EVENT-JOIN OPTIMIZATION IN TEMPORAL RELATIONAL DATABASES

Arie Segev and Himawan Gunadhi

School of Business Administration, University of California
and
Computer Science Research Department, Lawrence Berkeley Laboratory
Berkeley, CA 94720

ABSTRACT

An Event-Join combines temporal join and outerjoin properties into a single operation. It is mostly used to group temporal attributes of an entity into a single relation. In this paper, we motivate the need to support the efficient processing of event-joins, and introduce several optimization algorithms, both for a general data organization and for specialized organizations (sorted and append-only databases). For the append-only database we introduce a data structure that can improve the performance of event-joins as well as other queries. Finally, we evaluate the performance of the proposed algorithms.

1. INTRODUCTION AND MOTIVATION

Temporal data models are designed to capture the complexities of many time-dependent phenomena, something that traditional approaches, like the relational model, were not intended to do. Many new operators are needed in order to exploit the full potential of temporal data models in enhancing the retrieval power of a database management system (DBMS). Many temporal operators have been introduced in the literature, (e.g. [Clifford & Tansel 85, Adiba & Quang 86, Clifford & Croker 87, Snodgrass 87]), yet with few exceptions, (e.g., [Lum et al 84, Rotem & Segev 87, Snodgrass & Ahn 87]), the issues of performance and optimization have not received as much attention. In a previous paper [Gunadhi & Segev 88], we identified a set of temporal joins and carried out preliminary investigation into their optimization.

In this paper, we study the optimization of *event−join* operations. The event-join operator was first introduced by [Segev & Shoshani 88a]; it is unique in that it combines temporal join and outerjoin components into a single operation. It is used primarily to group temporal attributes of an entity into a single relation; temporal attributes belonging to the same entity, but which are not synchronous in their event points, are likely to be stored in separate relations. Many queries require that they be grouped together as one relation, but differences in their behavior over time brings up the possibility that *null* values are involved in the operands and the join result.

This paper deals with optimizing event-joins in temporal relational databases. Its contributions are the following:

- Motivating and demonstrating the need to support the efficient processing of event-joins.
- As traditional processing cannot support event-joins, we have developed optimization algorithms for various situations, including static sorted databases, and dynamic databases with general data organization and append-only organization.
- In the context of the append-only database, we have developed a new data structures called the *AP*-Tree (*Append−Only* Tree). This tree is a variation of an *ISAM* and a B^+-tree combination, and is useful for other temporal queries besides event- joins.
- We compare the proposed algorithms by evaluating their costs and present some computational results.

The paper is organized as follows: in the next section, we discuss the relational representation of temporal data. In section 3, the event-join operator is defined and explained through an an example. Section 4 explores the optimization of event-joins for data that is sorted and data in a generalized setting; an algorithm for each is described in this section. Section 5 deals with the third main type of data: append-only databases, for which we propose two algorithms to optimize the event-join operator for such a database. The *AP*-tree is introduced in section 6. Section 7 presents an analysis of the costs and relative performance of the four algorithms developed. Conclusions and directions for further research are given in section 8.

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

2. RELATIONAL REPRESENTATION OF TEMPORAL DATA

A convenient way to look at temporal data is through the concepts of *Time Sequences* (*TS*) and *Time Sequence Collection* (*TSC*) [Segev & Shoshani 87]. A *TS* represents a history of a temporal attribute(s) associated with a particular instance of an entity or a relationship. The entity or the relationship are identified by a surrogate (or equivalently, the *time-invariant key* [Navathe & Ahmed 86]). For example, the salary history of an employee is a *TS*. In this paper, we are concerned with two types -- *stepwise constant* and *discrete*. Stepwise constant (*SWC*) data represents a state variable whose values are determined by events and remains the same between events; the salary attribute represents *SWC* data. Discrete data represents an attribute of the event itself, e.g. number of items sold. Time sequences of the same surrogate and attribute types can be grouped into a time sequence collection (*TSC*), e.g. the salary history of all employees forms a *TSC*. There are various ways to represent temporal data in the relational model; detailed discussion can be found in [Segev & Shoshani 88a]. In this paper we assume a representation as shown in Table 1.

MANAGER	E#	MGR	T_S	T_E
	E1	TOM	1	5
	E1	MARK	9	12
	E1	JAY	13	20
	E2	RON	1	18
	E3	RON	1	20

COMMISSION	E#	C_RATE	T_S	T_E
	E1	10%	2	7
	E1	12%	8	20
	E2	8%	2	7
	E2	10%	8	20

Table 1: Representing SWC Data with Lifespan = [1, 20]

We use the terms *surrogate*, *temporal attribute*, and *time attribute* when referring to attributes of a relation. For example, in Table 1, the surrogate of the MANAGER relation † is E#, MGR is a temporal attribute, and T_S and T_E are time attributes. We assume that all relations are in first temporal normal form (1TNF) [Segev & Shoshani 88a].

† We refer to the data construct as a 'relation', but we mean a 'temporal relation'. It is different from a standard relation because of the associated meta-data.

3. EVENT JOINS

An *Event-Join* groups several temporal attributes of an entity into a single relation. This operation is extremely important because due to normalization, temporal attributes are likely to reside in separate relations. To illustrate this point, consider an employee relation in a conventional database. If the database is normalized we are likely to find all the attributes of the employee entity in a single relation. If we now define a subset of the attributes to be temporal (e.g., salary, job-code, manager, commission-rate, etc.) and they are stored in a single relation, a tuple will be created whenever an event affects at least one of those attributes. Consequently, grouping temporal attributes into a single relation should be done if their event points are synchronized. Regardless of the nature of temporal attributes, however, a physical database design may lead to storing the temporal attributes of a given entity in several relations. The analogy in a conventional database is that the database designer may create 3NF tables, but obviously, the user is allowed to join them and create an unnormalized result.

Let $r_i(R_i)$ be a relation on scheme $R_i = \{S_i, A_{i1}, ..., A_{im}, T_S, T_E\}$. In many instances we illustrate the concepts using a single temporal attribute, that is, $m = 1$; all apply to any $m > 1$. Also, when the two surrogate *types* S_i of R_i and S_j of R_j are the same, we simply use S. Instances of surrogate S are denoted by $s1, s2, \cdots$. We use x_i to refer to an arbitrary tuple of r_i; $x_i(A)$ is the value of attribute A in tuple x_i. In order to describe the event-join between r_1 and r_2, we first present two basic operations *TE-JOIN* and *TE-OUTERJOIN*. TE-JOIN is the temporal equivalent of a standard equijoin; two tuples $x_1 \in r_1$ and $x_2 \in r_2$ are concatenated † if their join attribute's values are equal and the intersection of their time intervals is non-empty; the T_S and T_E of the result tuple correspond to the intersection interval. Semantically, this join condition is "where the join values are equal at the same time". In the case of event-joins, we are concerned only with a special case of TE-JOINs where the joining attribute is the surrogate. A TE-OUTERJOIN is a directional operation from r_1 to r_2 (or vice versa). For a given tuple $x_1 \in r_1$, outerjoin tuples are generated for all points $t \in [x_1(T_S), x_1(T_E)]$ where there does not exist $x_2 \in r_2$ such that

† It is not a standard concatenation since only one pair of T_S and T_E is part of the result tuples.

$x_2(S) = x_1(S)$ and $t \in [x_2(T_S), x_2(T_E)]$. Note that all consecutive points t that satisfy the above condition generate a single outerjoin tuple. Using those operations the event-join, r_1 EVENT-JOIN r_2, is done as: templ $\leftarrow r_1$ TE-JOIN r_2 on S; temp2 $\leftarrow r_1$ TE-OUTERJOIN r_2 on S; temp3 $\leftarrow r_2$ TE-OUTERJOIN r_1 on S; result $\leftarrow$ templ $\cup$ temp2 $\cup$ temp3. Table 2 shows the result of an event-join performed between the MANAGER and COMMISSION relations of Table 1.

result	E#	MGR	C_RATE	T_S	T_E
	E1	TOM	∅	1	1
	E1	TOM	10%	2	5
	E1	∅	10%	6	7
	E1	∅	12%	8	8
	E1	MARK	12%	9	12
	E1	JAY	12%	13	20
	E2	RON	∅	1	1
	E2	RON	8%	2	7
	E2	RON	10%	8	18
	E2	∅	10%	19	20
	E3	RON	∅	1	20

Table 2: Result of Event-Join

The most troublesome components of the event-join are the outer-joins. The situation is further complicated by the time interval predicate associated with the TE-outerjoin, preventing the usage of non-temporal outerjoin procedures [Rosenthal & Reiner 84, Dayal 87]. An easy solution that comes to mind is to store all non-existence tuples explicitly, e.g., tuples like (1, ∅, 6, 8) are added to the MANAGER relation of Table 1. In that case the outerjoin components disappear, and the problem reduces to a TE-JOIN on S. Unfortunately, there are many situations where such a 'fix' will degrade overall performance rather than improve it. For example, if the whole S_i domain is represented in relation r_i, representing all non-existence data explicitly will in the worst case double the size of the table (this is the case of alternating state transitions between existence and non-existence). A much worse problem may arise when a relation contains only a fraction of the S-domain values, e.g., if on the average, only 5% of the employees of a large corporation earn commissions, adding to the non-existence data for the 95% other employees to the commission relation will add to storage cost, querying cost (including event-joins), and maintenance of the commission relation and any of its associated secondary indexes. Consequently, we divide event-joins into two types -- 'easy' and 'difficult'. Easy cases are those where the relations contain explicit tuples for all non-existence data and are sorted by (S, T_S) (the sorted case is detailed in the next section). Other cases are regarded difficult. In the remainder of the paper we are mostly concerned with the difficult cases.

4. EVENT-JOIN OPTIMIZATION

In this section we discuss the optimization of event-joins where the relations are either sorted or unsorted. Before we proceed with details of the algorithms, the important concept of tuple covering, which is used throughout the discussions, is presented first.

4.1. Concept of Tuple Covering

We first introduce the notion of *covering* which is used in all the event-join algorithms. To illustrate the concept, consider the example of Table 3.

r_1	r_2	Cover of x_1	Modified x_1
$s1,a,5,15$	$s1,b,1,2$	*None*	$s1,a,5,15$
	$s1,c,3,7$	$s1,a,c,5,7$	$s1,a,8,15$
	$s1,d,9,12$	$s1,a,∅,8,8$ $s1,a,d,9,12$	$s1,a,13,15$
	$s1,e,16,20$	$s1,a,∅,13,15$	*Full cover*

Table 3: Example of Tuple Covering

Relation r_1 has a scheme $R_1 = (S, A_1, T_S, T_E)$ and a single tuple <$s1, a, 5, 15$>. r_2 has a scheme $R_2 = (S, A_2, T_S, T_E)$ and four tuples as shown in the table. During the event-join, $x_1 \in r_1$ has to be compared with tuples $x_2 \in r_2$; assume that the order of comparisons is as shown in the table (top-down). A tuple x_2 contributes to the covering of x_1 if one or two result tuples $\{x_1(S), x_1(A_1), x_2(A_2), I_C\}$ can be derived, where $I_C \subseteq [x_1(T_S), x_1(T_E)]$. I_C can be viewed as a covered portion of x_1. The 'modified x_1' column in the table represents the uncovered portion of x_1. Note that in the covering process we have relied on the ordering of r_2 by time in deriving the outerjoin tuples (those with $x_2(A_2) = ∅$). Also, the covering column of the table contains only a subset of the final result since the covering of r_2's tuples is incomplete. The remaining result tuples should be derived from a TE-outerjoin from r_2 to r_1. In this particular example, the remaining result tuples are <$s1, ∅, b, 1, 2$>, <$s1, ∅, c, 3, 4$> and <$s1, ∅, e, 16, 20$>.

Determining and maintaining the information about the covered portion of a tuple is substantially different if the relations are not sorted by T_S. In the sorted case we can determine outerjoin tuples as the scanning progresses and the information about the covered portion of the tuple is maintained by simply modifying its T_S. In the general case, the covered subintervals can be encountered in a random order; moreover, an outerjoin result tuple associated with $x_1 \in r_1$ can be determined only when the scanning of r_2 is complete. We first present an algorithm for the case where r_1 and r_2 are sorted by S (primary order) and by T_S (secondary order). In the next subsection we discuss the general case. As can be seen from the above example, the particular values of A_1 and A_2 are immaterial as far as the logic of the event-join is concerned; we are only interested in existence or non-existence of these attributes. Consequently, in the remainder of the paper, whenever convenient, we use examples with relation schemas of (S_i, T_S, T_E), but the reader should keep in mind that at least one A_i attribute is part of the actual tuples. Also, the algorithms presented in this paper involve lots of housekeeping details. For lack of space we omit the details and provide only an outline of the algorithms. The logic of all algorithms is described ignoring blocking of tuples; it is trivially extended to handle blocking.

4.2. Event-Join Sort-Merge Algorithm

The *Sort–Merge* algorithm processes the event-join by taking advantage of the fact that both relations are in sort order. Unlike a conventional relation which requires only primary key order for sorting, the temporal relation needs to be sorted on S as the primary order and T_S as the secondary order. The event-join sort-merge algorithm, which will be referred to as Algorithm One, scans each relation just once in order to produce the result relation. At each iteration, two tuples (possibly with modified T_S), $x_1 \in r_1$ and $x_2 \in r_2$, are compared to each other and one or two result tuples will be produced based on the relationship between the tuples on their surrogate values and time intervals.

The first comparison in Algorithm One is on the surrogate value -- if they are unequal, it means that the tuple with the lower S value, say x_1, does not have any matching surrogates in the other relation; this implies that x_1 is fully covered, an outerjoin result tuple is generated, and the next x_1 tuple is read. If on the other hand $x_1(S) = x_2(S)$, there are many possible relationships that can exist between the time intervals of the two tuples; but there are just three distinct possibilities in terms of result tuples that have to be generated. The three cases are identified in Step 3 of Algorithm One.

Algorithm One

(1). Read x_1 and x_2. Repeat steps 2 to 4 until End-of-File (EOF). If EOF occurred for r_i, generate outerjoin tuples for the remainder of r_j's tuples (including the current tuple if not fully covered).

(2). If $x_i(S) < x_j(S)$, generate an outerjoin result tuple for x_i.

(3). For the situation where $x_1(S) = x_2(S)$, there are three cases to consider.

Case 1: $x_i(T_S) = x_j(T_S)$. Write an intersection result tuple.

Case 2: $x_i(T_S) < x_j(T_S)$ and $x_i(T_E) \ge x_j(T_S)$. Write one outerjoin tuple for x_i and one intersection tuple. Modify x_1 and x_2 and read next tuple(s).

Case 3: $x_i(T_E) < x_j(T_S)$. Write an outerjoin tuple for x_i.

(4). Modify x_1 and x_2 and read next tuple(s).

The next tuple of r_i is read only when the current tuple has been fully covered. Note that whenever we use the subscripts i and j in Algorithm One, $i = 1$ and $j = 2$ or $i = 2$ and $j = 1$. Also an intersection result tuple is equivalent to a TE-JOIN result tuple.

4.3. Event-Join Nested-Loop Algorithm

The Nested-Loop method described below does not assume any kind of ordering among the tuples in either relation. The event-join is achieved in two stages, the first of which is nested-loop with r_1 and r_2 being the inner and outer relations respectively. Tuples produced in the first stage are the result of either intersections or outerjoins from r_1 to r_2. In the second stage, the order of relations are now reversed for another nested-loop, but the only result tuples created here will be outerjoins from r_2 to r_1.

Unlike the sorted case, maintaining the information about the covered portion of x_i's time interval cannot be done by simply modifying T_S, and the following procedure is followed. In the first nested-loop, whenever a tuple x_1 from r_1 is first read, a list U is initialized with the pair of time-stamps associated with x_1. This list corresponds to the uncovered portions of x_1. For each tuple x_2, the algorithm applies the test of equality on the surrogate values and a non-null intersection over time. The second condition is needed because if two tuples share a common surrogate value but are disjoint over time, no conclusion can be derived (in contrast to the sorted case) as to whether an outerjoin is appropriate, unless the EOF for r_2 has been reached. Thus, while scanning r_2, the covering of x_1 is achieved only through interval intersections, and for each x_2, at most one intersection result tuple will be produced. Once this is accomplished, the uncovered subintervals associated with x_1 are determined, and appropriate outerjoin result tuples are generated. At the end of r_2's scan the interval of x_1 will either be completely covered, has one uncovered segment, or at most two segments. For each uncovered segment, the time pair representing them are inserted into U *in place* of the original entry. This ensures that U remains an *ordered* list; the ordering within U helps the search for the appropriate interval that is relevant for a TE-JOIN in subsequent iterations through r_2. Regardless of the number of entries in the list, any tuple x_2 can only intersect with *one* entry, otherwise it would mean that there are two or more tuples in r_2 having the same surrogate value and overlap in time. This implies that the condition of 1TNF has not been satisfied.

Unlike conventional nested-loop procedures, we need not retrieve all the tuples of the outer relation, since an empty U indicates that the original x_1 has been fully covered. In the event that the loop terminates because the end of file r_2 is reached, either the whole, or parts of x_1's time interval were left uncovered. An outerjoin result tuple is generated from each time pair in U; the time pair determines the time-start and time-end of the result tuple.

The second nested-loop differs from the first in that it produces only outerjoin tuples from r_2. Thus no result tuple duplicating a tuple already produced in the first stage is created. In order to reduce the number of unnecessary scans of r_i, the Algorithm uses a *hash-filter* [Bloom 70] created during the first stage as follows: when r_2 is scanned, each time an x_2 is found that participates in a TE-JOIN, the hash-filter is updated for that tuple. The hash-filter maintains H bits to represent N_{r_2} tuples, where $H \le N_{r_2}$. The hash-filter entries corresponding to $h(x_2)$, where h is the hash-function, are initialized to 0, and whenever an x_2 generates an intersection result tuple for the current x_1, $h(x_2)$ is set to 1. This table is kept in main memory, and in the best case scenario where there is sufficient memory to maintain one bit per tuple, the hash function is the count of x_2 tuples already accessed, and the table is a one dimensional array indexed by this count.

During the second stage, for each tuple in the inner relation r_2, if it hashes to a value of 0, then an outerjoin tuple is produced without scanning r_i. Otherwise, as in the first nested-loop, we carry out the same updates on the coverage of x_2, although no intersection tuples are produced. As before, outerjoin tuples are produced when it can be determined that no x_1 exists to cover the current x_2. Below we outline the steps of the algorithm, labeled as Algorithm Two. U_i denotes the list U for x_i, i =1, 2.

Algorithm Two

(1). [*Nested-Loop-1*] For each tuple in r_1: read r_2 and execute Step 2 until EOF for r_2 or x_1 is fully covered. If EOF for r_2, produce outerjoin tuples for x_1 based on U_1.

(2). If $x_1(S) = x_2(S)$ and the two time intervals intersect, then do: write an intersection result tuple. Update U_1. Set hash-filter entry for x_2 to 1.

(3). [*Nested-Loop-2*] For each tuple x_2 of r_2: if hash-filter bit = 0 produce outerjoin tuple immediately, and read next x_2. Otherwise read r_1 and execute Step 4 until EOF for r_1 or x_2 is fully covered.

(4). if $x_2(S) = x_1(S)$ and the two time intervals intersect then update U_2.

In the case of having space for a second bit for each of r_2's tuples, Algorithm Two can be further improved if a second filter is used. During the first stage, while covering x_1 it is possible that the time interval of x_2 contains that of x_1. In that case we set the corresponding filter entry to 1. Then, in Step 3 we also avoid the scan of r_1 if the first filter bit is 1 and the second filter bit is also 1.

5. APPEND-ONLY DATABASES

In the case of static history databases, one can store the data sorted by (S, T_S) and then apply Algorithm One; this provides the maximum efficiency for event-joins. For a dynamic temporal database, it may be too inefficient to keep the data sorted by (S, T_S), and consequently, either the operands are sorted prior to the application of Algorithm One, or Algorithm Two is used. If the database is append-only, the event-join algorithms can utilize this fact to enhance their efficiency.

There are several variations of append-only databases, some of which are not 'truly' append-only. As far as event-joins are concerned we view a database to be append-only if tuples are inserted at the end of the file and in order of the events that generated them. The tuples can have open-end or closed-end time intervals. To illustrate these points, consider Figure 1 that shows the time sequences for three surrogate instances with life-spans of $[1, NOW]$; each event point (X in the figure) corresponds to the generation of a new tuple for the surrogate (we are not concerned with the values of the temporal attributes). Let relation r_i represent that data; the states of that relation (for $t \ge 10$) are shown in Table 4. Note that such data is inappropriate for a WORM device since insertions also cause updates; for example, the event at time 10 led to updating $(s3, 1, NOW)$ to $(s3, 1, 19)$ and appending the tuple $(s3, 20, NOW)$. If the representation of the data in this example uses time points instead of time intervals, it would be truly append-only.

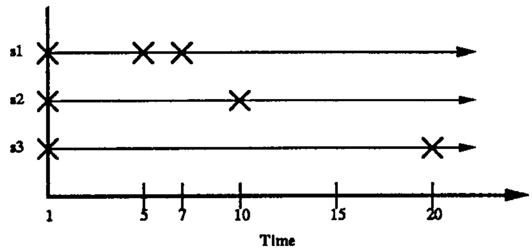

Figure 1: Time Sequences for 3 Surrogates with Lifespans = [1, *NOW*]

Snapshot at Time	State of r_i: $\{S_i, T_S, T_E\}$ (A_i is omitted)
$10 \le t < 20$	$s3, 1, NOW$ $s1, 5, 6$ $s1, 7, NOW$ $s2, 10, NOW$
$20 \le t <$ Next Event Pt	$s1, 1, 4$ $s2, 1, 9$ $s3, 1, 19$ $s1, 5, 6$ $s1, 7, NOW$ $s2, 10, NOW$ $s3, 20, NOW$

Table 4: Progression of an Append-Only Database States

Deletions in append-only temporal databases are significantly different than in conventional databases. In our case, they are storage management activities rather than user transactions. From a logical point of view deletions are a result of a change in the lifespan †, i.e. an increase in the value of *LS.START*. An example is a 'moving-window' lifespan $[NOW - l, NOW]$ where l is the length of history. In the case of step-wise constant sequences, deletion of data to reflect the new lifespan is not guaranteed to be contiguous; Table 5 illustrates this issue. The table shows the state of r_i at $t = 21$ (reproduced from Table 4) and the effect of changing the lifespan at $t = 22$ from $[1, NOW]$ to $[7, NOW]$. As can be seen from the table a new lifespan can cause updates and deletions at any point in the file. Although this example used open-end time intervals, the same problem occurs for any step-wise constant data regardless of its representation. It also demonstrates that maintaining the lifespan for an active database with small time granularity on a real-time basis can be prohibitively expensive. Fortunately, these updates and deletions can be done periodically without affecting the logical view of the data, that is, the physical lifespan can be different than the logical lifespan provided that the first contains the latter. For discrete data, the situation is much simpler and implementing a change in the lifespan can be done by simply updating a begin-of-file pointer to the first tuple whose time value is greater than or equal to the new *LS.START*.

† We use *LS.START* and *LS.END* to refer to the boundary points of the lifespan.

State of r_i: $\{S_i, T_S, T_E\}$		
Tuple #	LS = $[1, NOW]$ $t = 21$	LS = $[7, NOW]$ $t = 22$
1	$s1, 1, 4$	*deleted*
2	$s2, 1, 9$	$s2, 7, 9$
3	$s3, 1, 19$	$s3, 7, 19$
4	$s1, 5, 6$	*deleted*
5	$s1, 7, NOW$	$s1, 7, NOW$
6	$s2, 10, NOW$	$s2, 10, NOW$
7	$s3, 20, NOW$	$s3, 20, NOW$

Table 5: Effect of Modifying the Lifespan of r_i at $t = 22$

If r_i is an append-only relation the order of its tuples corresponds to the order of their events, thus, they are ordered by T_S. Unfortunately, the event-join needs the primary order to be by S, and the surrogate instances of r_i can be in an arbitrary order. Nevertheless, we can take advantage of the ordering by T_S. We assume that if retroactive corrections to the history are necessary, they are done in batch mode offline and the file is reorganized to preserve the T_S-order; this is a reasonable course of action in most environments where the normal mode of operation is not error-correction. Another solution is to use an overflow area to store the 'correction records'; if their number is small (relative to the data file) they will not affect the performance of the event-join algorithms.

We present two event-join algorithms in this section. The first algorithm, stated as Algorithm Three below, follows the logic of the Nested-Loop algorithms, but is different in two important ways. First, when x_1 is compared against tuples of r_2 we do not necessarily have to complete r_2's scan -- since r_2 is append-only it follows that x_1 is fully covered if $x_1(S_1) = x_2(S_2)$ and x_2 fully covers x_1, or if $x_1(S_1) \neq x_2(S_2)$ and $x_2(T_S) > x_1(T_E)$. Second, as in the sorted case, the covered portions of x_1 are always contiguous and thus we can maintain that information by updating $x_1(T_S)$ as was done in Algorithm One. Unlike the sorted case we cannot write outerjoin tuples for x_2 when r_2 is scanned to cover x_1 (see Step 3 of Algorithm Three).

> **Algorithm Three**
>
> (1). [*Nested-Loop-1*] For each x_1: read r_2 and execute Step 2 until x_1 is fully covered or EOF for r_2 is reached. If EOF, generate outerjoin tuple for x_1.
>
> (2). There are four cases to consider in this step.
>
> *Case 1*: $x_1(T_S) > x_2(T_E)$ -- no result tuple is generated.
>
> *Case 2*: $x_1(S) \neq x_2(S)$ and $x_2(T_S) > x_1(T_E)$ -- generate an outerjoin tuple for x_1.
>
> *Case 3*: $x_1(S) \neq x_2(S)$ and $x_2(T_S) \leq x_1(T_E)$ -- no result tuple is generated.
>
> *Case 4*: $x_1(S) = x_2(S)$ and $x_1(T_S) \leq x_2(T_E)$ -- do Step 3.
>
> (3). Execute Step 3 of Algorithm One, except that no outerjoin tuple is written for x_2 if $x_2(T_S) < x_1(T_S)$, and the hash filter is updated whenever the time intervals of x_1 and x_2 intersect.
>
> (4). [*Nested-Loop-2*] The procedure is similar to Steps 1 to 3, except: (i) If hash-filter entry for x_2 is 0, produce an outerjoin tuple without scanning r_1; (ii) Do not produce any intersection tuples; (iii) No filter updates occur and on EOF for r_2 the algorithm stops.

The second algorithm, stated as Algorithm Four below, avoids the final outerjoin from r_2 to r_1 by writing updated time-intervals for r_2's tuples while they are scanned for each x_1 tuple. This is achieved by creating a copy of r_2 which is updated during the first nested-loop. The benefit of this approach is that the second nested-loop is replaced by a single scan through r_2 in order to determine which tuples require outerjoins where no tuple has been found in r_1 with matching surrogates. The updating procedure for tuples in r_1 and r_2 is similar to that of Algorithm One.

> **Algorithm Four**
>
> (1). Create a working copy of r_2, and call it r_2'.
>
> (2). [*Nested-Loop-1*] Procedure is the same as Steps 1 to 3 of Algorithm Three, except: (i) Step 3 is done exactly as in Algorithm One, that is, we write outerjoin tuples for x_2; (ii) x_2' is updated by writing in place its modified T_S; if x_2' is fully covered, its T_S is set to $T_E + 1$; (iii) No hash-filter is used.
>
> (3). Read r_2' in a single scan, and for those tuples where $T_S \leq T_E$, produce an outerjoin result tuple.

Note that Step 1 of Algorithm Four can be done while scanning r_2 for the first x_1 tuple; subsequent x_1 tuples scan r_2'. Both of the above algorithms contains a nested-loop component to cover x_1 tuples by scanning r_2. This component is the most expensive part of the algorithms, and reducing the number of r_2's tuples scanned for each x_1 is very important. The append-only property helps in achieving that objective but we may further improve the performance by using a secondary index as described in the next section.

6. THE APPEND-ONLY TREE

Let r_1 and r_2 be append-only relations. We use a second subscript x_i whenever we need to identify specific tuples, that is, x_{ij} is the tuple x_i in location j (note that there is a one-to-one correspondence between tuple number and location number). We know that if $j_1 > j_2$, then $x_{ij_1}(T_S) \geq x_{ij_2}(T_S)$. Let x_1 be an arbitrary tuple of r_1 and assume we know the location of $x_{2\bar{j}}$, where $\bar{j}$ is the j that attains $\max_j\{x_{2j}(T_S) | x_{2j}(T_S) \leq x_1(T_E) \text{ and } x_2(S_2) = x_1(S_1)\}$. Then, we can start a backward scan of r_2 from location $\bar{j}$ until x_1 is covered. Location $\bar{j}$ can be identified using an index on (S, T_S). Such an index, however, if not available to support other queries, may be too expensive for a dynamic database. In this section we describe an index on T_S which is far cheaper to maintain compared to an S or (S, T_S) index. The index, referred to as *AP*-tree (Append-only Tree), can be viewed as a form of a load-only B^+-tree. Since the index points to records based on T_S, we omit the requirement that $x_{2\bar{j}}(S_2) = x_1(S_1)$, and thus start from the tuple which has the desired T_S and is the farthest (towards the end of the file). Figure 2 illustrates the process of covering x_1 when the *AP*-tree is used. As a specific example, consider the tuples of relation r_i in Table 4 at $t \geq 20$. Let a tuple of r_j be $(s1, 6, 7)$. To cover this tuple, only tuples of r_i with $T_S \leq 7$ should be examined. If we use an *AP*-tree, the tuple $(s1, 7, NOW)$ of r_i can be accessed directly, and following a backward scan the latest tuple to be read is $(s1, 5, 6)$. Without the index, we would have to scan r_i from the beginning and read 5 tuples (compared to two tuples with the index). In deciding whether or not to use the index, the cost of accessing it should also be taken into consideration. Using the index may be beneficial since the worst case of the backward scan is processing all the way to the beginning of the relation, e.g. if the first tuple of r_i in the above example would have been $(s1, 1, NOW)$. The main property that affects the usefulness of the index is the uniformity of event rate among surrogates of the outer relation.

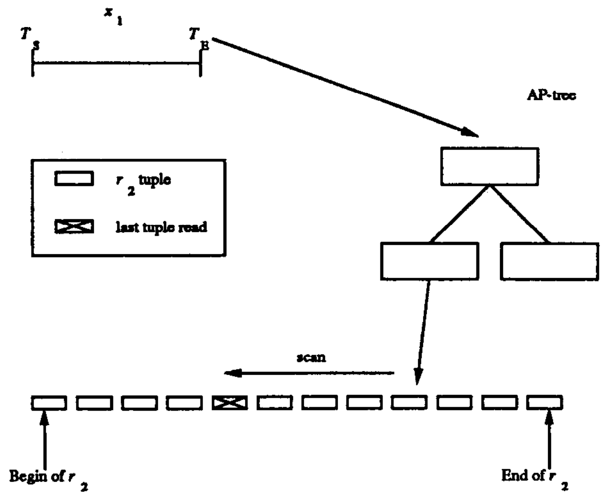

Figure 2: Covering Tuple x_1 Using AP-tree

Note that a uniform rate of events for an outer relation r_2 does not imply that the AP-tree need not be used for all $x_1 \in r_1$. Those x_1 tuples closer to the beginning of the file may benefit more from a forward scan. Currently, if the event rate is not uniform among the surrogates of r_2, an $x_1 \in r_1$ is likely to benefit from using the AP-tree if $x_1(S_1)$ is a very active surrogate in both r_1 and r_2.

We will now describe the basics of the AP-tree (more details can be found in [Gunadhi & Segev 89]). An AP-tree indexing r_i on T_S is shown in Figure 3. This tree is a hybrid of an *ISAM* index and a B^+-tree. The leaves of the tree contain all the T_S values in r_i; for each T_S value, the leaf points to the last (towards the end of the file) tuple with the specific T_S value. For example, tuples numbers 7, 8 and 9 have $T_S = 10$ (those tuples must belong to different surrogates since a given surrogate cannot have two tuples with the same T_S). Each non-leaf node indexes nodes at the next level. Note that the pointer associated with a non-leaf key value points to a node at the next level having this key value as the smallest node value. The significance of this decision is explained later on. Access to the tree is either through the root or through the right-most leaf. The AP-tree is different than the B^+-tree in several respects. First, if the tree is of degree $2d$, there is no constraint that a node must have at least d keys. Second, there is no node splitting when a node gets full. Third, the online maintenance of the tree is done by accessing the right-most leaf. Given the premise that deletions are treated as offline † storage management, only the right-hand side of the tree can be affected. The only online transactions that affects the T_S values in an append-only database is appending a new tuple. In most cases, just the right-most leaf is affected, either a pointer is updated or a new key-pointer pair is added, but if it is full a new leaf has to be created to its right, and in the worst case nodes are added along the path from the root to the right-most node and a new root node has to be created. In Figure 3 we show the effect of new tuples on the tree.

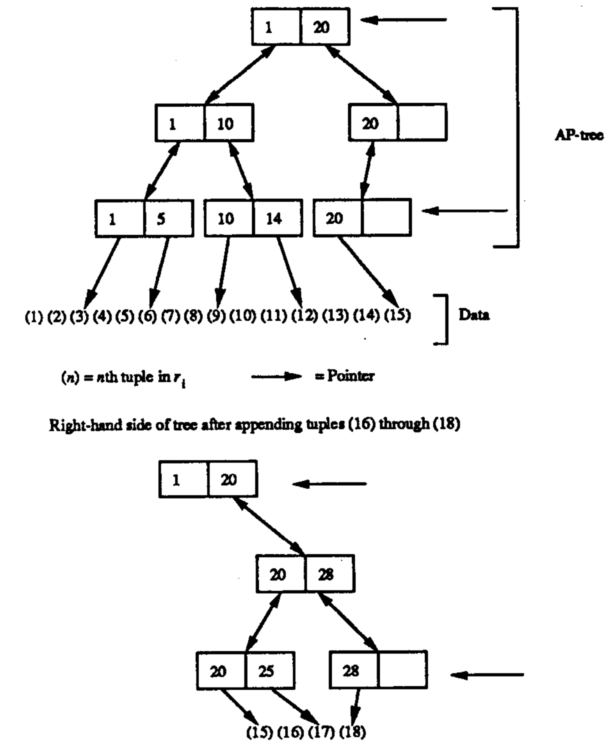

Figure 3: Example of AP-tree Before and After an Insertion

In the case of event-joins, an AP-tree on r_2 is used in the process of covering $x_1 \in r_1$. Therefore, we need to get to the leaf node pointing to x_{2j}. The following procedure is adopted (v is a key value):

Procedure AP

1 Start at the root of AP-tree.

2 For each node visited, follow the pointer corresponding to $v^+ = \max\{v \mid v \le x_1(T_E)\}$.

† Reorganizing the tree to reflect deletions can be done during idle periods or low load periods. All the procedures function correctly regardless of the timing; the only issue is performance.

Several notes are in order. The fact that non-leaf nodes index lower level nodes based on the smallest rather than the largest key value assures that only one leaf node is visited, and that the maintenance of the tree is significantly cheaper than when the indexing is based on the smallest value. In step 2, we assumed that a v^+ exists. It is easy to see that a v^+ exists for all nodes except possibly for nodes on the path from the root to the left-most node. This case can be identified prior to accessing the *AP*-tree and thus prevents unnecessary index search.

7. COST ANALYSIS

In this section, we analyze the costs of the four algorithms. The following definitions are needed. W_{r_i} is the width (bytes) for each tuple in r_i. N_{r_i} is the number of tuples in r_i. B is the page size (bytes). P_{r_i} is the number of pages used for $r_i = \lceil (N_{r_i} \times W_{r_i}) / B \rceil$. Variables W_{r_i}, N_{r_i} and P_{r_i} apply to the operand relations and also r_{EJ}, which denotes the relation resulting from the event-join. M is the size (pages) of main memory available for an algorithm. $C_i(j)$ is the cost in disk I/Os of step j of algorithm i. α_i is the percentage of tuples in r_i that produce outerjoin tuples in r_{EJ}. β_i is the selectivity of the hash-filter on the tuples of r_i that require outerjoins. Finally, γ_i measures the average scan length through relation r_i when r_j is the inner relation for Algorithm Two, and γ_i' does the same for Algorithms Three and Four.

7.1. Algorithm One Costs

If the two relations are already sorted, the cost is $P_{r_1} + P_{r_2} + P_{EJ}$, which is the disk I/O time to join the two relations. For the case where the data need to be sorted first, each relation r_i is first sorted into F_{r_i} files, each M pages in size, where F_{r_i} is the number of files needed for the sort, and is equal to $\lceil P_{r_i} / M \rceil$. The F_{r_i} files are then merged together, and the total cost for the sorting/merging is $2(MF_{r_i} + P_{r_i})$. We are assuming that (1) $P_{r_i} \le M$, and (2) the system allows F_{r_i} files to be opened simultaneously. If one or both of these assumptions are unsatisfied, the I/O costs will be greater. The total cost, $C_1(total)$, is $P_{r_1} + P_{r_2} + P_{r_{EJ}}$ if the relations are sorted, and $2M(F_{r_1} + F_{r_2}) + 3(P_{r_1} + P_{r_2}) + P_{EJ}$, if sorting is required.

7.2. Algorithm Two Costs

Assume that the hash-filter is kept in main memory and maintains one bit per tuple. This means that the selectivity factor β_i represents the portion of tuples in r_i with no matching surrogate values to be found in r_j. Take r_1 as the inner relation in the first nested-loop procedure. We present the cost of the algorithm in terms of its two nested-loop procedures which we label here as NL1 and NL2. $C_2(NL1) = P_{r_1} + \lceil (1-\alpha_2)N_{EJ} / B \rceil + \gamma_2(1-\alpha_1)\lceil P_{r_1} / M \rceil P_{r_2} + \alpha_1 \lceil P_{r_1} / M \rceil P_{r_2}$. The first term represents the cost of reading in r_1, the second term is the number of pages of result tuples written, the third term reflects the average number of reads in order to produce result tuples where x_1 is fully covered by r_2, and finally the last component is the cost of producing outerjoin tuples for r_1, which requires complete iteration through r_2 for every M pages of r_1.

As for NL2, $C_2(NL2) = P_{r_2} + \lceil \alpha_2 N_{EJ} / B \rceil + \alpha_2\beta_2 \lceil P_{r_2} / M \rceil + \gamma_1(1-\alpha_2)\lceil P_{r_2} / M \rceil P_{r_1} + \alpha_2(1-\beta_2)\lceil P_{r_2} / M \rceil P_{r_1}$. The first two components are the one time read cost of r_2 and the write cost for the outerjoin result tuples for r_2; the third subexpression is the cost of producing the outerjoin tuples with the help of the hash-filter; the fourth is the average cost of reads over the outer relation to determine that r_2 tuples are fully covered; and the last item is the cost of exhaustive search related to producing outerjoin tuples. The total cost is $C_2(total) = C_2(NL1) + C_2(NL2)$.

7.3. Algorithm Three Costs

For the first case of the append-only nested-loop, the hash filter is also employed; thus we assume that one bit per tuple is used. The difference in cost between Algorithms Three and Two are:(1) outerjoins can be performed on average as cheaply as covered tuples in terms of disk reads for Algorithm Three; (2) the average length of a scan through the outer relation, γ_i', is likely to be better than the γ_i of Algorithm Two, since there is a clustering of tuples on T_S. As before, $C_3(total) = C_3(NL1) + C_3(NL2)$. For the first nested-loop, $C_3(NL1) = P_{r_1} + \lceil (1-\alpha_2)N_{EJ} / B \rceil + \gamma_2' \lceil P_{r_1} / M \rceil P_{r_2}$, where the second expression denotes the cost of iterating through r_2. For the second nested-loop, $C_3(NL2) = P_{r_2} + \lceil \alpha_2 N_{EJ} / B \rceil + \gamma_1'(1-\alpha_2\beta_2)\lceil P_{r_2} / M \rceil P_{r_1} + \alpha_2\beta_2 \lceil P_{r_2} / M \rceil P_{r_1}$.

7.4. Algorithm Four Costs

The final algorithm differs further from the previous two nested-loop algorithms. The second part of the algorithm needs only a single scan through r_2. Although a temporary file needs to be created, it can be done during the first iteration through r_2 in order to save I/Os. Thus the total cost expression is: $C_4(total) = [P_{r_1} + 2P_{r_2}] + 2\left\lceil(1 - \alpha_2\beta_2)N_{EJ} / B\right\rceil + \left\lceil\alpha_2\beta_2 N_{EJ} / B\right\rceil + \left\lceil P_{r_1} / M\right\rceil \gamma_i' P_{r_2}$. The first expression (in brackets), represent the total cost of reading in the relations when they are in the inner loop, plus the additional overhead of creating r_2'. The second component is the write cost of event-join tuples during the first loop plus the cost of updating r_2'. The third component is the cost of generating the outerjoin result tuples during the second nested-loop. The fourth term is the cost of scanning through r_2 to produce the other result tuples.

7.5. Comparisons Among Algorithms

It is clear that Algorithm One is superior if the relations are already sorted. Also, the append-only algorithms dominate the algorithm for the general case. The interesting question is whether the relations, if not sorted, should be sorted, and then processed by Algorithm One. Figure 4 shows some preliminary results. It should be noted that we have assumed favorable conditions for the sorting, e.g., no limit on the number of files that can be opened simultaneously during a sort-merge execution; if this is not the case, the results will make Algorithms Three and Four more attractive.

Figure 4 shows the total I/O cost of the algorithms as a function of γ_i. We set the other parameters to be equal, i.e. $P_{r_i} = 100{,}000$ pages, $P_{r_{EJ}} = 200{,}000$ pages, $\alpha_i = 0.1$, and $\beta_i = 0.5$. Additionally, we assumed that γ_i' is equal to γ_i. γ_i measures the percentage of blocks in the relation that have to be scanned. The graph in Figure 4(a) shows the performance of all four methods when γ_i was varied between 0.001 to 0.01. It shows that Algorithm Two does worst among the algorithms, while Algorithm Four's efficiency increases as the scan length gets shorter. It is better than Algorithm One at approximately $\gamma_i = 0.001$. Note that γ_i may be much more selective than 0.001 for an append-only database, since measured in disk I/Os, 0.001 is 100 blocks, which is still a large number. Figure 4 (b) highlights just the three best algorithms, so that a better comparison can be made at lower values of γ_i. The values of the above parameters reflect the filter selectivity and the number of tuples scanned for each inner relation tuple. The results validate our conjecture that one can do better than sorting in the append-only environment.

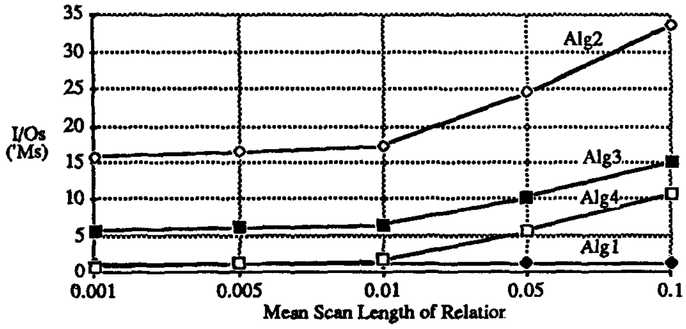

Figure 4(a)

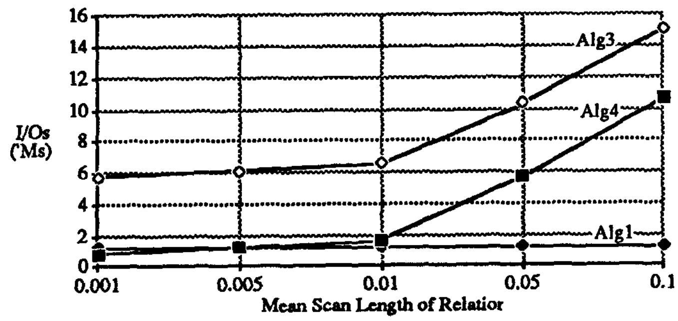

Figure 4(b)

Figure 4: Comparisons of Algorithms Over Gamma

8. Summary and Future Research

In this paper, we have addressed the problem of optimizing event-joins in a temporal relational database. Event-joins are important because normalization considerations are likely to split the temporal attributes of an entity among several relations. The event-join combines a temporal equijoin component and a temporal outerjoin component. Unlike a conventional outerjoin, the temporal counterpart consists of two asymmetric outerjoins, a fact that complicates its optimization. The complexity of processing event-join strategies depends on the nature of the data, its organization, and whether or not all non-existing data are represented explicitly. We addressed three cases of data organization; these are (in increasing order of complexity) data sorted by surrogate and time, append-only, and general optimization. For the sorted case (appropriate for static databases), the processing of an event-join is the most efficient since each relation has to be read only once. The append-only database is an appropriate organization for many dynamic temporal databases and an event-join algorithm can take advantage of the time ordering. For the append-only case we have introduced the AP-Tree, which is used to reduce the cost of scanning an outer relation in a nested-loop procedure.

In section 7, we have presented a cost analysis of the proposed algorithms. The algorithm for the sorted case (Alg. One) obviously dominates all others. The append-only algorithms (Algs. Three & Four) dominate the general nested-loop algorithm (Alg. Two); this is also expected. The interesting questions are whether, for the non-sorted case, the data should be sorted and then processed by Algorithm One. For the general case, the answer is yes (under the favorable sorting conditions that we assumed). For the append-only case the answer is dependent on the selectivity of the filter and the number of tuples scanned for each inner-loop tuple. Also, if the inner relation is significantly smaller than the outer relation, and the selectivity factors associated with the append-only algorithms are small, sorting will be less favorable. We are currently working on a comprehensive simulation test to validate our initial findings.

Finally, it should be noted that many of the concepts presented in this paper are applicable to other temporal queries; in particular other joins since the concept of covering is also relevant to them. In current and future research we try to devise more elaborate rules on when to use the AP-Tree. Also, as evident from the cost equations, estimation of several parameters are required.

References

Adiba, M, Quang, N.B., Historical Multi-Media Databases, *Proc. Int. Conf. on VLDB*, pp. 63-70, 1986.

Bloom, B.H., Space/Time Trade-offs in Hash Coding with Allowable Errors, *Comm. of the ACM*, 13, 7, 1970.

Clifford, J., Croker, A., The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans, *Proc. Int. Conf. on Data Engineering*, pp. 528-537, 1987.

Clifford, J., Tansel, A., On an Algebra for Historical Relational Databases: Two Views, *Proc. ACM SIGMOD Int. Conf. on Mgt. of Data*, pp. 247-265, 1985.

Dayal, U., Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers, *Pro. Int. Conf on VLDB*, pp.197-208, 1987.

Gunadhi, H., Segev, A. A Framework for Query Optimization in Temporal Databases, Lawrence Berkeley Lab Technical Report LBL-26417, 1988.

Gunadhi, H., Segev, A. Indexing Structures for Temporal Database, Lawrence Berkeley Lab Technical Report, 1989.

Lu, H., Carey, M.J., Some Experimental Results on Distributed Join Algorithms in a Local Network, *Proc. Int. Conf. on VLDB*, 1985.

Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., Woodfill, J., Designing DBMS Support for the Temporal Dimension, *Proc. ACM SIGMOD Int. Conf. on Mgt. of Data*, pp. 115-130, 1984.

Mackert, L.F., Lohman, G.M., "R" Validation and Performance Evaluation for Local Queries, *Proc. ACM SIGMOD Int. Conf. on Mgt. of Data*, pp. 84-95, 1986.

Navathe, S., Ahmed, R., A Temporal Relational Model and a Query Language, UF-CIS Technical Report TR-85-16, Univ. of Florida, 1986.

Rosenthal, A., Reiner, D., Extending the Algebraic Framework of Query Processing to Handle Outerjoins *Proc. Int. Conf. on VLDB*, pp. 334-343, 1984.

Rotem, D., Segev, A., Physical Organization of Temporal Data, *Proc. Int. Conf. on Data Engineering*, pp. 547-553, 1987.

Segev, A., Shoshani, A., Logical Modeling of Temporal Databases, *Proc. ACM SIGMOD Int. Conf. on Mgt. of Data*, pp. 454-466, 1987.

Segev, A., and Shoshani, A., The Representation of a Temporal Data Model in the Relational Environment, *Lecture Notes in Computer Science*, Vol. 339, M. Rafanelli, J.C. Klensin, and P. Svensson (eds.), Springer-Verlag, pp. 39-61, 1988a.

Segev, A., Shoshani, A., Functionality of Temporal Data Models and Physical Design Implementations, *IEEE Data Engineering*, 11, 4, pp. 38-45, 1988b.

Snodgrass, R., The Temporal Query Language TQuel, *ACM Transactions on Database Systems*, pp. 247-298, 1987.

Snodgrass, R., Ahn, I., Performance Analysis of Temporal Queries, TempIS Document No. 17, Dept. of Comp. Sci., Univ. of North Carolina, 1987.