), (Tampa,⟨5,Ω⟩)}	{(Bithlo,⟨1,5⟩), (Tampa,⟨6,Ω⟩)}

can be satisfied even if the employee has never lived and worked in Tampa at the same time. Expression E2, on the other hand, uses AND^s and therefore requires that the employee has worked in Tampa while (i.e. at the same time as) living in a home located in Tampa. The time-sensitive operator AND^s takes into consideration the time intervals associated with its two operands and checks to see if these intervals are overlapping. Below, we discuss the time-sensitive Boolean operators in more details.

2.1. Time-sensitive Boolean operators

The time-sensitive Boolean operators accept input operands which are sets of time intervals and produce a set of time intervals (possibly empty) as their output result. As shall be seen later, the extended comparison operators will also return sets of time intervals as their output (rather than a binary value of TRUE or FALSE). For example, when expression E2 is evaluated for employee Fred, it becomes equivalent to the expression

$$\{\langle 5, \text{NOW}\rangle\} \ \text{AND}^s \ \{\langle 6, \text{NOW}\rangle\}$$

which returns the set $\{\langle 6, \text{NOW}\rangle\}$. As another example, assume that the evaluation of the left-hand operand of expression E2 (i.e. `c_office='Tampa'`) for some employee returned the set:

$$\{\langle 3,4\rangle, \langle 6,8\rangle, \langle 12, \text{NOW}\rangle\}$$

and that the evaluation of the right-hand operand (`c_home='Tampa'`) produced the set

$$\{\langle 7,8\rangle, \langle 10,15\rangle\}$$

Then expression E2 becomes equivalent to

$$\{\langle 3,4\rangle, \langle 6,8\rangle, \langle 12, \text{NOW}\rangle\} \ \text{AND}^s \ \{\langle 7,8\rangle, \langle 10,15\rangle\}$$

which returns the set $\{\langle 7,8\rangle, \langle 12,15\rangle\}$ as its output result. The two intervals $\langle 7,8\rangle$ and $\langle 12,15\rangle$ in this case are the only time intervals during which that employee has lived and worked in Tampa at the same time.

Definition If S1 and S2 are two sets of time intervals, then the expression S1 AND^s S2 returns the null set Φ if S1 and S2 do not have any overlapping intervals. Otherwise AND^s returns the time sub-intervals that are contained in both S1 and S2 (i.e. it returns their overlapping).

Below, we apply similar extensions to the operators NOT and OR.

Definition. The time-sensitive unary operator NOT^s returns the set of intervals which is the complement of its operand with respect to the universal interval $\langle 0, \text{NOW}\rangle$. Notice therefore that if S is a set of time intervals, then the expression NOT^s (S) returns Φ only if the union of the intervals of S is the universal interval.

For example, the expression

$$\text{NOT}^s\{\langle 2,2\rangle, \langle 6,8\rangle, \langle 7,9\rangle\}$$

returns the set

$$\{\langle 0,1\rangle, \langle 3,5\rangle, \langle 10, \text{NOW}\rangle\}$$

while the expression

$$\text{NOT}^s\{\langle 0,5\rangle, \langle 2, \text{NOW}\rangle\}$$

returns the null set Φ.

The operator NOT^s can be very useful in expressing certain temporal conditions. The expression NOT^s (`c_home='Orlando'`) produces the set of time intervals during which the employee has not lived in Orlando.

Definition. If S1 and S2 are two sets of time intervals, then the expression S1 OR^s S2 returns Φ only if both S1 and S2 are null sets. Otherwise, it returns the union of the intervals of S1 and S2.

For example, the expression

`manager='Fred'` OR^s `manager='John'`

returns the set of time intervals during which the employee has been under the supervision of either Fred or John. As another example, the expression

$$\{\langle 2,5\rangle, \langle 9,10\rangle\} \ \text{OR}^s \ \{\langle 4,7\rangle, \langle 14, \text{NOW}\rangle\}$$

returns the set

$$\{\langle 2,7\rangle, \langle 9,10\rangle, \langle 14, \text{NOW}\rangle\}.$$

2.2. Properties of the extended Boolean operators

It is easy to see that the extended logic satisfies the properties of the normal Boolean logic. The following lemma establishes this result.

LEMMA 1. *Let R, S, and T be three sets of time intervals defined over integer time points in the range 0 to Ω. The extended Boolean logic satisfies the following axioms.*

1. DeMorgan's laws

NOT^s (R AND^s S)
is equivalent
(NOT^s R) OR^s (NOT^s S)
Similarly,
NOT^s (R OR^s S)
is equivalent to
(NOT^s R) AND^s (NOT^s S)

2. Distributivity of AND over OR
R AND^s (S OR^s T)
is equivalent to
(R AND^s S) OR^s (R AND^s T)

3. Distributitivy of OR over AND
R OR^s (S AND^s T)
is equivalent to
(R OR^s S) AND^s (R OR^s T)

4. Commutativity of AND
R AND^s S
is equivalent to
S AND^s R

5. Commutativity of OR
R OR^s S
is equivalent to
S OR^s R

6. Idempotence of AND
R
is equivalent to
R AND^s R

7. Indempotence of OR
R
is equivalent to
R OR^s R

8. Associativity of AND
(R AND^s S) AND^s T
is equivalent to
R AND^s (S AND^s T)

9. Associativity of OR
(R OR^s S) OR^s T
is equivalent to
R OR^s (S OR^s T)

10. Double negation
R
is equivalent to
NOT^s (NOT^s R)

11. Identity for OR (null set)
R OR^s Φ
is equivalent to
R

12. Identity for AND (universal set)
R AND^s $\{\langle 0, \Omega \rangle\}$
is equivalent to
R

13. Complement properties
R AND^s (NOT^s R)
is equivalent to
Φ
R OR^s (NOT^s R)
is equivalent to
$\{\langle 0, \Omega \rangle\}$

14. Universal and null sets
R OR^s $\{\langle 0, \Omega \rangle\}$
is equivalent to
$\{\langle 0, \Omega \rangle\}$
R AND^s Φ
is equivalent to
Φ

Proof to the above lemma is quite straightforward and follows directly from the definition of the extended operators. It is worth mentioning in this context that our generalization of the Boolean operators is quite different, in nature and purpose, from that used in multiple-valued switching logic. In the latter case, the AND and OR operators are defined to be MAX and MIN functions, respectively, operating on integer-valued inputs.

3. IMPLEMENTATION ISSUES

The extended Boolean operators defined earlier have been found to be amenable to efficient software and hardware implementations. In what follows, we discuss our approaches for the implementation of these extended operators. Our discussion will deal with aspects that are general to the different potential applications and will concentrate on the extended operators themselves. In some applications, additional factors may need to be considered to further improve the overall performance of the system. For example, in the case of temporal/historical databases, the algorithms implementing the extended operators will need to interact with lower-level indexing schemes used to reduce the time needed to scan the data files and to filter the input relations in order to collect candidate tuples that have the potential to satisfy the Boolean qualification. Such indexing schemes, though important, represent a separate problem that is outside the scope of this paper.

We shall illustrate our approaches by discussing the implementation of the expression S AND^s T. Only minor extensions will be needed to apply this approach to the other generalized operators. Our discussion will start by considering the case when each of the two operands, S and T, consists of mutually disjoint intervals that are ordered in increasing time value. Two equivalent algorithms to handle this case are presented: a conventional method and a method based on a new bit-wise approach. The bit-wise method is more suitable for hardware-assisted implementations; our preliminary investigation indicates that a temporal VLSI chip for implementing the extended operators based on the bit-wise approach could yield very high speeds.

Consider the expression

S AND^s T

where S and T are two sets of time intervals. Let

$$S = \{\langle ss(1), sf(1)\rangle, \langle ss(2), sf(2)\rangle, \ldots, \langle ss(n), sf(n)\rangle\}$$

$$T = \{\langle ts(1), tf(1)\rangle, \langle ts(2), tf(2)\rangle, \ldots, \langle ts(m), tf(m)\rangle\}$$

The notation ss(j) denotes the value of the start point (left endpoint) of the jth interval of S while sf(j) denotes the finish point (right endpoint) of the jth interval of S. Similarly, ts(k) and tf(k) are used to denote the start point and finish point, respectively, of the kth interval of T. If S and T consist of sorted, mutually disjoint, syntactically valid intervals, then the following conditions hold true

$$ss(j) \leqslant sf(j) \quad \text{where} \quad 1 \leqslant j \leqslant n$$

$$sf(j) < ss(j+1) \quad \text{where} \quad 1 \leqslant j < n$$

$$ts(k) \leqslant tf(k) \quad \text{where} \quad 1 \leqslant k \leqslant m$$

$$tf(k) < ts(k+1) \quad \text{where} \quad 1 \leqslant k < m$$

We first present a simple, but efficient, algorithm to evaluate the expression S AND^s T in the above case; we call this algorithm S_AND, S for simple.

```
Algorithm  S_AND (S,n,T,m);
   /* S and T consist of sorted mutually disjoint
                                    intervals */
   j := k :=1  /* initial value of pointers is one
                                                */
   while (j ⩽ n and k ⩽ m) do
   begin
         /* skip intervals in T that precede the
                          jth interval of S */
         while ss(j) > tf(k) do k:=k+1 endwhile;
         /* does the jth interval of S overlap with
               the kth interval of T? */
         if sf(j) < ts(k) then j:=j+1 /* if no,
                       get next interval in S */
         else
         begin /* there is overlapping */
               s:= max {ss(j), ts(k)};
               f:= min {sf(j), tf(k)};
               output the interval <s,f>
               /* now adjust pointers */
               if sf(j) > tf(k) then k:=k+1 else
                                  j:=j+1 endif;
          endif;
   endwhile;
end S_AND
```

The time complexity of algorithm S_AND is O(n+m). We next consider an alternative bit-wise method which we call algorithm B_AND, B for bit-wise operations. Algorithm B_AND is equivalent to algorithm S_AND and also has a time complexity of O(n+m).

Algorithm B_AND(S,n,T,m):
Input: two sets

$$S = \{\langle ss(1), sf(1)\rangle, \langle ss(2), sf(2)\rangle, \ldots, \langle ss(n), sf(n)\rangle\}$$
$$T = \{\langle ts(1), tf(1)\rangle, \langle ts(2), tf(2)\rangle, \ldots, \langle ts(m), tf(m)\rangle\}$$

where the time intervals in both S and T are sorted and mutually disjoint.

1. Sort all the time points ss(1), sf(1),..., sf(n), ts(1), tf(1), ..., tf(m) into one master sequence, deleting any repeated elements. This sort is a merge-like sort and has a time complexity O(n+m). Let us denote the resulting sequence by Y.
2. For the set S, repeat the following action for j = 1 to n − 1. If the finish point sf(j) and the start point ss(j+1) appear adjacent to each other in Y, then separate them by inserting into Y the special symbol # between sf(j) and ss(j+1).
3. For the set T, repeat the following step for k = 1 to m − 1. If the points tf(k) and ts(k+1) appear adjacent to each other in Y, then separate them by adding to Y the special symbol # between tf(k) and ts(k+1).
4. Assume that Y now consists of r elements (including the extra special symbols) as follows
$$Y = y(1), y(2), \ldots, y(r)$$
5. The set S′ is obtained from S by transforming each interval $\langle ss(j), sf(j)\rangle$ into a corresponding interval $\langle a, b\rangle$ where a and b are integers such that ss(j) = y(a) and sf(j) = y(b). Similarly, the set T′ is obtained from T by transforming each interval $\langle ts(k), tf(k)\rangle$ into a corresponding interval $\langle c, d\rangle$ where c and d are integers such that ts(k) = y(c) and tf(k) = y(d).
6. The two sets S′ and T′ are next transformed into the two binary strings S″ and T″, of length r bits each. The jth bit in S″ has the value 1 only if there is an interval $\langle a, b\rangle$ in S′ such that $a \leqslant j \leqslant b$. Similarly, the kth bit in T″ has the value 1 only if there is an interval $\langle c, d\rangle$ in T′ such that $c \leqslant k \leqslant d$.
7. A logical bitwise AND operation is then performed on the two binary strings S″ and T″. Let us denote the resulting r-bit binary vector by V.
8. The vector V resulting from step 7 is next transformed into a set of intervals, say R, using the inverse of the logic used in step 6.
9. Finally, using the integer values in the set R as indices to the sequence Y (i.e. applying the inverse of the transformation used in step 5), we can map R into a set of time intervals. Print these intervals, they are the correct result of S AND^s T.

Example 1. Consider the two sets

$$S = \{\langle 3, 8\rangle, \langle 15, 20\rangle, \langle 28, 41\rangle, \langle 44, 45\rangle\}$$

$$T = \{\langle 2, 3\rangle, \langle 7, 10\rangle, \langle 18, 22\rangle, \langle 40, 44\rangle\}$$

Step 1 in the above method produces the master sequence

$$Y = \{2, 3, 7, 8, 10, 15, 18, 20, 22, 28, 40, 41, 44, 45\}$$

Step 2 will insert # between 41 and 44 since these two points appear adjacent to each other in Y and they are the finish point and start point, respectively, of two consecutive intervals in S. Similarly, step 3 will insert # between 3 and 7 since these two points are adjacent in Y and they are the finish point and start point, respectively,

of two consecutive intervals in T. Thus according to step 4, the final value of Y and the corresponding indices of its elements are given by

$$Y = 2,3,\#,7,8,10,15,18,20,22,28,40,41,\#,44,45$$

$$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16$$

Step 5 produces the new sets

$$S' = \{\langle 2,5\rangle, \langle 7,9\rangle, \langle 11,13\rangle, \langle 15,16\rangle\}$$

$$T' = \{\langle 1,2\rangle, \langle 4,6\rangle, \langle 8,10\rangle, \langle 12,15\rangle\}$$

Notice that the values in the above sets represent indices and are no longer time values. Step 6 produces the two 16-bit binary strings

$$S'' = \text{'0111101110111011'}$$
$$T'' = \text{'1101110111011110'}$$

The interval $\langle 2,5\rangle$ in S', for example, generates four 1's, occupying positions 2 through 5 in S''. The AND^s temporal operator can now be implemented by simply performing the logical bit-wise AND operation on S'' and T''. In our example, this operation produces the following 16-bit binary vector

$$V = \text{'0101100110011010'}$$

Translating the above string back into intervals (i.e. reversing the logic of the earlier steps), we get

$$R = \{\langle 2,2\rangle, \langle 4,5\rangle, \langle 8,9\rangle, \langle 12,13\rangle, \langle 15,15\rangle\}$$

Using the integer values in the above set as indices to the sequence Y, we finally map the set R into the following set

$$\text{Answer} = \{\langle 3,3\rangle, \langle 7,8\rangle, \langle 18,20\rangle, \langle 40,41\rangle, \langle 44,44\rangle\}$$

which is the correct result of S AND^s T.

LEMMA 2.

(a) Method B_AND correctly computes the value of S AND^s T when the operands S and T consist of sorted mutually disjoint time intervals.

(b) The time complexity of algorithm B_AND is $O(n+m)$.

Proof of the above lemma follows easily from the definition of the AND^s operator and the sequence of transformations performed in method B_AND. Notice that when the two operands are transformed into binary strings (step 6), each special symbol # in the sequence Y will insert a 0 between two adjacent sequences of 1's corresponding to two non-overlapping time intervals in one of the two operands. This will ensure that these two intervals will not be mistakenly treated as one continuous interval when the logical bit-wise operation is performed (e.g. without the special symbol #, the two intervals $\langle 2,3\rangle$ and $\langle 7,10\rangle$ in the operand T would have been treated in Step 8 as the larger interval $\langle 2,10\rangle$). The concept of using the special symbol # is therefore necessary for the correctness of the above method.

Although algorithm B_AND has more preparation steps than S_AND, a careful examination of algorithm B_AND would reveal that these preparation steps don't introduce any significant overhead in the linear time complexity of the algorithm. For example, steps 2 through 6 preceding the bit-wise AND operation can be simply implemented during the same linear scan needed to compute Y from the intervals of both S and T (similar to the scan used in MERGE_SORT to merge two sorted lists into a third sorted list).

Both algorithm S_AND and B_AND have the same linear time complexity and are quite efficient. Algorithm S_AND would be the logical choice for software implementations because it is easier to code and because the bit-wise operation of algorithm B_AND is not expected to give its full benefit in software implementations. The approach represented by method B_AND, on the other hand, is quite suitable for efficient hardware (VLSI) implementations. In addition to the fast bit-wise AND operation, all other steps of method B_AND are straightforward and amenable to efficient VLSI designs. Appendix I discusses some high-level details of the VLSI design of algorithm B_AND. Detailed description of a complete VLSI design of method B_AND is, however, beyond the scope of this paper.

4. THE PARTITION PROBLEM

We next consider the problem when the operands S and T are not composed of mutually disjoint intervals. For example, consider the evaluation of A AND^s T where the two operands are given by

$$S = \{\langle 2,5\rangle, \langle 3,4\rangle, \langle 7,8\rangle, \langle 12,13\rangle, \langle 11,\text{NOW}\rangle\}$$

$$T = \{\langle 5,8\rangle, \langle 1,2\rangle, \langle 12,12\rangle, \langle 0,3\rangle\}$$

It is easy to sort each operand and reduce it into an equivalent set of intervals that are mutually disjoint as follows

$$S = \{\langle 2,8\rangle, \langle 11,\text{NOW}\rangle\}$$

$$T = \{\langle 0,3\rangle, \langle 5,8\rangle, \langle 12,12\rangle\}$$

We can then use algorithm S_AND (or the alternative algorithm B_AND) on the resulting (reduced) operands to get

$$S \text{ AND}^s T = \{\langle 2,3\rangle, \langle 5,8\rangle, \langle 12,12\rangle\}$$

In many situations, however, it is desirable to compute the temporal Boolean expression using the original (non-reduced) intervals of S and T (e.g. for purposes of indexing when the intervals are derived from different comparison operators, or for the computation of statistical queries that deal with the count of such intervals, etc.). We would like therefore to be able to compute the result of S AND^s T without reducing the individual intervals of each operand. Below, we present one approach to solve this problem.

TABLE 2. Evaluation of AND^s using partitions

First operand	*Second operand*	*Result of AND^s*
S(1)	T(1)	$\{\langle 2,2\rangle, \langle 5,5\rangle, \langle 7,8\rangle, \langle 12,12\rangle\}$
S(1)	T(2)	$\{\langle 2,3\rangle\}$
S(2)	T(1)	$\{\langle 12,12\rangle\}$
S(2)	T(2)	$\{\langle 3,3\rangle\}$
S(3)	T(1)	$\{\langle 5,7\rangle\}$
S(3)	T(2)	$\{\langle 3,3\rangle\}$

Example 2. Consider the two sets

$$S = \{\langle 2,5\rangle, \langle 3,4\rangle, \langle 7,8\rangle, \langle 3,7\rangle, \langle 12,13\rangle, \langle 11,\text{NOW}\rangle\}$$

$$T = \{\langle 5,8\rangle, \langle 1,2\rangle, \langle 12,12\rangle, \langle 0,3\rangle\}$$

We can partition each operand into subsets containing mutually disjoint intervals. For example, the set S can be partitioned into the following three sets

$$S(1) = \{\langle 2,5\rangle, \langle 7,8\rangle, \langle 12,13\rangle\}$$

$$S(2) = \{\langle 3,4\rangle, \langle 11,\text{NOW}\rangle\}$$

$$S(3) = \{\langle 3,7\rangle\}$$

Similarly, the set T is partitioned into the following two sets

$$T(1) = \{\langle 1,2\rangle, \langle 5,8\rangle, \langle 12,12\rangle\}$$

$$T(2) = \{\langle 0,3\rangle\}$$

As shown in Table 2, we can now apply algorithm S_AND or B_AND to find the value of S(j) AND^s T(k), where $1 \leqslant j \leqslant 3$ and $1 \leqslant k \leqslant 2$.

The union of the sets given in the third column of Table 2 produces the final result of S AND^s T. Notice that when this union is reduced, we get the same result that was obtained earlier. Notice also that the partitions of S and T are not unique. For example, one (extreme) way is to partition S and T into sets each of which contains a single interval. The opposite extreme case would be to find the partition that gives the minimum number of subsets. Both S and T in the previous example were decomposed into the minimum number of subsets. Notice that this latter strategy does not produce a unique decomposition since there may exist several distinct partitions having the minimum number of subsets. All different partitions of S and T, however, will still give the same result of S AND^s T. The approach used in the above example is outlined in algorithm P_AND, P for partition, which works by decomposing the operands into sets containing sorted mutually disjoint time intervals.

```
Algorithm  P_AND (S,n,T,m);
     Sort  the n intervals of S in nondecreasing
                          order of start points
     Call PARTITION (S,n);
     /*    S has now been partitioned into the sets
           S(1), S(2), S(3), ..., S(p) where the
           number of intervals in the set S(j) is
           given by n(j),
           thus we have n = n(1) + n(2) +... +
                                          n(p)  */
     Sort the m intervals of T in nondecreasing
                          order of start points
     Call PARTITION (T,m);
     /*    T has now been partitioned into the sets
           T(1), T(2), T(3), ..., T(q) where the
           number of intervals in the set T(k)
           is given by m(k), thus we have m = m(1)
           + m(2) + ... + m(q)  */
     /* The following is the main portion of code */
     for j= 1 to p do  /* nested for loop */
           for k = 1 to q do
                       Call S_AND ( S(j),n(j),T(k),
                       m(k) );  /* or call B_AND */
           endfor;
     endfor;  /* end of main portion of code */
end P_AND;
```

It is obvious that the time complexity of algorithm P_AND depends on the speed of the PARTITION subroutine. A careful examination of the main portion of code (nested for loop) in algorithm P_AND shows that the time complexity of this portion of code is O(n*q + m*p). Thus the time complexity of algorithm P_AND not only depends on the time complexity of subroutine PARTITION, but also depends on the value of the partition produced by this subroutine. But since the time complexity of the 'nested for loop' is O(n*q + m*p), the algorithm is optimized if we use a PARTITION subroutine which always decomposes its argument into the minimum number of sets (i.e. which always gives the minimum values of p and q). An algorithm to partition S and T into the minimum number of sets containing mutually disjoint time intervals (which we call the optimal PARTITION algorithm) can be easily developed and would have a worst-case time complexity of $O(n^2)$ for S and $O(m^2)$ for T. However, the actual gain achieved in this case may be offset by the extra computational overhead of the optimal PARTITION subroutine itself. On the other hand, if we use a fast partitioning strategy that simply decomposes the argument into singleton sets (i.e. p = n and q = m), the PARTITION subroutine will have a total time complexity of O(n + m), but the time complexity of the 'nested for loop' will be at its worst value of O(n*m). A fast heuristic algorithm may therefore be used as a good compromise for this problem (recall that all different partitions will still give the same correct solution). Below, we discuss the optimal algorithm for partitioning as well as a greedy heuristic that can be used as a good alternative.

Algorithm OPT_PARTITION decomposes its argument S into buckets (subsets) while linearly scanning the intervals of S in nondecreasing order of left endpoints. The intervals in any bucket are divided into two lists: active and inactive. The active list of a bucket contains those intervals which are still current (with respect to the linear scan) and could therefore overlap with incoming intervals. The code of OPT_PARTITION is given below. Details of the data structures (linked lists)

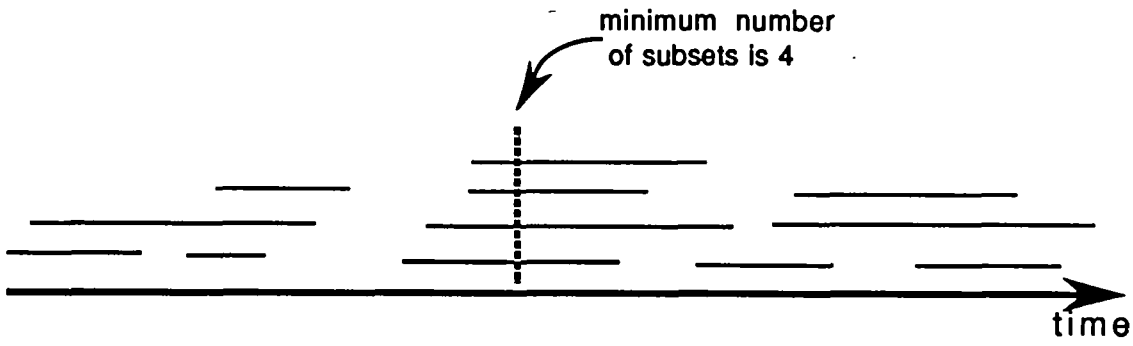

FIGURE 1.

and the associated code to manage the active lists are not given.

```
Algorithm  OPT_PARTITION (S,n);
/* S is the set {<ss(1),sf(1)>, <ss(2),sf(2)>,...,
                                  <ss(n),sf(n)>} */
/* where  ss(j) ⩽ ss(j+1)  for 1 ⩽ j < n */
/* At the end of this algorithm, the set S will be
      decomposed into the subsets  S(1), S(2), ...,
                                                S(p)
       where p is minimum */
       p := 0; /* initial value of p */
       /* process intervals in nondecreasing order
                          of left endpoints */
       for j := 1 to n do
       begin
            k := 1;   Done := False ;
            while (k ⩽ p and Done=False) do;
            begin
                 remove from the active list of
                                 bucket S(k) any
                       interval <s,f> such that
                                              f<ss(j)
                 if (active list of S(k) is empty)
                                               then
                       begin
                       add the interval <ss(j),
                       sf(j)> to S(k);
                       active list of S(k) :=
                       {<ss(j),sf(j)>}
                        Done := True
                        end
                 else  k := k + 1 endif;
            endwhile;
            if Done = False then
            begin
                 p := p + 1 ; /* create a new
                                          bucket */
                 add the interval <ss(j),sf(j)> to
                                        bucket S(p);
                 active list of S(p) := {<ss(j),
                                           sf(j)>}
            endif;
       endfor ;
end OPT_PARTITION;
```

LEMMA 3. *Algorithm OPT_PARTITION decomposes its argument into the minimum number of subsets containing mutually disjoint intervals.*

Proof of this lemma follows directly from the observation that the minimum number of such subsets is equal to the maximum number of intervals that are mutually overlapping (as shown in Figure 1). It is easy to see that the solution obtained by OPT_PARTITION does not exceed this number.

LEMMA 4. *The time complexity of algorithm OPT_PARTITION is $O(n^2)$. If this algorithm is used, the time complexity of algorithm P_AND becomes $O(n^2 + m^2)$.*

Proof of the above lemma follows easily from the code of algorithms OPT_PARTITION and P_AND.

Since algorithm P_AND gives the same correct solution regardless of the way S and T are partitioned, the following greedy heuristic can be used as a good alternative to OPT_PARTITION.

```
Algorithm  GREEDY_PARTITION (S,n);
/* S is the set  { <ss(1),sf(1)>, <ss(2),sf(2)>,...,
                                  <ss(n),sf(n)> } */
/* where ss(j) ⩽ ss(j+1)  for 1 ⩽ j < n */
/* At the end of this algorithm, the set S will be
      decomposed into the subsets S(1), S(2),...,
                                              S(p) */
       p := 0 ; /* initial value of p */
       while (S is not empty) do
       begin
           p := p + 1;
           S(p) := Φ; /* initialize the new subset
                                                 */
           j := 1 ; /* pick the first interval of S */
           Done := False ;
           while (Done=False) do;
           begin
                s := ss(j) ; f := sf(j) ;
                add the interval <s,f> to the subset
                                              S(p);
                remove the interval <s,f> from S ;
                search the set S to find the
                           smallest value i
                      such that f < ss(i)
                if (search successful) then j := i
                else Done := True endif;
           endwhile
       endwhile;
end PARTITION;
```

Notice that since S is sorted on the left endpoint, the

process of searching it (in the above algorithm) has a time complexity of O(log n). This leads directly to the following lemma.

LEMMA 5. *The time complexity of algorithm GREEDY_PARTITION is O(n*log n). If this greedy algorithm is used, the time complexity of algorithm P_AND becomes O(n*log(n) + m*log(m) + n*q + m*p).*

5. TIME-SENSITIVE COMPARISON OPERATORS

Applying the same notation used for the time-sensitive Boolean operators, the definition of comparison operators can be extended in order to allow for temporal comparisons. For example, the time-sensitive 'equal to' operator denoted by '$=^s$' returns the set of intervals during which the two operands have had equal value. Thus, the expression

$$\text{Fred.salary} =^s \text{Mark.salary}$$

returns the set of time intervals during which Fred has had the same salary as that of Mark. According to Table 1, the above expression returns the set $\{\langle 4, 5\rangle\}$. Similarly, the expression

$$\text{Fred.c\_office} =^s \text{Fred.c\_home}$$

returns the set $\{\langle 1, 4\rangle, \langle 6, \text{NOW}\rangle\}$ and the expression

$$\text{Fred.manager} =^s \text{Mark.manager}$$

returns Φ. The following list gives the time-sensitive version of some of the popular comparison operators

$$=^s, <^s, >^s, \neq^s, \leqslant^s, \geqslant^s$$

The input to each of the above operators consists of two sets of value/time interval triples. The output is a set (possibly empty) of time intervals. Notice that if there exists no point of time at which the comparison condition is met, the empty set Φ is returned. Otherwise, the comparison operator will return the set of time intervals during which the comparison condition is satisfied.

The rule for handling operands that are constants or nontemporal attributes is quite simple: such operands are automatically given the universal interval $\langle 0, \text{NOW}\rangle$. For example, the expression

$$\text{Fred.Salary} >^s 28\,000$$

has a left operand which is a temporal attribute (i.e. a set of salary/time interval triples) and a right operand which is a constant. The latter operand is then treated as if it were the set $\{(28\,000, \langle 0, \text{NOW}\rangle)\}$. The extended comparison is then performed to produce the set of intervals during which Fred has a salary larger than $28 000.

6. APPLICATION TO QUERY LANGUAGES

In this section, we illustrate by examples the application of the extended operators to programming and database systems. It must be stressed that the following discussion is not intended to give the detailed syntactic and semantic description of a new programming/query language, nor should the examples and the syntax used in this section be interpreted to imply our preference to any particular existing language. Our emphasis is that the generalized operators can be used to enrich these languages and to provide an easier and more natural temporal interface. Using the generalized operators, users can express their queries more succinctly and elegantly.

Example 3. For each employee, the qualification

$$(\text{salary} >^s 50000)\text{AND}^s(\text{c\_home} =^s \text{"Tampa"})$$

returns the set of intervals (if any) during which that employee has had a salary of $50 000 or greater while residing in Tampa. For example, using a SQL-based syntax, the following query gives the names of employees who satisfy the above qualification (i.e. have made a salary of $50 000 or greater while residing in Tampa).

```
SELECT    name
FROM      EMP
WHERE     (salary >s 50 000) ANDs (c_home =s Tampa)
```

For each employee in relation EMP, the selection qualification in the WHERE clause is used to determine whether that employee is a candidate for output (the qualification must evaluate to a non-empty set of intervals). Notice that any of the operands of AND^s can be replaced by a set of intervals (constant). For example the expression

$$(\text{salary} >^s 50000)\text{AND}^s\{\langle 2, 7\rangle\}$$

selects employees who made a salary of $50 000 or greater at some time during the period from 2 to 7.

Example 4. If the generalized logic is incorporated into the Tansel's historical relational algebra language proposed in Clifford and Tansel (1985), the following expression can then be used to extract the names of the employees who, while living in 'Tampa' have earned more than $40 000.

$$\Pi_{\text{name}}(\sigma_F(\text{EMP}))$$

where Π denotes projection, σ denotes selection and F is the expression

$$\text{c\_home} =^s \text{'Tampa' AND}^s \text{ salary} =^s 40000$$

Notice that in Tansel's original algebra, the expression F must be a standard Boolean condition. This forces the user to apply a sequence of UNPACK and TRIPLE_DECOMPOSITION operations (see Clifford and Tansel, 1985) to transform the historical relation EMP to another relation having simple attributes (i.e. each triple is split into three attributes with only a single value per attribute). Time slicing and normal selection are then applied followed by Pack and TRIPLE_FORMATION. Thus the solution for

the above query in Tansel's original algebra would be as follows.

$$\Pi_{name}(\sigma_Q(T\_DEC_{c\_home}(T\_DEC_{salary}(SLICE_{c\_home,salary}(UNPACK_{c\_home}(UNPACK_{salary}(EMP)))))))$$

where Q is the standard Boolean expression:

c_home ='Tampa' AND salary = 40000

A brief explanation of the above expression is as follows. First relation EMP is first unpacked on attributes 'c_home' and 'salary'. The UNPACK operator transforms each tuple (comprising a set of triples in each of the two attributes) into a collection of tuples each having a single triple in attributes c-home and salary. The slice operator checks the interval portion of pairs of triples from attributes c_home and salary, and keeps only those tuples whose intervals overlap. Next a triple decomposition, T_DEC, is performed on attributes c_home and salary. The T_DEC operator splits attribute c_home into three attributes denoted c_home, c_home_L, c_home_U as follows:

1. The new attribute c_home contains the value portion of the original c_home attribute.
2. Attribute c_home_L contains the lower (start) time of the interval. and
3. Attribute c_home_U contains the upper (finish) time of the interval.

A similar transformation occurs to attribute salary. The selection operation is now applied using a standard Boolean qualification. Clearly, the version of the query using the generalized logic is more elegant, natural and more efficient.

7. INTEGRATION WITH STANDARD OPERATORS

The standard (True/False) Boolean operators and the Extended operators discussed in this paper can be easily integrated into a common interface. Output of the time-sensitive operators can be used as input to the standard operators by mapping the null set Φ to the value False and mapping any non-empty set of intervals to the value True. Conversely, output of the standard operators can be used as input to the time-sensitive operators by simply mapping each False value to the null set Φ and each True value to the universal set $\{\langle 0, NOW\rangle\}$. Time-sensitive comparison operators and their standard counterparts can be similarly integrated into this common interface. As mentioned earlier, constant values and non-temporal attributes are automatically mapped to the universal interval. Thus the standard comparison '$3 < 4$' is equivalent to the extended comparison $\{(3, \langle 0, NOW\rangle)\} <^s \{(4, \langle 0, NOW\rangle)\}$ which returns the universal interval, while the comparison '$13 > 4$' is equivalent to the extended comparison $\{(13, \langle 0, NOW\rangle)\} >^s \{(4, \langle 0, NOW\rangle)\}$ which returns the empty set. Appendix II gives the BNF specifications of the integrated syntax of temporal expressions.

Example 5. This example uses two relations: relation EMP and the new relation, PROJ, defined below.

```
PROJ (title,          /* name of project */
      manager)        /* name of manager */
```

where attribute 'title' is constant while attribute 'manager' is temporal. The following conditions are assumed: a project has only one manager at a time, all employees supervised by the manager of a project work in that project, and not every manager has to be assigned to a project.

The following SQL query gives the titles of the common projects in which employees 'Fred' and 'Mark' have worked together.

```
SELECT      title
FROM        EMP E1 E2, PROJ P
WHERE       (E1.name = 'Fred') AND
            (E2.name = 'Mark')  AND
            ((E1.manager =s E2.manager) AND s
              (E1.manager =s P.manager))
```

To avoid ambiguity in referencing attribute names the above query uses E1 and E2 as aliases for relation EMP and P as an alias for relation PROJ. Notice that both the extended and standard operators are used in the qualification of the above query. If the superscript of the generalized operators are omitted, the above code becomes a valid SQL code that computes the answer of the query when it is applied to a snapshot database (i.e. one which stores only the current snapshot of the real world). Similar statement holds for the other examples and for queries written in relational algebra.

8. CONTRIBUTION AND RELATED WORK

The literature on temporal databases is rich and rapidly growing. There have been numerous research activities to develop and implement temporal/historical query languages. All previous temporal models introduced new operators and/or extended the syntax and semantic of existing query language constructs in order to handle the new time dimension. This is necessary since many temporal queries cannot be expressed in traditional query languages. Even with the new operators/extensions, some of the proposed models do not provide full retrieval power with respect to time. For example, the historical relational algebra for the HRDM model described in Clifford and Croker (1993) is not complete: although the algebra is appealing, Clifford and Croker admit that it is not as powerful as they would like.

The historical algebra of HRDM uses two versions of the select operators: SELECT_IF to choose tuples over their entire lifespan, and SELECT_WHEN to extract a relevant subset of the lifespan of selected tuples. To specify for which times the selection criterion must be

satisfied, existential and universal quantification is introduced. The syntax of the SELECT_IF statement, for example is,

$$\sigma\_\text{IF}_{(A\theta a,Q,L)}$$

where $A\theta a$ is a simple Boolean predicate over the attributes of the tuple and constants, Q is either $\exists$ or $\forall$, and L is the lifespan. For example, the expression

$$\sigma\_\text{IF}_{(c\_home=Tampa,\exists,[2,7])}(\text{EMP})$$

shows which employee lived in Tampas at some time during the period from 2 to 7. This expression can be easily expressed using our extended Boolean operators (see earlier comment on Example 3).

The language TempSQL proposed in Gadia and Nair (1993) is an extension of SQL. TempSQL is three-sorted consisting of relational expressions, temporal expressions and Boolean expressions (returning relations, temporal elements, and True/False, respectively). Temporal and Boolean expressions are used in the SELECT statement: the Boolean expression (WHERE clause) is used as the main selection criterion to select/reject tuples, while the temporal expression (WHILE clause) is used to restrict the time domain of the selected tuples. For example, the query

```
SELECT    title
WHILE     ||EMP.manager = PROJ.manager||
                /* temporal expression */
FROM      EMP,PROJ
WHERE     name = John
              /* Boolean expression   */
```

retrieves the projects in which John worked. Notice that the expression $||A\theta B||$ is used to return the time domain where A and B are/were in θ-relationship. The Boolean expression uses the traditional AND, OR, and NOT operators (denoted $\wedge, \vee$ and $\neg$ in Gadia and Nair, 1993) while the temporal expression uses operators suitable for sets of intervals, e.g. union, intersection, difference. For example, the WHILE clause may contain the temporal expression $||\text{salary} < 24\,000|| \cup ||\text{manager} = \text{Fred}||$ which returns the time intervals when the employee was working under the supervision of Fred and had a salary less than $24 000. It is obvious that much simplicity and clarity for many queries can be obtained if the extended Boolean operators are used; in many queries the WHILE and WHERE clauses can be simple coalesced into a single qualification that uses the extended Boolean logic.

The IXRM model proposed in Lorentzos (1993) uses an interval relational algebra that has the new operators FOLD and UNFOLD (similar to PACK/UNPACK discussed earlier in example 4) in addition to extended versions of the standard relational algebra operators. The selection condition in IXRM is a standard (True/False) Boolean condition in which predefined temporal operators are allowed (e.g. Pure-Subinterval, SubInterval, Pure-SuperInterval, SuperInterval, Overlaps, Precedes, Follows, Adjacent, Common_Points). For example, to retrieve John's salary for times 2 to 7, the following selection is used

$$\sigma_F(\text{EMP})$$

where F is the Boolean condition

(name = John) AND (Time Common_Points:2–7)

The AND operator in the above query is a standard one (returning True or False) and the 'Common-Points' function restricts the time domain to the interval 2 to 7. The comment made earlier regarding the usefulness of the extended Boolean logic to the HRDM model applies also to the IXRM model.

A temporal extension to SQL is TSQL (Navathe and Ahmed, 1993) which is a superset of SQL that introduces new semantic and syntactic components. Among other things, TSQL allows specification of the time domain using the new TIME SLICE clause and the length of a time interval using the new MOVING WINDOW clause. The WHEN clause of TSQL is similar to the WHERE clause of SQL. A number of predefined temporal comparison operators (Before, After, Overlap, During, Adjacent, Follows, Precedes, Equivalent) are allowed in the WHEN clause. For example, to retrieve the salary of John when he worked in project P35, the following TSQL query is used.

```
SELECT    salary
FROM      EMP,PROJ
WHERE     name=John AND EMP.manager=PROJ.manager
                                AND title="P35"
WHEN      EMP.Interval Overlaps PROJ.Interval
```

Notice that the extended Boolean logic allows writing the above query with a single qualification. Notice also that TSQL is based on a tuple timestamping scheme. A single time interval is associated with the entire tuple, rather than with each attribute as in Lorentzos (1993) and Tansel *et al.* (1989). For example, to retrieve the names of those who have experience with (have worked in) the P35 project and who also have served at any time in the Orlando Office, the following TSQL query is needed (note: two aliases are used for relation EMP).

```
SELECT  name
FROM    EMP E1 E1, PROJ
WHERE   E1.name=E2.name AND E1 c_office=Orlando AND
        E2.manager=PROJ.manager AND PROJ.title=P35
WHEN    E2.Interval Overlaps PROJ.Interval
```

With attribute timestamping, our extended Boolean logic can be used to replace the WHERE and WHEN clause of the above query by the following single qualification

```
(EMP.c_office = Orlando) AND
(EMP.manager =" PROJ.manager AND" PROJ.title =" P35)
```

Historical SQL is another superset of SQL proposed in Sarda (1993). The language uses interval comparison operators (Overlap, Contains, Meets, Adjacent, Precedes) which return a Boolean result. Two new unary

operators are introduced: the EXPAND operation converts the interval-stamped tuples of the relation into instant-stamped tuples by replicating the tuples for each instant included in their intervals; the COALESCE operation performs the inverse operation, i.e. it combines tuples having the same attribute values but with consecutive or overlapping time intervals into a single tuple. The two new operators are similar to the FOLD/UNFOLD used in Lorentzos (1993) and the PACK/UNPACK operators used in Clifford and Tansel (1985). Like TSQL, Historical SQL uses tuple time stamping and the same remark on the above TSQL query applies.

In summary, the extended Boolean operators presented in this paper allow the smooth integration of Boolean and temporal expressions. These operators do not increase the retrieval power of existing query languages and are therefore not meant to replace the new (or non-standard) operators used in existing temporal/historical query languages. In addition to allowing many queries to be expressed more easily and naturally, the extended operators are amenable to efficient software and hardware implementations. The implementation algorithms given in this paper are new and differ in nature and flavor from other implementations in the area of temporal databases. Most of these latter implementations are restricted to software and deal primarily with other issues, e.g. indexing techniques for historical databases, temporal query processing, differential query processing and join optimization, system catalogs in temporal databases, storage structures, concurrency control and recovery. The implementation methods discussed in our paper also differ from the few implementations of the new operators reported in literature (e.g. algorithms to implement FOLD/UNFOLD in Lorentzos, 1993). It is important to notice in this context that many of the time-oriented operators/functions used in exisiting temporal languages discussed above are simple or straightforward. For example the following operators defined in Navathe and Ahmed (1993) return Boolean results and their operands are single intervals:

```
[a,b] Before [c,d]    iff   b < c
[a,b] After [c,d]     iff   a > d
[a,b] Overlaps [c,d]  iff   (a ⩽ d) and (c ⩽ b)
[a,b] Precedes [c,d]  iff   c−b = 1
```

The implementation of the above operands follows directly from their definition.

8. CONCLUSION

In this paper, we have generalized the Boolean and comparison operators by allowing their input and output operands to be sets of intervals. Efficient implementation of the generalized approach were presented. Both software and hardware implementations were considered. The application of the proposed generalization to query languages was illustrated by examples.

REFERENCES

Ahn, I. (1986) Towards an implementation of database management systems with temporal support. *PROC. IEEE Conf. on Data Engineering*, pp. 374–381.

Bassiouni, M. (1988) A logic for handling time in temporal databases. *Proc. IEEE COMPSAC Conf.*, pp. 345–352.

Bassiouni, M. and Llewellyn, M. (1992) A relational-calculus query language for historical databases. *J. Comp. Languages*, **17**, 185–197.

Bassiouni, M., Llewellyn, M. and Mukherjee, A. (1993) Time-based operators for relational algebra query languages. *J. Comp. Languages*, **19**, 261–276.

Clifford, J. and Croker, A. (1993) The historical relational data model (HRDM) revisited. In Tansul, A., Clifford, J., Gadia, S., Jajodia, A., Segev, A. and Snodgrass, R. (eds), *Temporal Databases*, pp. 6–27, Benjamin/Cummings,

Clifford, J. and Tansel, A. (1985) On an algebra for historical relational databases: two views. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 247–265,

Gadia, S. (1986) Towards a multihomogeneous model for a temporal database. *Proc. IEEE Conf. on Data Engineering*, pp. 390–397,

Gadia, S. and Nair, S. (1993) Temporal databases: a prelude to parametric data. In Tansul, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A. and Snodgrass, R. (eds), *Temporal Databases*, pp. 28–66. Benjamin/Cummings,

Gadia, S. and Yeung, C. (1988) A generalized model for a relational temporal database. *ACM SIGMOD* 251–259.

Llewellyn, M. and Bassiouni, M. (1991) Historical database views. *J. Inform. Software Technol.*, **33**, 113–120.

Lorentzos, N. (1993) The interval-extended relational model and its applications to valid-time databases. In Tansul, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A. and Snodgrass, R. (eds), *Temporal Databases*, pp. 67–91. Benjamin/Cummings,

McKenzie, E. and Snodgrass, R. (1987) Extending the relational algebra to support transaction time. *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 467–478.

Navathe, S. and Ahmed, R. (1989) A temporal relational model and a query language. *Int. J. Inform. Sci.*, 147–175.

Navathe, S. and Ahmed, R. (1993) Temporal extensions to the relational model and SQL. In Tansul, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A. and Snodgrass, R. (eds), *Temporal Databases*, pp. 92–109. Benjamin/Cummngs.

Sarda, N. (1993) HSQL: a historical query language. In Tansul, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A. and Snodgrass, R. (eds), *Temporal Databases*, pp. 110–140, Benjamin/Cummings.

Sarda, N. (1990) Extensions to SQL for historical databases. *IEEE Trans. Knowledge and Data Eng.*, **2**, 220–230.

Snodgrass, R. (1987) The temporal query language TQUEL. *ACM Trans. Database Sys.*, **12**, 247–298.

Snodgrass, R. and Ahn, I. (1986) Temporal databases. *Computer (IEEE)*, **19**, 35–42.

Tansel, A., Arkun, M. and Ozsoyoglu, G. (1989) Time-by-example query language for historical databases. *IEEE Trans. Software Eng.* **15**, 464–478.

APPENDIX I: HARDWARE (VLSI) IMPLEMENTATION

Figure AI.1 gives a high level schematic diagram of the hardware to implement Method B_AND. The hardware consists of three logical components. The Interval Processor is responsible for sorting (step 1 in method

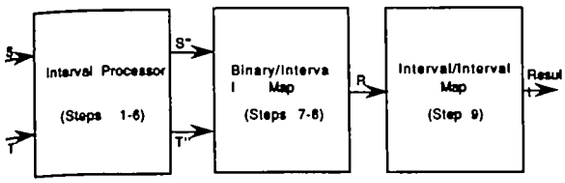

FIGURE AI.1. Schematic diagram of hardware for method B_AND.

B_AND), searching (steps 2, 3 and 5) and finally producing the two binary vectors S″ and T″ (step 6). VLSI schemes for sorting and searching have been extensively investigated. Numerous high speed VLSI circuits for sorting and searching have been designed and fabricated. Other operations needed in the hardware of the Interval Processor are quite straightforward and are amenable to efficient VLSI designs. The binary/interval map corresponds to steps 7 and 8 of method B_AND and is explained below in more details. Finally, the interval/interval map implements step 9 and produces the final result.

Figure AI.2 shows a functional diagram of a VLSI circuit that implements the binary/interval map (steps 7 and 8 of the above method). Specifically, this circuit performs the logical bitwise AND of the two binary vectors S″ and T″ and then computes the corresponding set of intervals R. Using the same values of the previous examples, if the input to this circuit is the two binary strings

$$S'' = \text{'0111101110111011'}$$

$$T'' = \text{'1101110111011110'}$$

the intermediate bitwise AND operations produces the binary vector

$$V = \text{'0101100110011010'}$$

and the circuit generates the output sequence

$$\langle 2,2\rangle, \langle 4,5\rangle, \langle 8,9\rangle, \langle 12,13\rangle, \langle 15,15\rangle$$

which corresponds to the set R of step 8. The method we used to compute this output is to scan the vector V from left to right (as it is generated bit-serially), incrementing a counter at each step (i.e. every clock pulse) and performing the appropriate action whenever the conditions stated below are satisfied:

1. Action 1: if at time t, the bit value is 1 and at time $t-1$, the bit value was 0, then start a new interval whose left endpoint has the value t.
2. Action 2: if at time t, the bit value is 0 and at time $t-1$, the bit value was 1, then close the current interval with a finish endpoint equal to $t-1$.

The implementation of the above actions is incorporated in the circuit of Figure AI.2.

Operation. Initially, the output wires of 'Previous Count', 'Current Count' and the 'Unit Delay' are set to the value 0. Each clock pulse first transfers the contents of 'Current Count' to 'Previous Count' and then increments the contents of 'Current Count' by 1 (a two-phase clock Φ_1, Φ_2 is therefore required).

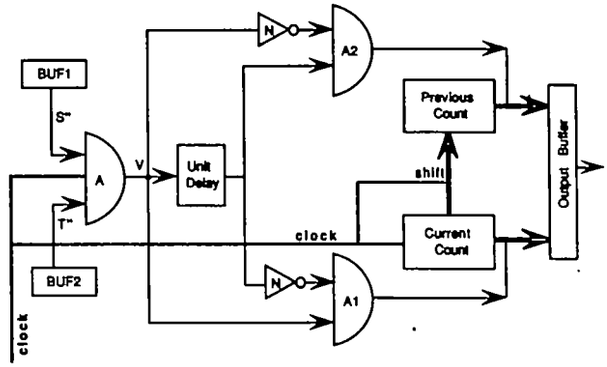

FIGURE AI.2. Circuit for steps 7 and 8.

The AND gate A computes the bitwise AND of the two input operands S″ and T″. The computation is done bit-serially producing one bit of the vector V at each clock. The gates A1 and A2 implement the logic described in Action 1 and Action 2, respectively. For example, the output of gate A1 is the value of the logical AND of the current bit of vector V and the inverse of the previous bit of V. When the output of A1 is 1, the contents of Current Count is transferred to Output Buffer and is treated as the start point of the new interval. Similarly, when the output of gate A2 is 1, the contents of Previous Count is transferred to Output Buffer and is treated as the finish point of the current interval.

APPENDIX II. BNF SPECIFICATION OF THE GENERALIZED OPERATORS

This appendix gives the BNF description for the syntax of the extended time-based operators integrated with the standard operators. We concentrate only on the new aspects of the extension and do not formally define the obvious non-terminals (such as ⟨constant⟩, ⟨variable id⟩, etc.).

```
<temporal expr> ::= <temporal expr> <or op>
                    <temporal term> | <temporal term>
<temporal term> ::= <temporal term> <and op>
             <temporal factor> | <temporal factor>
<temporal factor> ::= <not op> <temporal primary>
                                | <temporal primary>
<temporal primary> ::= '('<temporal expr>')'
                     | <temporal comparison>
<temporal comparison> ::= <c_operand>
          <temporal comp op> <c_operand>
<and op> ::= AND^s | AND
<or op> ::= OR^s | OR
<not op> ::= NOT^s | NOT
<temporal comp op> ::= <sensitive comp op>
                     | <standard comp op>
<sensitive comp op> ::= =^s | <^s | >^s | ≠^s | ≤^s | ≥^s
<standard comp op> ::= = | < | > | ≠ | ≤ | ≥
<c_operand> ::= <constant> | <variable id>
                     |<built-in function>
```