ALGORITHMS FOR EVENT – JOIN OPTIMIZATION FOR EFFICIENT PERORMANCE & AN ACCESS STRUCTURE FOR TEMPORAL DATA

[1]Mir Abid Ali, [2]Dr. Syed Abdul Sattar

[1]Research Scholar, Department of Computer Science Engineering, PP.COMP.SCI.0578, Rayalaseema University, A.P, India.

[2] Director, Research & Development, Nawab Shah Alam Khan College of Engineering & Technology, Hyderabad.

Abstract: An Event-Join consolidates temporal join and external join properties into a single operation. It is, for the most part, used to group transient characteristics of an element into a single relation. This paper motivates the need to support the efficient processing of event-joins, & introduce several optimization algorithms, both for a general information association and for specific associations (arranged and annex just databases). For the affix just information base we present an information structure that can enhance the execution of event-joins and additionally different questions we depict another ordering system, the time index, for enhancing the performance of specific classes of temporal queries. The time index can be used to recover versions of objects that are valid during a specific time period. It supports the preparing of the temporal When administrator and temporal aggregate functions efficiently. The time indexing scheme is additionally reached out to enhance the execution of the temporal SELECT administrator, which recovers objects that fulfill a specific condition amid an explicit era. We will portray the ordering system and its search and insertion algorithms.

***IndexTerms* - temporal join, temporal queries, time index, database, event-join .**

I. 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. Numerous new administrators are required with the end goal to abuse the maximum capacity of fleeting information models in upgrading the recovery intensity of a database the executives' framework (DBMS). Numerous fleeting administrators have been presented in the writing, (e.g. [Clifford and Tansel 4, Adiba and Quang 1, Clifford and Croker 3, Snodgrass 6]), yet with couple of special cases (e.g., tLurn et al 10, Rotem & Segev 7, Snodgrass & Ahn 6]), the issues of performance and optimization have not received as much attention. In an earlier paper [Gunadhi & Segev 11, 12], A set of temporal joins are identified and carried out initial investigation into their optimization. This paper explains the study of optimization of event-join operations. It was first introduced by [Segev & Shoshani 9]; it is unique in that it cartels temporal join and outer join components into a single operation. It is used primarily to group temporal attributes of an entity into a single relation; temporal attributes be appropriate to the same entity, but which are not synchronous in their event points, are probable to be stored in separate relations.

Numerous inquiries require that they are gathered together as one connection, however, contrasts in their conduct after some time raises the likelihood that invalid qualities are engaged with the operands and the join result.

It deals with streamlining occasion participates in fleeting social databases. Its commitments are the accompanying: a. Rousing and showing the need to support the proficient preparing of event-joins. As traditional handling can't bolster event-joins, we have created optimization algorithms for different circumstances, including static arranged databases and dynamic databases with general information organization and attach just association.

With regards to the append-only database, we have built up another information structures called the AP-Tree (Append-Only Tree). This tree is a variety of an ISAM and a B+-tree blend and is valuable for other transient inquiries other than an event- joins.

We consider our time index to be a fundamental indexing technique for temporal data. It very well may be joined with a regular credit ordering plan to productively process worldly choices and transient join operations.

II. RELATIONAL REPRESENTATION OF TEMPORAL DATA

A suitable way to look at temporal data is through the perceptions of Time Sequence Collection (T_X) and Time Sequences (T_S) [Segev & Shoshani 9]. A T_S represents a history of a temporal attribute(s) associated with a particular instance of an entity or a relationship.

Table-1 Representing SWC Data with Lifespans = [1,20]

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

In this paper, we are fretful with two types -- stepwise constant and discrete.

Stepwise constant (SWC) data speaks to a state variable whose qualities are controlled by events and continues as before between events: the pay trait represents to SWC information. Discrete data speaks to a trait of the event itself, e.g. number of things sold. Time sequences of a similar surrogate and quality sorts can be assembled into a time sequence collection (T_X), e.g. the history of the salary of all employee forms a TSC.

III EVENT JOINS

Event-Join group's numerous temporal attributes of an entity into a single relation. This activity is critical in light of the fact that because of standardization, temporal attributes is probably going to reside in separate relations. To explain this point, an employee relation is considering in a conventional database. If the database is normal, we are likely to find all the attributes of the employee entity in a single relation. If we now define temporal as a subset of the attributes (e.g., salary, job code, manager, commission-rate, etc.) and they are put away in a single relation, a tuple will be made at whatever point an occasion influences something like one of those attributes. Thusly, gathering temporal attributes into a single relation ought to be done if their event points are synchronized. Despite the idea of temporal attributes, in any case, a physical database design may prompt putting away the temporal attribute8 of a given element in a few relations. The similarity in a conventional database is that the database creator may make 3NF tables, however clearly, the client is permitted to join them and make a unnormalized outcome

Let $r_i(R_i)$ be a relation on scheme $R_i = \{S_i, A_{i1}, \ldots, A_{im}, T_S, T_E\}$. 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 $x_2(S) = x_1(S)$ 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: temp1 $\leftarrow r_1$ 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 $\leftarrow$ temp1 $\cup$ temp2 $\cup$ temp3. Table 2 shows the result of an event-join performed between the MANAGER and COMMISSION relations of Table 1.

Table 2: Results of Event-Joint

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

The most troublesome segments of the event- join are the external joins. The circumstance is additionally confused when time interval predicate related with the TE-external join, keeping the use of non-fleeting external join methods [Rosenthal and Reiner 14, Dayal 13]. A simple arrangement that rings a bell is to store all non-presence tuples explicitly, e.g., tuples like (1, Ø, 6,8) are added to the MANAGER connection of Table 1. All things considered, the external join parts vanish, and the issue decreases to a TE: JOIN on S. Unfortunately, there are numerous circumstances where such a 'fix' will debase by and large execution as opposed to enhancing it. For instance, if the entire Si domain is spoken to in connection RI, speaking to all non-presence information unequivocally will in the most pessimistic scenario twofold the extent of the table (this is the situation of rotating state changes among presence and non-presence). A much more awful issue may emerge when a connection contains just a small amount of the S-domain values, e.g., if, on the normal, just 5% of the workers of an extensive organization win commissions, adding to the non-presence information for the 95% different representatives to the commission connection will add to capacity cost, questioning cost (counting event joins), and upkeep of the commission relation and any of its related optional records. Thus, we partition divide event-joins into two sorts - 'simple' and 'troublesome'. Simple cases are those where the relations contain explicit tuples for all non-presence information and are arranged by (S, 7's) (the arranged case is point by point in the following segment). Different cases are respected troublesome. In the rest of the paper, we are for the most part worried about the difficult cases.

IV. EVENT- JOIN OPTIMIZATION

Optimizations of event-joins were discussed in this section where the relations are either sorted or unsorted. Before we ensue with details of the algorithms, the significant concept of tuple covering, which is used throughout the discussions, is presented first.

4.1. The 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.

Table 3 Example of Tuple Covering

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,Ø,8,8 s1,a,d,9,12	s1,a,13,15
	s1,e,16,20	s1,a,Ø,13,15	*Full cover*

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 $x_2(A_2) = \emptyset$). 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 <$s1, \emptyset, b, 1, 2$>, <$s1, \emptyset, c, 3, 4$> and <$s1, \emptyset, e, 16, 20$>.

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 outer-join 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 outer-join result tuple associated with x 1 E rl can be determined only when the scanning of r2 is complete. We first present an algorithm for the case where r1 and r2 are sorted by S (primary order) and by T_s (second order). In the next subsection, we discuss the general case. As can be seen from the above example, the particular values of A1 and A2 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 (Si, T_s, TE), but the reader should keep in mind that at least ON Ai 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 the primary key order for sorting, the temporal relation needs to be sorted on S as the primary order and T_s as the second 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), x1 E r1 and x2 E r2, 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 xl, does not have any matching surrogates in the other relation; this implies that x1 is fully covered, an outer-join result tuple is generated, and the next x1 tuple is read. If on the other hand x1(S) = x2(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 $x_i(T_E) \geq x_j(T_S)$. 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=l. 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 r1 and r2 being the inner and outer relations respectively. Tuples produced in the first stage are the result of either intersections or outer joins from r1 to r2. In the second stage, the order of relations are now reversed for another nested-loop, but the only result tuples created here will be outer joins from r2 to r1.

Unlike the sorted case, maintaining the information about the covered portion Of Xi's time interval can't be done by simply modifying T_s, and the following procedure is followed. In the first nested-loop, whenever a tuple x1 from r1 is first to read a list U is initialized with the pair of time-stamps associated with x1. This list corresponds to the uncovered portions of x1. For each tuple x2, 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 outer- join is appropriate unless the EOF for r2 has been reached. Thus, while scanning r2, the covering of x1 is achieved only through interval intersections, and for each x2, at most one intersection result tuple will be produced. Once this is accomplished, the uncovered subintervals associated with x1 are determined, and appropriate outer join result tuples are generated. At the end of r2's scan, the interval of x1 will either be completely covered, has one uncovered segment, or at most two segments. For each uncovered segment, the time pair's 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 r2. Regardless of the number of entries in the list, any tuple x2 can only intersect with one entry, otherwise, it would mean that there are two or more tuples in r2 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 xi has been fully covered. In the event that the loop terminates because the end of file r2 is reached, either the whole or parts of xi's time interval were left uncovered. An outer join 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 outer join tuples from r2. 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 ri, the Algorithm uses a hash-filter [Bloom 2] created during the first stage as follows: when r2 is scanned, each time an x2 is found that participates in a TE-JOIN, the hash-filter is updated for that tuple. The hash-filter maintains H bits to represent $_{Nr2}$ tuples, where $H<=N_{r2}$. The hash-filter entries corresponding to h(x2), where h is the hash- function, are initialized to 0, and whenever an x2 generates an intersection result tuple for the current x1, h (~9 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 x2 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 r2, if it hashes to a value of 0, then an outer join tuple is produced without scanning r1. Otherwise, as in the first nested-loop, we carry out the same updates on the coverage of x2, although no intersection tuples are produced. As before, outer join tuples are produced when it can be determined that no x1 exists to cover the current x2. Below we outline the steps of the algorithm, labeled as Algorithm TWO. U_i denotes the list U for X_i, i =I, 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 r2's tuples, Algorithm Two can be further improved if a second filter is used. During the first stage, while covering x1 it is possible that the time interval of x2 contains that of xi. In that case, we set the corresponding filter entry to 1. Then, in Step 3 we also avoid the scan of r1 if the first filter bit is 1 and the second filter bit is also 1.

V. THE TIME INDEX ACCESS STRUCTURE

In this section, we first give a storage model for temporal data based on the object versioning approach' [SA 15]. The time indexing technique can be adapted to other temporal database proposals, such as time normalization [NA 16] or attribute versioning [GY17]. We use object versioning because it is a simpler approach for storage management, and allows us to concentrate our presentation on the properties of the time index itself. In Section 5.2, we will describe our time index, and provide search, insertion, and deletion algorithms. Sections 5.3 and 5.4 show how the time index may be used to efficiently process the temporal WHEN operator and aggregate functions.

5.1 The Temporal Storage Model

The time dimension is represented, as in [GY17, CW18, Gad19, and others, using the concepts of discrete time points and time intervals. A time interval, denoted by [t1,t2], is defined to be a set of consecutive equidistant time instants (points), where t1 is the first time instant and t2 is the last time instant of the interval. The time dimension is represented as a time interval [0, now], where 0 represents the starting time of our database mini-world application, and now is the current time, which is continuously expanding. The distance between two consecutive time instances can be adjusted based on the granularity of the application to be equal to months, days, hours, minutes, seconds, or any other suitable time unit. A single discrete time point t is easily represented as an interval [t, t] or simply [t].

We will assume an underlying record-based storage system which supports object versioning. Records are used to store versions of objects. In addition to the regular record attributes, Ai, each record will have an interval attribute, called valid-time, consisting of two sub-attributes t_s (valid start time) and t_e (valid end time). The valid-time attribute of an object version is a time interval during which the version is valid. In object versioning, a record r with r.valid-time.t_e = now is considered to be the current version of some object. However, numerous past versions of the object can also exist. We assume that the versions of an object are linked to the current version using one of the basic storage techniques (chaining, clustering, accession list) proposed in [AS88, Lum84]. In addition, we assume that the current version of an object can be efficiently located from any other version; for example, by using a pointer to a linked list header, which in turn points to the current version.

Whenever an object o is updated with new attribute values, the current version, r, becomes the most recent past version, and a new current version T' is created for o. If the valid time of the update is t_u, then the update is executed as follows:

r.valid-time.t_e <- (t_u, - 1) ;
create a new object version r^I by setting r^I <- r ;
for each modified regular attribute Ai
set $r^I.A_i$ <- the new attribute value ;
set r^I.valid-time.t_s <- t_u ;
set r^I.valid-time.t_e <- now;

Such a database is called append only since older object versions are never deleted, so the file of records continually has object versions appended to it. An operation to delete an object o at time t_d is executed as follows:

find the current version r of the object o;
set r.valid_time.t_e <- t_d ;

Finally, an operation to insert an object o at time t_i is executed as follows:

create the initial version T for o ;
set r.valid-time.t_s <- t_i ;
set r.valid-time.t_e <- now ;

Because the append-only nature of such a temporal database will eventually lead to a very large file, we assume that a purge(t_p) operation is available. This operation purges all versions r with r.valid-time.t_e < t_{ps} by moving those versions to some form of archival storage, such as optical disk or magnetic tape.

Name	Dept	Valid_Time
emp1	A	[0, 3]
emp1	B	[4, now]
emp2	B	[0, 5]
emp3	C	[0, 7]
emp3	A	[8, 9]
emp4	C	[2, 3]
emp4	A	[8, now]
emp5	B	[10, now]
emp6	C	[12, now]
emp7	C	[11, now]

The EMPLOYEE table

Dept	Manager	Valid_Time
A	Smith	[0, 3]
A	Thomas	[4, 9]
A	Chang	[10, now]
B	Cannata	[0, 6]
B	Martin	[7, now]
C	Roberto	[0, now]

The DEPARTMENT table

Figure 1 A Temporal Database

5.2 Description of the Time Index

Conventional indexing schemes assume that there is a total ordering on the index search values. The properties of the temporal dimension make it difficult to use traditional indexing techniques for time indexing. First, the index search values, the valid-time attribute, are intervals rather than points. The valid-time intervals of various object versions will overlap in arbitrary ways. Because one cannot define a total ordering on the interval values, a conventional indexing scheme cannot be used. Second, because of the nature of temporal databases, most updates occur in an append mode, since past versions are kept in the database. Hence, deletions of object versions do not generally occur, and insertions of new object versions occur mostly in increasing time value. In addition, the search condition typically specifies the retrieval of versions that are valid during a particular time interval.

A time index is defined over an object versioning record-based storage system, TDB, which consists of a collection of object versions, TDB = {el, e2,....en}, and supports an interval-based search operation. This operation is formally defined as follows.

Given a Search Interval, Ts = [ta, ta], find the following set of versions:

S(Is) = {ej E TDB | (ej.validtime n Is) ≠ 0 }

A simple but inefficient implementation of this search operation is to sequentially access the entire storage system, TDB, using linear search, and to retrieve those records whose valid-time intersects with Is. Such a search will require O(N*M) accesses to the storage system, where N is the number of objects and M is the maximal number of versions per object.

Notice that the interval-based search problem is identical to the k-dimensional spatial search problem, where k = 1. There have been a number of index methods proposed for k-dimensional spatial search [Gut20, OSD21], which are not suitable for the time dimension for the reasons discussed below. These index methods support the spatial search for 2-dimensional objects in CAD or geographical database applications. The algorithms proposed in [Gut20, OSD21] use the concept of a region to index spatial objects. A search space is divided into regions which may overlap with each other. A sub-tree in an index tree contains pointers to all spatial objects located in a region. Since spatial objects can overlap with each other, handling the boundary conditions between regions is quite complex in these algorithms. In temporal databases, there can be a very high degree of overlap between the valid-time intervals of object versions. A large number of long or short intervals can exist at a particular time point. Furthermore, the search space is continuously expanding and most spatial indexing techniques assume a fixed search space. In addition, temporal objects are appended mostly in increasing time value, making it difficult to maintain tree balance for traditional indexing trees. Because of these differences between temporal and spatial search, we do not consider the spatial algorithms in [Gut20, OSD21] to be suitable for temporal data if they are directly adapted from 2-dimensions to a single dimension.

The idea behind our time index is to maintain a set of linearly ordered indexing points on the time dimension. An indexing point is created at the time points where (a) a new interval is started, or (b) the time point immediately after an interval terminates. The set of all indexing points is formally defined as follows:

(PR1) BP = {t_i | €j E TDB ((t_i = ej.validtime.t_s) V (t = ej.validtime.t_e + 1))) U {now}
The concept of indexing points is illustrated in Figure 2 for the temporal data shown in the EMPLOYEE table of Figure 1. In Figure 2, e_{ij} refers to version j of object e_i. There exist 9 indexing points in BP for all employee versions, BP = {0,2,4,6,8,10,11,12,now). Time point 2 is an index point since the version e41 starts at 2. Time point 6 is an index point since e21 terminates at 5. Before proceeding to describe our index structure, we define some additional notation that will be useful in our discussion. Let tj be an arbitrary time point, which may or may not be a point in BP.

Let t_j be an arbitrary time point, which may or may not be a point in BP. We define t_j- (t_j+) to be the point in BP such that t_j- < t_j (t_j < t_j+) and there does not exist a point t_m E BP such that t_j- < t_m < t_j (t_j < t_m, < t_j+). In other words, t_j- (t_j+) is the point in BP that is immediately before (after) t_j. We also define t_{j-} = as follows:

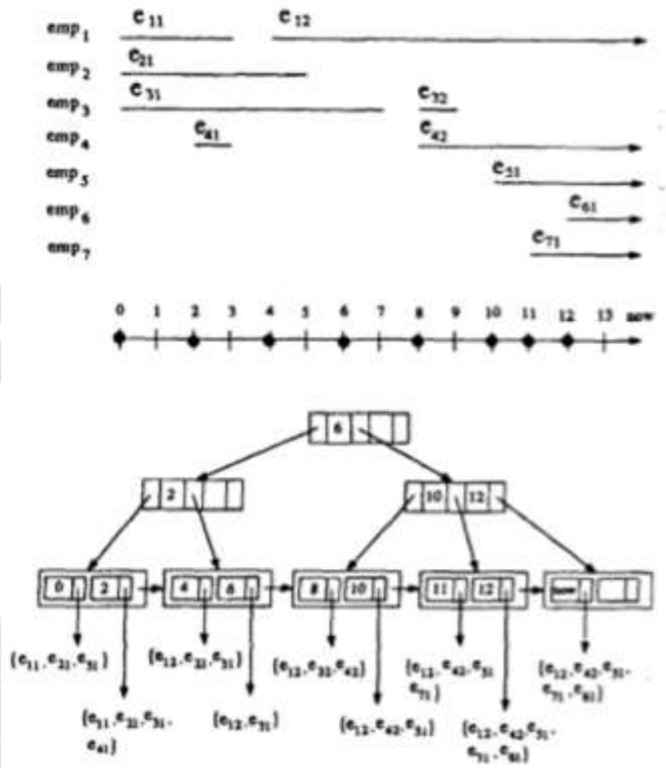

Figure 2 Versions of Employees Object and Time Index

1. If there exists a point t_k E BP such that t_j = t_k, then t_{j-}= = t_k.
2. Otherwise, tj-= = t_i

Since all the indexing points t_i in BP can be totally ordered, we can now use a regular B+tree [Com22, EN23] to index these time points. Each leaf node entry of the B+-tree at point t_s is of the form:

[t_s, bucket]

where the bucket is a pointer to a bucket containing pointers to object versions. Each bucket B(t_i) in our index scheme is maintained such that it contains pointers to

all object versions whose *valid_time* contains the interval $[t_i, t_i^+ - 1]$. Such a property can be formally specified as follows:

(PR2) $B(t_i) = \{e_j \in TDB \mid ([t_i, t_i^+ - 1] \subseteq e_j.valid\_time)\}$

Figure 2 shows a B^+-tree of order 3, which indexes the BP set of points of the EMPLOYEE versions. Each node in the B^+-tree contains at most two search values and three pointers. Consider the leaf entry for search time point 4, for instance; (PR2) indeed holds.

$$B(4) = \{e_{12}, e_{21}, e_{31}\}$$
$$= \{e_j \in TDB \mid ([4,5] \subseteq e_j.valid\_time)\}$$

In a real temporal database, there can be a large number of object versions in each bucket, and many of those may be repeated from the previous bucket. For example, in Figure 2 the object version e12 appears in multiple consecutive buckets. To reduce this redundancy and make the time index more practical, an incremental scheme is used. Rather than keeping a full bucket for each time point entry in BP, we only keep a full bucket for the first entry of each leaf node. Since most versions will continue to be valid during the next indexing interval, we only keep the incremental changes in the buckets of the subsequent entries in a leaf node. For instance, in Figure 3 the entry at point 10 stores {+e31, -e32} in its incremental bucket indicating e31 starts at point 10 and e32 terminates at the point immediately before point 10. Hence, the incremental bucket $B(t_i)$ for a non-leading entry at time point t_i can be computed as follows:

$B(t_i) = B(t_l) \cup (\cup_{t_j \in BP,\, t_i < t_j < t_l} SA(tj)) - (\cup_{t_j \in BP}, t, < tj < ti\ SE(tj))$

Where $B(t_l)$ is the bucket for the leading entry in the leaf node where point t; is located, $SA(t_j)$ is the set of object versions whose start time is t_j and $SE(t_j)$ is the set of object versions whose end time is t_j - 1.

We now describe our search algorithm as follows:

1. Suppose the time search interval is I_s = [ta, tb]. Perform a range search on the B+-tree to find

(C1) PI(Is) = {ti E BP/t_a < ti < t_b) U {t_a-=}

2. Then compute the following set as the result of the algorithm.

(C2) $T(I_s) = \cup_{ti \in PI}\ B(t_i)$

Insertion or deletion of a new object version should maintain the properties (PR1) and (PR2). The algorithms for inserting and deleting an object version ek are shown in Algorithm A.

Note that, in general, version deletion will not occur in append-only databases except for an exception such as correction of an error. It is easy to argue that (PR1) and (PR2) is maintained after each execution of the Insert or Delete operation. We will not show the proof argument here due to the lack of space.

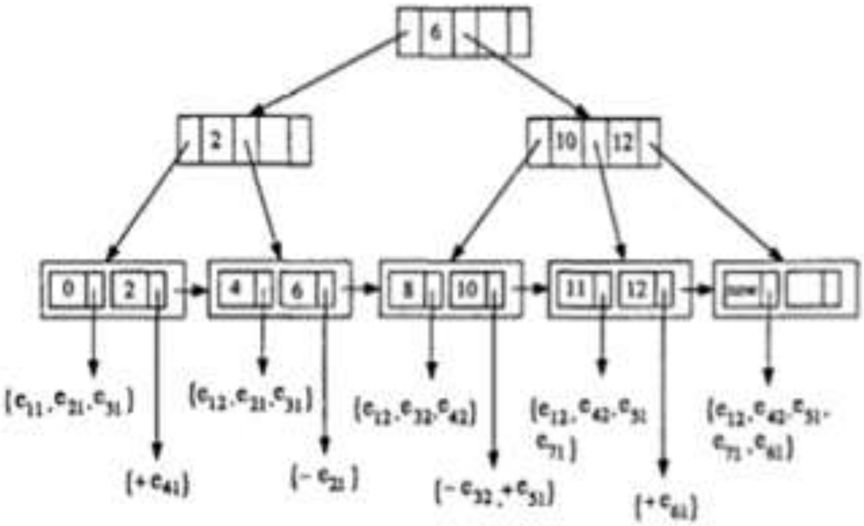

Figure 3 Storing Incremental changes in Time Index Buckets

Algorithm A

```
Insert(e_k)
begin
    t_a ← e_k.valid_time.t_s ;
    t_b ← e_k.valid_time.t_e + 1 ;
    search the B+-tree for t_a;
    if (¬found) then
        insert t_a in the B+-tree;
    if entry at t_a is not a leading entry in a leaf node
        add e_k into SA(t_a);
    search the B+-tree for t_b;
    if (¬found) then
        insert t_b in the B+-tree;
    if entry at t_b is not a leading entry in a leaf node
        add e_k into SE(t_b);
    for each leading entry t_l of a leaf node
    where t_a ≤ t_l < t_b
        add e_k in B(t_l);
end

Delete(e_k)
begin
    t_a ← e_k.valid_time.t_s ;
    t_b ← e_k.valid_time.t_e + 1 ;
    search the B+-tree for t_a;
    if entry at t_a is not a leading entry in a leaf node
      remove e_k from SA(t_a);
    search the B+-tree for t_b;
    if entry at t_b is not a leading entry in a leaf node
      remove e_k from SE(t_b);
    for each leading entry t_l of a leaf node
    where t_a ≤ t_l < t_b
      remove e_k from B(t_l);
end
```

5.3 Using the Time Index for Processing the WHEN Operator

The time index can be used to efficiently process the WHEN operator [GY17] with a constant projection time interval. An example of the type of query is: List the salary history for all employees during the time interval [4, 51]. The result of such a query can be directly retrieved using the time index on the EMPLOYEE object versions shown in Figure 3. We will discuss in Section 3 how an extension to the time index will permit efficient processing of temporal SELECT operations. Notice that a simple query such as the one given above is very expensive to process if there was no index on time.

5.4 Using the Time Index for Processing Aggregate Functions

In this section, we will describe how the time index scheme is used to process aggregate functions at different time points or intervals. In a non-temporal conventional database, the aggregate functions, such. as COUNT, EXISTS, SUM, AVERAGE, MIN, and MAX are applied to sets of objects or attribute values of sets of objects. In temporal databases, an aggregate function is applied to a set of temporal entities over an interval. For instance, the query 'GET COUNT EMPLOYEE: [3, 81' [EW24] should count the number of employees at each time point during the time interval [3, 81]. The result of the temporal COUNT function is a function mapping from each time point in [3, 81 to an integer number that is the number of employees at that time point. For instance, the above query is evaluated to the following result if applied to the database shown in Figure 1:

{ [3] → 4, [4, 5] -> 3, [6, 7] → 2, [8] → 3 }

Our time index can be easily used to process such aggregate functions. Let 1s be the interval over which the temporal aggregate function is evaluated. The query performs a range search to find Pl(Is). Each point in Pl(ls) p represents a point of state change in the database. That is, the database mini-world changes its state at each change point and stays in the same state until the next change point. Therefore the aggregate function only needs to be evaluated for the points in Pl(ls). The query is evaluated by applying the function on the bucket of object versions at each point. If the incremental index shown in Figure 3 is used, the running count from the previous change point is updated at the current change point by adding the number of new versions and subtracting the number of removed versions at the change point. Similar techniques can be used for other aggregate functions that must be computed at various points over a time interval

VI CONCLUSION

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 outer join component. Unlike a conventional outer join, the temporal counterpart consists of two asymmetric outer joins, 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. We described a new indexing technique, the time index, for temporal data. The index is different from regular B+-tree indexes because it is based on objects whose search values are intervals rather than points. We create a set of indexing points based on the starting and ending points of the object intervals and use those points to build an indexing structure. At each indexing point, all object versions that are valid during that point can be retrieved via a bucket of pointers. We used incremental buckets to reduce the bucket sizes. Search, insertion, and deletion algorithms are presented.

Our structure can be used to improve the performance of several important operations associated with temporal databases. These include temporal selection, temporal projection, aggregate functions, and certain temporal joins.

REFERENCES

1. Adiba, M, Quang, N.B., Historical Multi-Media Data- bases, Proc. ht. Co& on VLDB, pp. 63-70, 1986.

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

3. Clifford, J., Croker, A., Historical Relational Data Model (HRDM) and Algebra Based on Lifespans, Proc. ht. Co@. on Data Engineering, pp. 528-537, 1987.

4. Clifford, J., Tansel, A., On an Algebra for Historical Relational Databases: Two Views, Proc. ACM SIG- MOD ht. Co@ on Mgt. of Data, pp. 247-265, 1985

5. I. Ahn. Towards an implementation of database management systems with temporal support. In IEEE Data Engineering Conference, February 1986.

6. I. Ahn and R. Snodgrass. Partitioned storage for temporal databases. Information Systems, 13(4), 1988.

7. Rotem D, Segev. A, Physical design of Temporal Databases. to appear in Proceedings of the Third International Conference on Data Engineering. February. 1987.

8. Snodgrass. R .. The Temporal Query Language TQuel. Proceedings of the Third ACM SIG- .\lOD S.vmposium on Principles of Database SJ'stems (PODS). Waterloo .. Canada. April 1984. pp. 204-213.

9. Arie Segevt and Arie Shoshani, LOGICAL MODELING OF TEMPORAL DATA ,School of Business Administration The University of California Berkeley, CA 04720 +Computer Science Research Department Lawrence Berkeley Laboratory University of California Berkeley, California 04720 March, 1987.

10. Lum, V., Dadam. P, Erbe, R., Guenauer. J, Pistor. P., Walch. G , Werner. H , Woodfill .. J , Designing Dbms Support for the Temporal Dimension. Proceedings of the ACM SIGMOD International Conference on Management of Data. June 1984. pp. 115-130. March. 1986.

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

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

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

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

15. R. Snodgrass and I. Ahn. A taxonomy of time in databases. In ACM SIGMOD Conference, May 1985.

16. S.B. N avathe and R. Ahmed. A temporal relational model and a query language. In Infomnation Sciences, North-Holland, Vol. 49, No. 1, 2, and 3, 1989.

17. S. Gadia and C. Yeung. A generalized model for a temporal relational database. In ACM SIGMOD Conference, June 1988.

18. J. Clifford and D. Warren. Formal semantics for time in databases. ACM TODS, 8(2), June 1983.

19. S. Gadia. A homogeneous relational model and query language for temporal databases. ACM TODS, 13(4), December 1988.

20. A. Guttman. R-trees: A dynamic index structure for spatial searching. In A CM SIGMOD Conference, May 1984.

21. K. Ooi, B. McDonell and R. Sacks-Davis. Spatial kd-tree: Indexing mechanism for spatial database. In IEEE COMPSAC 87, 1987.

22. D. Comer. The ubiquitous b-tree. ACM Computing Surveys, 11(12), June 1979.

23. R. Elmasri and S. Navathe. Fundamentals of Database Systems. Benjamin/Cummings, 1989.

24. R. Elmasri and G. Wuu. A temporal model and language for er databases. In IEEE Data Engineeting Conference, February 1990.