The Australian National University

TR-CS-95-02

Scheduling Issues in Partitioned Temporal Join

Jeffrey X. Yu and Kian-Lee Tan

May 1995

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

techreports@cs.anu.edu.au

A list of technical reports, including some abstracts and copies of some full reports may be found at:

file://dcssoft.anu.edu.au/pub/www/dcs/techreports/

Recent reports in this series:

TR-CS-95-01	Craig Eldershaw and Richard P. Brent. *Factorization of large integers on some vector and parallel computers.* January 1995.
TR-CS-94-10	John N. Zigman and Peiyi Tang. *Implementing global address space in distributed local memories.* October 1994.
TR-CS-94-09	Nianshu Gao and Peiyi Tang. *Vectorization using reversible data dependencies.* October 1994.
TR-CS-94-08	David Sitsky. *Implementation of MPI on the Fujitsu AP1000: Technical details release 1.1.* September 1994.
TR-CS-94-07	Markus Hegland. *Real and complex fast fourier transformations on the Fujitsu VPP500.* June 1994.
TR-CS-94-06	Richard P. Brent. *Uses of randomness in computation.* June 1994.

Scheduling Issues in Partitioned Temporal Join

Jeffrey X. Yu
Department of Computer Science
Australian National University
Canberra, ACT 0200, Australia
email: yu@cs.anu.edu.au

Kian-Lee Tan
Dept. of Information Systems & Computer Science
National University of Singapore
Kent Ridge, Singapore
email: tankl@iscs.nus.sg

Abstract

One of the major problems of temporal databases is to develop efficient algorithms for operations that involves the time attributes. An operation that has received much attention in recent years is the temporal join which matches records from two temporal relations whose time intervals overlap. Under a partition-based algorithm, temporal data are split into partitions. During the join process, a partition in one relation only needs to join with some, but not all, partitions of the other relation. In this paper, we address scheduling issues in such an algorithm. Depending on the orders in which partitions are read, the number of I/Os incurred varies. We propose a three-phase scheduling framework to minimize the number of I/Os incurred. From the framework, a large number of scheduling strategies can be derived. We also study several representative scheduling strategies and report our findings in this paper.

1. Introduction

Temporal join matches records from two temporal relations whose time intervals overlap. It is frequently used in applications that find events that happen at the same time. Some example queries include "Retrieve the employees who work on the same department or project in a company database"; "Retrieve the nodes being connected (disconnected) during certain period of time in a communication network" and "Retrieve the suspects that appeared at the same location at the same time in a criminal database system of the police department". To answer such queries requires computing the overlap among the time intervals of different records in the temporal relations. Unlike joins in traditional relational database systems where equi-joins are the commonest form, joins on time intervals are non-equijoins and are more expensive to process.

Among the three basic join algorithms – nested-loops, sort-merged and partition-based – the partition-based algorithms pose interesting challenge. Under a partition-based algorithm, temporal data are split into partitions. During the join process, a partition in one relation only needs to join with some, but not all, partitions of the other relation. However, as pointed out in [6], the performance bottleneck of partition-based temporal join method is that those partitions containing so-called "long-lived" records (records whose time intervals span over a long period) need to be compared with almost all the other partitions. As a result, the savings obtained from the reduction of comparisons among partitions may not be able to offset the overhead of partition-based methods – the cost of partitioning. To solve the problem, Lu, et. al. [6] avoided the partitioning phase by clustering records into partitions based on the time attributes. To facilitate direct access to the partitions, a "spatial" index is built on the set of partitions.

In this paper, we address a subproblem in designing partition-based temporal joins – the scheduling of partitions to be processed. Depending on the orders in which partitions are read, the number of I/Os incurred varies. We propose a three-phase scheduling framework to minimize the number of I/Os incurred. From the framework, a large number of scheduling strategies can be derived. We also study several representative scheduling strategies and report our findings in this paper. To study the scheduling strategies, we adopt the partition-based algorithm proposed in [6] as the join algorithm.

The rest of this paper is organized as follows. Section 2 provides the background information to our study. We also review existing work on partition-based temporal joins, and discuss scheduling issues. In Section 3, we present the scheduling strategies. Section 4 presents the results of a performance study on the scheduling algorithms. Finally, we summarize our conclusions in Section 5 with discussions on possible extensions of the current work.

2. Preliminaries

In this section, we present some terminologies on temporal databases. We will also review related work on partition-based temporal join. A more in-depth discussion on the spatially partitioned join algorithm [6] is presented. Finally, we discuss and introduce the scheduling issues.

2.1. Terminologies

Attributes of a temporal relation can be non-time varying attributes (such as employee id, name, sex), time-varying attributes (such as salary, qualifications) and time attributes that indicate the time interval that the given values of the time-varying attributes are valid. The *time interval*, denoted $[\mathrm{T_S}, \mathrm{T_E}]$, $T_E > T_S$, where $\mathrm{T_S}$ is the start time and $\mathrm{T_E}$ the end time, semantically represents the lifespan of the record in question. The *time dimension* is represented as a time

interval [0, T_{now}], where 0 represents the starting time of the application and T_{now} refers to the current time which is continuously increasing. Moreover, all relations are assumed to be in first temporal normal form [10]. As such, there are no two intersecting time intervals for a given surrogate instance. We say that two records, r and s, *intersects* if and only if their time intervals overlap, i.e. $r.T_S \leq s.T_E \wedge r.T_E \geq s.T_s$. We also say that an interval $[T_S,\ T_E]$ contains another interval $[t_s, t_e]$ if and only if $T_S \leq t_s \wedge t_e \leq T_E$.

A time join, denoted $\bowtie^T$ on two temporal relations R and S, consists of the concatenation of all records $r \in R$ and $s \in S$ such that the time attribute values in r and s intersect. The start and end times of a resulting record, say z, are given as follows:

$$z.T_S = \max(r.T_S, s.T_S) \quad \text{and} \quad z.T_E = \min(r.T_E, s.T_E)$$

For ease of reference in sequel, we use the term "join" to refer exclusively to the time join, and the term "relation" to mean temporal relation, unless otherwise stated.

2.2. Partitioned-based Temporal Joins

Initial work on temporal joins focused on refinements of the conventional nested-loops algorithm [2, 3, 9]. These algorithms exploit the sort order of the relations to avoid full scan of the inner and/or outer relations.

Partition-based algorithms for temporal join proposed in the literature can be classified into the following three types:

Static Partitioning [4]

In this method, R and S are range-partitioned into n non-overlapping intervals I_i, $1 \leq i \leq n$ that covers completely the time line. A record of R appear in the ith partition if its start timestamp is in the interval I_i. However, a record of S will appear in partition S_i if its temporal interval intersects with I_i. In other words, records of S may be replicated across several partitions. The advantage of replicating S is that each partition of R needs to be joined with the corresponding partition of S only. However, both storage and processing cost may be high.

Dynamic Partitioning [12]

In this case, relation R is also range-partitioned into n non-overlapping intervals I_i , $1 \leq i \leq n$ that covers completely the time line. To avoid replicating records of R and S into multiple partitions, Soo, *et. al.*, keeps each record in the last partition that the record overlaps [12]. The join computation is performed backward by processing partition n first, followed by partition $n - 1$, and so on. To compute the join results correctly, those records whose time intervals

intersect more than one partition range will be retained in memory to be combined with records in the next partition. For example, a record t_R whose time interval intersects partitions k and $k-1$, is stored in partition k. After partition k of R has been joined with partition k of S, records whose time intervals are contained in the interval of partition k are swapped out to prepare for the join of partitions R_{k-1} and S_{k-1}. However, t_R will be kept in memory so that it can be joined with records in S_{k-1}. In other words, the partitions are dynamically adjusted during the join computation. This method avoids replication of records at the expense of more sophisticated memory management during join computation.

Spatial Partitioning [6]

For the spatial partitioning technique, records in a temporal relation are mapped into discrete data points in a two-dimensional space using the function

$$f: I \rightarrow N \times N \qquad \text{where } f([a,b]) = (a, b-a)$$

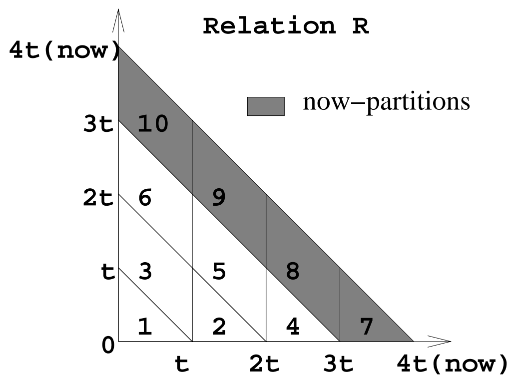
Applying the function f on records of a temporal relation results in a *spatial rendition.* Pictorially (see Figure 1(a)), the spatial rendition at T_{now} is a triangle formed by the lines $x=0$, $y=0$ and $x+y=T_{now}$. The temporal relation can then be partitioned as illustrated in Figure 1(a):

(1) The spatial rendition is split into n *diagonal strips.* The ith strip is bounded by the lines $x=0$, $y=0$, $x+y=T_{i-1}$, $x+y=T_i$, where $T_0=0$ and $T_n \geq T_{now}$.

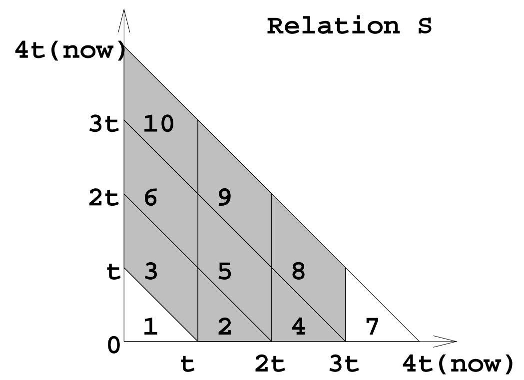
(2) The strips obtained are split into partitions by the lines $x=0$, $x=T_1, \ldots, x=T_n$. Thus, each partition is bounded by four lines: $x=T_i$, $x=T_{i+1}$, $x+y=T_j$, $x+y=T_{j+1}$, where $n \geq i \geq 0$ and $n \geq j \geq i \geq 0$. Given n strips, there will be a total of $\sum_{i=1}^{n} i = n \cdot (n+1)/2$ partitions. In the figure, we have $n=4$ and hence 10 partitions. For simplicity, we assume that the *partitioning interval* is the same for all partitions, that is $T_i - T_{i-1} = T_{i-1} - T_{i-2}$ for all i.

Unlike the other partition-based join where only those corresponding partition pairs R_i and S_i need to be compared to find matching records (with replication either statically or dynamically), partition R_i in a spatially partitioned join needs to join with more than one partition of S. Let Figure 1(a) and (b) represent the partitions of two relations R and S. Partition 5 in R, R_5, needs to be compared with all the partitions in the shaded region of Figure 1(b), i.e. partitions $S_2, S_3, S_4, S_5, S_6, S_8, S_9$ and S_{10}. More general, Figure 1(c) shows the set of partitions of S that must be compared with partitions of R, indicated by shadowed and black squares. For example, R_1 has to join with S_1, S_3, S_6 and S_{10}.

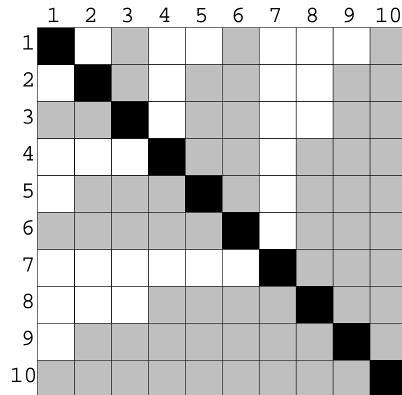
To cut down the cost of partitioning, records are clustered into buckets based on the mapping function, i.e. the records belonging to the same partition in the spatial rendition are stored



(a) Partitions of a spatial rendition.



(b) Region to be joined for partition 5.



(c) Join Processing of two partitioned renditions.

Figure 1: Partition-based Join Algorithm.

together as a bucket, and a bucket may have one or more pages. Furthermore, to facilitate direct access, a spatial index on the buckets is built.

The join can then be performed as follows: (1) traverse the index of relation R to read some of its partitions into memory, (2) traverse the index of S to read those partitions that should be joined with the partitions of R in memory, (3) perform the join, (4) repeat the process until all partitions of R are read.

2.3. Scheduling Issues

The performance of the spatially partitioned join method depends on two factors: (a) the index structure used, and (b) the ordering in which partitions are read. While the paper demonstrates that for a large class of indexes, the algorithm is effective, the latter issue has not been addressed.

The ordering in which partitions are read affects the performance of the algorithm. To illustrate, suppose all the buckets in Figure 1 are of the same size, and equal 1/4 that of the memory. Then the join can be processed in three iterations. If we read buckets 1-4 first, then 5-8, and finally 9-10, we require reading a total of 29 buckets of S. On the other hand, if we read buckets 1,3,6,10 first, then 2,5,9,4 and finally 7,8 we need to read only 26 buckets of S. Hence, depending on the order in which the partitions are read, the cost of the join varies.

A straightforward solution is to try all combinations to find the optimal solution. However, this can be avoided by either of the following ways:

(1) *Static Join Index.* In this method, we can build a join-index-like structure on the partitions. Unlike the traditional join-index [13] where each record in the index is a pair representing matching record pairs, the join index for partitions stores pairs of matching partitions. A similar structure has also been proposed in [5], and is shown to be effective in terms of join processing as well as maintenance cost.

(2) *Dynamic Join Index.* Instead of building a join index, one can be created dynamically when needed. This can be done by traversing the indexes and generates the matching partition pairs.

In this paper, we focus on the scheduling problem. For simplicity, we assume the following:

- The matching pairs are available as input to the scheduler, whose output is the sequence in which the partitions should be read. This is not unreasonable since we can apply either the dynamic or static join indexing techniques described above.
- Each partition fits in memory. This is also reasonable since a partition that is larger than the memory can be split into smaller partitions that fit in memory.

This work is similar to the problem of finding an optimal schedule for page fetches in

relational join operations [7, 8]. However, our work is different in several ways. First, the number of pages for the partitions of a temporal relation may be different. In [7, 8], all pages are of the same size.[1] Second, these work look at the case of a two-page memory constraint only. Relaxing the constraints to a fixed buffer size greater than two requires more novel algorithms. Finally, and more importantly, there is no predetermined relationship between records and pages in relational join. On the other hand, the way partitions are generated are predetermined. Exploiting such knowledge may lead to more effective heuristics.

To minimize the number of disk I/Os by ordering the buckets subjected to the constraints imposed by the memory available is an NP problem [7]. In the next section, we shall describe formally several heuristics to achieve this.

3. Scheduling Strategies

In this section, we present formally the scheduling strategies. We first describe our basic framework which essentially decomposes the problem into three subproblems. This has the advantage of pruning the search space. Subsequent subsections present strategies for the subproblems. From the strategies to the subproblems, we can derive a large set of scheduling algorithms. Finally, we describe how a schedule is evaluated under different scheduling strategies.

3.1. Basic Framework

Before we looked at the basic framework, let us look at some definitions and observations that we made.

Definition 1. Bipartite Graph: *Let N_r and N_s be sets of nodes. A labelled bipartite graph is defined as $B(N, E, L, f)$. It consists a set of nodes $N = N_r \cup N_s$ where $N_r \cap N_s = \phi$, a set of edges $E \subseteq N_r \times N_s$, and a function $f : N \rightarrow L$ where L is a set of labels.*

Intuitively, N_r and N_s represent a set of partitions for relations R and S, respectively. Each node in N needs multiple pages to store data in that partition. Therefore, in the labelled bipartite graph, if $f(n_i)$ is 5 and $n_i \in N_r$, then it implies five pages are used to keep data in the partition of n_i in relation R. Each edge $e_i = (n_j, n_k)$ represents the relationship between n_j and n_k that all the pages held by n_j must join all the pages held by n_k. Given a node n_i, let , (n_i) denote a set of nodes it has to join, i.e.

$$, (n_i) = \begin{cases} \{n_j \mid (n_i, n_j) \in E\}, & \text{if } n_i \in N_r \\ \{n_j \mid (n_j, n_i) \in E\}, & \text{if } n_i \in N_s \end{cases}$$

[1] Of course, we can map our problem into theirs by splitting a partition into multiple pages.

Example 1. *Let $B(N, E, L, f)$ be a bipartite graph. Here, $N = N_r \cup N_s$, $N_r = \{n_1, n_2\}$ and $N_s = \{n_3, n_4\}$. $E = \{(n_1, n_3), (n_1, n_4), (n_2, n_3)\}$. $L = \{1, 2, 3\}$. And $f = \{(n_1, 2), (n_2, 1), (n_3, 3), (n_4, 1)\}$. A bipartite graph is shown in Figure 2.*

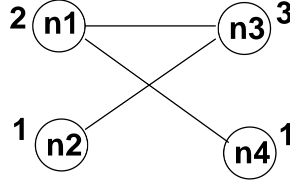


Figure 2: A simple labelled bipartite graph.

A labelled bipartite graph $B(N, E, L, f)$ is a simplified form of an unlabelled bipartite graph $B'(N', E')$ where N' is a set of pages rather than a set of partitions of pages. As a matter of fact, any labelled bipartite graph $B(N, E, L, f)$ can be converted into an unlabelled bipartite graph $B'(N', E')$. For example, the labelled bipartite graph in Example 1 can be converted into an unlabelled bipartite graph $B'(N', E')$. Here, $N' = N'_r \cup N'_s$, $N'_r = \{n_1, n_2, n_3\}$ and $N'_s = \{n_4, n_5, n_6, n_7\}$. $E' = \{(n_1, n_4), (n_1, n_5), (n_1, n_6), (n_2, n_4), (n_2, n_5), (n_2, n_6), (n_3, n_4), (n_3, n_5), (n_3, n_6), (n_3, n_7)\}$. In other words, an unlabelled bipartite graph is a special case of a labelled bipartite graph $B'' = (N'', E'', L, f)$. For example, $B'(N', E') = B''(N'', E'', F, f)$ if the following holds. $N' = N''$, $E' = E''$, $L = \{1\}$ and $f(n_i) = 1$ for all $n_i \in N''$. Obviously, a labelled bipartite graph has its advantages due to the fact that it needs smaller search space compared with that used in an unlabelled bipartite graph.

Definition 2. A rooted subgraph: *Let $B(N, E, L, f)$ be a bipartite graph. A rooted subgraph of B on node n_i, denoted by $B_i(N_i, E_i, L, f)$, is defined as follows. $N_i = , (n_i) \cup \{n_i\}$. $E_i = \{(n_i, n_j) \mid n_i \in (N_r \cap N_i) \ \wedge \ n_j \in (N_s \cap N_i)\}$. For simplicity, we also refer to $B_i(n_i)$ as the* rooted subgraph *on n_i.*

The cost of page accesses for a given rooted subgraph $B_i(n_i)$ is given by a function $f_p(n_i)$. If all $f(n_i)$ pages held by n_i can be kept in buffer, we have

$$f_p(n_i) = f(n_i) + \sum_{n_k \in \Gamma(n_i)} f(n_k)$$

In Example 1, the cost for evaluating the rooted graph n_1 is $2 + 4$ page accesses under the condition that buffer size is bigger than 4 pages. In this case, two pages are used to keep $f(n_1)$ pages held by node n_1. One page is used to read each of those pages that are held by both n_3 and n_4. Finally, one page is needed for writing the results of joins to secondary storage. For simplicity, in the following discussion, we ignore the page used for output when we refer to buffer.

Definition 3. A subgraph: *Let $B(N, E, L, f)$ be a bipartite graph. A subgraph of B is defined as $B^c = \bigcup_{n_i \in G} B_i(n_i) \subseteq B$ where $G = \{n_{g_1}, n_{g_2}, ..., n_{g_k}\} (\subset N)$. For any $n_{g_i} \in G$, $B_{g_i}(n_{g_i})$ must overlap with some $B_{g_j}(n_{g_j})$, for $j \neq i$.*

For simplicity, in the following discussion, we also use G, a set of nodes, to identify a subgraph. The motivation for introducing the concept of subgraph is described as follows. Given a subgraph $G = \{n_{g_1}, n_{g_2}, ...\}$. And let $\overline{G}$ indicates that $\sum_{n_{g_i} \in G} f(n_{g_i})$ can fit into buffer. Then, the cost of evaluating a subgraph $\overline{G}$ is as follows.

$$g_p(\overline{G}) = \sum_{n_{g_i} \in \overline{G}} f(n_{g_i}) + \sum_{n_l \in K} f(n_l)$$

where $K = \bigcup_{n_{g_i} \in \overline{G}}{}^{,} (n_{g_i})$. As can be seen below, $g_p(\overline{G})$ can be less than the cost of evaluating rooted graphs separately.

$$g_p(\overline{G}) \leq \sum_{n_{g_i} \in \overline{G}} f_p(n_{g_i})$$

For example, in Example 1, if $G = \{n_1, n_2\}$ and a four page buffer is available for keeping $f(n_1) + f(n_2)$ pages in buffer, then its cost of page accesses is $7 (= 3 + 4)$. It is less then the cost of $f_p(n_1) + f_p(n_2)$ which needs 10 page accesses. In the latter case, we need less pages to keep either $f(n_1)$ or $f(n_2)$ in buffer. Nevertheless, it still needs more than 7 page accesses. Suppose that we have a four page buffer for reading pages from secondary storage. Then, when evaluating the rooted graph n_1, we need two pages for holding all $f(n_1)$ pages. The cost of evaluating the rooted graph n_1 is 6. After evaluating the rooted graph n_1, two pages held by n_3 and/or n_4 can remain in buffer. Hence, the cost for evaluating the next rooted graph n_2 is 3 page accesses. In total, it costs 9 page accesses which is greater than 7 page accesses.

Observation 1. (Minimal Groups) *Let $\overline{G}$ be a set of nodes. Here, $\overline{G} = \overline{G}_1 \cup \overline{G}_2$ and $\overline{G}_1 \cap \overline{G}_2 = \phi$. Suppose that $(\bigcup_{n_i \in \overline{G}_1}{}^{,} (n_i)) \cap (\bigcup_{n_j \in \overline{G}_2}{}^{,} (n_j)) \neq \phi$. Then*

$$g_p(\overline{G}) \leq g_p(\overline{G}_1) + g_p(\overline{G}_2)$$

Using Example 1, let $\overline{G} = \overline{G}_1 \cup \overline{G}_2$, $\overline{G}_1 = \{n_1\}$ and $\overline{G}_2 = \{n_2\}$. As mentioned above, the cost for evaluating G is 7 page accesses, whereas the cost for evaluating G_1 and G_2 is at least 9 page accesses. As a consequence of Observation 1, if buffer size is big enough, the optimal evaluation of a bipartite graph can be defined in the following observation.

Definition 4. (Maximum and Minimum Conditions on G) *Let B and G be a bipartite graph and a subgraph, respectively.*

- **maximum condition:** $B \subset \bigcup_{n_i \in G} B_i(n_i)$.
- **minimum condition:** $B \not\subset \bigcup_{n_i \in G'} B_i(n_i)$ *if* $G' \subset G$.

Observation 2. (Optimal Group) *Let B and $\overline{G}$ be a bipartite graph and a subgraph, respectively. If $\overline{G}$ satisfy the maximum and the minimum conditions, then it results in an optimal scheduling under the two conditions.*

It is obvious because the formula $g_p(\overline{G})$ implies that all pages only need to be read from the disk once. Note that no other scheduling can reduce more page accesses than the optimal scheduling. As a matter of fact, for a bipartite graph B, given two $\overline{G}_1$ and $\overline{G}_2$ that satisfy the maximum and the minimum conditions given in Definition 4, the cost of evaluating $\overline{G}_1$ is the same as the cost of evaluating $\overline{G}_2$. Under the constraints imposed on buffer size, any set of subgraphs $W = \{\overline{G}_1, \overline{G}_2, ...\}$, where $|W| > 1$, cannot be an optimal scheduling. A *suboptimal* scheduling, W_{opt}, can only be defined in terms of all possible sets of subgraphs $W_1, W_2, ..., W_m$.

The strategy of evaluating a bipartite graph can start from dividing nodes into sets of subgraphs $W_k = \{\overline{G}_{k_1}, \overline{G}_{k_2}, ...\overline{G}_{k_m}\}$, for $k = 1, 2, ...$, under the condition that all pages held by nodes in any G_{k_i} can fit into buffer. In addition, W_k must satisfy the following conditions.

- $\bigcup_{\overline{G}_{k_i} \in W_k} \overline{G}_{k_i}$ must satisfy the maximum and the minimum conditions given in Definition 4.
- Let $\overline{G}_{k_l}$ be a set of nodes. $\overline{G}_{k_i} \cap \overline{G}_{k_j} = \phi$ for any i and $j (\neq i)$.

The cost of evaluating such a W_k is as follows.

$$h_p(W_k) = \sum_{\overline{G}_{k_i} \in W_k} g_p(\overline{G}_{k_i})$$

The following formula always holds.

$$h_p(W_k) \leq \sum_{n_i \in N'} f_p(n_i)$$

where $N' = \bigcup_{\overline{G}_{k_i} \in W_k} \overline{G}_{k_i}$. For example, we can divide all N_r, into a set of subgraphs $W_r = \{\overline{G}_{r_1}, \overline{G}_{r_2}, ..., \overline{G}_{r_m}\}$ where $N_r = \bigcup_{\overline{G}_{r_i} \in W_r} \overline{G}_{r_i}$ and $\overline{G}_{r_i} \cap \overline{G}_{r_j} = \phi$ for any i and $j (\neq i)$.

Observation 3. (Group Ordering) *Given a set of subgraphs $W = \{\overline{G}_1, \overline{G}_2, ..., \overline{G}_m\}$. The ordering of evaluating such a set of subgraphs must be specified also. Then a list of $W' = [\overline{G}_{g_1}, \overline{G}_{g_2}, ..., \overline{G}_{g_m}]$ can be generated based on W. Here, a 1:1 mapping exists between W and W'. Let A, B and C are sublists of W'. Then*

$$h_p([A, \overline{G}_{g_i}, B, \overline{G}_{g_j}, C]) \geq h_p([A, \overline{G}_{g_i}, \overline{G}_{g_j}, B, C])$$

on the condition that the overlap between $\overline{G}_{g_i}$ *and* $\overline{G}_{g_j}$ *is larger than that between* $\overline{G}_{g_i}$ *and any* $\overline{G}_{g_k} \in B$.

As can be seen, all the observations are based on an **assumption** that all subgraphs G_i in a W must be $\overline{G}_i$. In other words, the buffer must be larger than $f(n_{i_j})$ pages for any rooted graph $n_{i_j} (\in G_i)$. Under this assumption and the observations, it is possible to find an optimal scheduling by searching all combinations. However, as the problem is NP-complete, we adopt a three-phase optimization strategy. A suboptimal schedule can be obtained by the following three phases.

(1) **(Ordering Nodes)**
All necessary nodes are sorted somehow into a sequence. And the following two phases generate a suboptimal schedule based on the sequence of nodes.

(2) **(Grouping Nodes)**
Based on the sequence of nodes, find groups on the following two conditions. First, each group $G_k = \{n_{k_1}, n_{k_2}, ...\}$ should be able to keep all pages held by nodes $n_{k_i} (\in G_k)$ in buffer. Secondly, the overlap between pairs of $\{, (n_{k_1}),, (n_{k_2}), ...\}$ must be large. This phase results in a sequence of groups.

(3) **(Reordering Sequence)**
Based on the sequence of groups generated in the second phase, reorder the sequence of groups. The sequence $[..., \overline{G}_1, \overline{G}_2, \overline{G}_3, ...]$ can perform better than $[...,\overline{G}_1,\overline{G}_3,\overline{G}_2,$ $...]$, if $\overline{G}_1$ overlaps more nodes in $\overline{G}_2$ than those in $\overline{G}_3$.

3.2. Ordering Nodes

Among the three phases, ordering nodes is the most important phase because the other two phases depend on this order. The major advantage of using a predefined ordering is that it enables applications to provide its own strategy and therefore the applications can have more control over the scheduling. This phase comprises the following two steps:

- Given a bipartite graph $B(N, E, L, f)$, a subset of nodes $G = \{n_{g_1}, n_{g_2}, ..., n_{g_k}\} (\subseteq N)$ must be decided under the condition that the corresponding subgraph G satisfies the maximum and the minimum conditions. We call it **minimum set** condition from now on.
- Order nodes in G.

As shown in Figure 1, a partition may be assigned a number. A total ordering among all partitions may then be predefined. When two relations have to be joined, the system implicitly

uses this ordering to join one partition in one relation with those matching partitions in the other relation. In general, the first diagonal strip joins fewer partitions than those in the last diagonal strip. Example 2 shows a bipartite graph for a join between two relations. Each of the two relations has three diagonal strips and a total of six partitions.

Example 2. *Let $B(N, E, L, f)$ be a bipartite graph. Here, $N = N_r \cup N_s$, $N_r = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$ and $N_s = \{n_8, n_9, n_{10}, n_{11}, n_{12}\}$. $E = \{(n_1, n_7),(n_1, n_9),(n_1, n_{12}),(n_2, n_8),(n_2, n_9),(n_2, n_{11}), (n_2,n_{12}),(n_3, n_7),(n_3,n_8),(n_3, n_9), (n_3, n_{11}), (n_3, n_{12}),(n_4, n_{10}),(n_4, n_{11}), (n_4, n_{12}), (n_5, n_8), (n_5, n_9), (n_5, n_{10}), (n_5, n_{11}), (n_5, n_{12}), (n_6, n_7), (n_6, n_8), (n_6, n_9), (n_6, n_{10}),(n_6, n_{11}),(n_6, n_{12})\}$. $L = \{1, 3, 5, 10, 11, 12, 13, 30, 8\}$. $f = \{(n_1, 1),(n_2, 3),(n_3, 5), (n_4, 10),(n_5, 11),(n_6, 12),(n_7, 13),(n_8, 30), (n_9, 8),(n_{10}, 1),(n_{11}, 3),(n_{12}, 5)\}$.*

Let $B(N, E, L, f)$ be a bipartite graph. Two basic strategies are available.

- **Strategy 1:** First, find a subset of nodes, $G \subset N$, which satisfies the minimum set condition. Then, sort nodes in G by some means.
- **Strategy 2:** First, sort all nodes in N by some means, then find a subset G of N that satisfies the minimum set condition.

In the first strategy, we have to find a subset G which satisfies the minimum set condition. For example, in Example 2, we can see that N_r and N_s represent partitions belonging to relations R and S, respectively. And both N_r and N_r satisfy the minimum set condition. In order to evaluate the bipartite graph, we can simply evaluate either all the rooted graphs $n_i (\in N_r)$ or all the rooted graphs $n_j (\in N_s)$ instead. Suppose that we choose all nodes in N_r in order to evaluate the bipartite graph. Three possible orderings are available for the first strategy.

- **Forward Ordering**: Order nodes in N_r by sorting the number assigned to each partition in an ascending order. Referring to Figure 1, this means that partition 1 will be processed first, followed by partition 2 and so on.
- **Backward Ordering**: Similar with the forward ordering, order nodes in N_r by sorting the number assigned to each partition in a descending order. In this case, partition 10 will be processed first, followed by partition 9, and so on.
- **Sort Ordering**: The total number of pages a node $n_{r_i} \in N_r$ has to join is given by $\sum_{n_k \in \Gamma(n_{r_i})} f(n_k)$. Sort all nodes $n_{r_i} \in N_r$ in a descending order based on $\sum_{n_k \in \Gamma(n_{r_i})} f(n_k)$. The partition are then joined in this order.

The second strategy is specified by the following Probe Ordering.

- **Probe Ordering**: Given a bipartite graph $B(N, E, L, f)$. The total number of pages a node $n_i \in N$ has to join is given by $\sum_{n_k \in \Gamma(n_i)} f(n_k)$. Sort all nodes into a descending order based on $\sum_{n_k \in \Gamma(n_i)} f(n_k)$. Let P be such a sequence, $[n_{p_1}, n_{p_2}, ..., n_{p_m}]$. P does not satisfy the minimum set condition. Hence, we need to remove nodes from the tail of P until P satisfy the minimum set condition. The algorithm for removing nodes is given below.

Algorithm 1. *Let $i(P)$ be the i-th element in the sequence P. Function $rest(P, i)$ returns a sequence that removes the i-th element from P. Function $length(P)$ returns the length of P.*
Input: a sequence of nodes P.
Output: a sequence of nodes P' which satisfies the minimum condition.
Procedure:

(1) *Let $l = length(P)$, $n_l = l(P)$ and $P_{tmp} = rest(P, l)$.*

(2) *If $B_l(n_l) \subseteq \bigcup_{n_i \in P_{tmp}} B_i(n_i)$, then*
begin
$P = P_{tmp}$; $l = length(P)$; *goto step 4;*
end

(3) $l = l - 1$.

(4) *If $l > 0$ then begin $n_l = l(P)$; $P_{tmp} = rest(P, l)$; goto step 2; end*

(5) $P' = P$.

3.3. Grouping Nodes and Reordering Sequence

The previous phase generates a sequence of groups, $P = [n_{p_1}, n_{p_2}, ...]$. In this phase, we need to obtain a sequence of subgraphs $W = [\overline{G}_1, \overline{G}_2, ...,]$ based on the sequence P.

Let $i(P)$ be the i-th element in the sequence P. Let $length(P)$ be a function that returns the length of P, and $rest(P, i)$ be a function that returns a sequence by removing the i-th element from P. There are several heuristic strategies available. We illustrate two such strategies here. Suppose that buffer size is X.

Algorithm 2. First-Fit*:*
Input: A sequence $P = [n_{p_1}, n_{p_2}, ...]$.
Output: A sequence $W = [\overline{G}_1, \overline{G}_2, ...]$.
Procedure:

(1) *Let W be an empty sequence.*

(2) *Let* $l = 1$, $n_l = l(P)$ *and* G *be an empty set.*

(3) $G = G \cup \{n_l\}$; $P = rest(P, l)$.

(4) *For* $k = 1$ *until* $length(P)$ *do*
begin
$n_k = k(P)$.
If $\sum_{n_i \in G} f(n_i) + f(n_k) \leq X$ *and* $B_k(n_k) \cap \bigcup_{n_j \in G} B_j(n_j) \neq \phi$ *then*
Append n_k *at the end of* G;
end
$P = P - G$; *Append* G *at the end of* W.

(5) *If* $length(P) > 0$ *then goto step 2.*

Algorithm 3. Better-Fit/Best Fit*:*
Input: A sequence $P = [n_{p_1}, n_{p_2}, ...]$.
Output: A sequence $W = [\overline{G}_1, \overline{G}_2, ...]$.
Procedure:

(1) *Let* W *be an empty sequence.*

(2) *Let* $l = 1$, $n_l = l(P)$ *and* G *be an empty set.*

(3) $G = G \cup \{n_l\}$; $P = rest(P, l)$.

(4) *For* $k = 1$ *until* $length(P)$ *do*
begin
$n_t = k(P)$; $m_1 = \sum_{n_i \in G} f(n_i)$; $M_1 = \bigcup_{n_i \in G} B_i(n_i)$.
If $m_1 + f(n_k) \leq X$ *and* $B_t(n_t) \cap M_1 \neq \phi$ *then*
begin
For $j = k + 1$ *until* $length(P)$ *do*
If $m_1 + f(n_j) \leq X$ *and* $B_j(n_j) \cap M_1 \neq \phi$ *and*
$(f(n_j) > f(n_k)$ *logic-op* $|B_j(n_j) \cap M_1| > |B_t(n_t) \cap M_1|$ $)$ *then*
$n_t = n_j$.
Append n_t *at the end of* G;
end
end
$P = P - G$; *Append* G *at the end of* W.

(5) *If* $length(P) > 0$ *then goto step 2.*

Where the 'logic-op' can be either 'and' or 'or'. When the logic-op is 'and', we call it "best-fit" algorithm, whereas when the logic-op is 'or', we call it "better-fit" algorithm.

Reordering the sequence of subgraphs $W = [\overline{G}_1, \overline{G}_2, ...]$ can be done by using a similar algorithm like Algorithm 2. As such, we shall not discuss it here.

3.4. Evaluation of Schedule

The execution of a subgraph $\overline{G}_k = \{n_{k_1}, n_{k_2}, ...\}$ can be done by first fetching pages from , (n_j) which does not have any overlap with any node used in $\overline{G}_j$ where $j = k+1$. Following this, the remaining pages (that overlapped) are read. This allows us to keep these pages in memory (if memory is available) for reuse when processing n_{j+1}.

If the first strategy in Section 3.2 is adopted. the evaluation of a sequence of subgraphs $W = [\overline{G}_1, \overline{G}_2, ...]$ generated is simple. The evaluation strategy is given in the following Algorithm 4.

Algorithm 4. Standard Evaluation
Input: A sequence $W = [\overline{G}_1, \overline{G}_2, ...]$.
Output: A result relation for a temporal join.
Procedure:

(1) Let $i = 1$.

(2) $\overline{G}_{curr} = \overline{G}_i$; Fetch all $f(n_i)$ pages from all node $n_i \in \overline{G}_{curr}$ into buffer.

(3) Let $N_{tmp} = \bigcup_{n_j \in \overline{G}_i}$, (n_j) and $N_{next} = \bigcup_{n_k \in \overline{G}_{i+1}}(,\ (n_k) \cup \{n_k\})$.
Then, $N_{first} = N_{tmp} - N_{next}$ and $N_{second} = N_{tmp} - N_{first}$.

(4) Evaluate all rooted graphs $n_i \in N_{first}$.

(5) Evaluate all rooted graphs $n_i \in N_{second}$.

(6) Let $i = i + 1$.

(7) If $i \leq length(W)$, then goto step 2.

If the second strategy in Section 3.2 is adopted, the situation is slightly different. For example, if we use Probe-ordering against the bipartite graph given in Example 1, W can be $[\overline{G}_1, \overline{G}_2]$ where $\overline{G}_1 = \{n_2\}$ and $\overline{G}_2 = \{n_3\}$. Suppose that we evaluate the rooted graph n_2 first, then when we need to evaluate the rooted graph n_3, we don't need to evaluate the edge (n_2, n_3) again. We should evaluate only the edge (n_3, n_1) when evaluating the rooted graph n_3. One possible evaluation can be described in Algorithm 5.

Algorithm 5. Partial-Evaluation
Input: A sequence $W = [\overline{G}_1, \overline{G}_2, ...]$.

Output: A result relation for a temporal join.
Procedure:

(1) *Let* $i = 1$ *and* E *be an empty set.*

(2) $\overline{G}_{curr} = \overline{G}_i$; *Fetch all* $f(n_i)$ *pages from all node* $n_i \in \overline{G}_{curr}$ *into buffer.*

(3) *Let* $N_{tmp} = \bigcup_{n_j \in \overline{G}_i} , (n_j)$ *and* $N_{next} = \bigcup_{n_k \in \overline{G}_{i+1}} (, (n_k) \cup \{n_k\})$.
Then, $N_{first} = N_{tmp} - N_{next}$ *and* $N_{second} = N_{tmp} - N_{first}$.

(4) *For each rooted graph* $B_i(n_i) \in N_{first}$ *do*
begin
  evaluate the edges in $B_i(n_i) - E$ *only.*
  $E = E \cup B_i(n_i)$.
end

(5) *For each rooted graph* $B_i(n_i) \in N_{second}$ *do*
begin
  evaluate the edges in $B_i(n_i) - E$ *only.*
  $E = E \cup B_i(n_i)$.
end

(6) *Let* $i = i + 1$.

(7) *If* $i \leq length(W)$, *then goto step 2.*

Obviously, Algorithm 5 adopts the same evaluation strategy as Algorithm 4, except that Algorithm 5 needs to maintain a set of edges, E, and check an edge (n_i, n_j) whether it belongs to E or not each time.

4. A Performance Study

In this section, we describe the performance study conducted to evaluate some representative scheduling techniques. For each algorithm, we simulate its join execution on a SPARCstatic 2 to obtain the number of I/Os needed. Since the result size is the same for all algorithms, we ignore the I/Os for writing the result size to disks. All the algorithms are compared on the basis of the number of I/Os required for the join execution.

Table 1 shows the parameters used and their default settings. Most of the parameters are self-explanatory. Each relation begins at time 0 unit, and the current time is 10,000 time units. Each relation has 20,000 objects. The start time of each object follows a Poisson distribution. Each object has an average of 5 versions, the duration of each of which is determined by an exponential distribution, i.e. the lifespan of each record is exponentially distributed. Thus,

each relation has 100,000 records. The partitioning interval at which the relations are to be partitioned is fixed at 200 time units, and hence each relation has a maximum of 1275 buckets. Each page is assumed to contain at most 20 objects.

Parameter	Default
Lifespan of relation	[0, 10,000]
Number of objects per relation	10,000
Average number of versions per object	5
Number of records per relation	100,000
Page size	20 objects
Partitioning interval	200 time units
Number of buckets	$(50 \times 51)/2 = 1{,}275$

Table 1: Parameters used and their default settings.

We have conducted a large set of experiments, and can only present some of the more interesting and representative ones here. The algorithms we studied and the notations are:

- SO denotes the strategy that employs sort ordering;
- FO denotes the strategy that employs forward ordering;
- BO denotes the strategy that employs backward ordering;
- PO denotes the strategy that employs probe ordering.

Moreover, all the four strategies used the first-fit grouping technique.

4.1. Experiment 1: Effect of Memory Size

Partition-based algorithms are sensitive to memory size [1, 11]. In this section, we study how sensitive the various algorithms are to the amount of memory available. We vary the memory size from 10% to 100% of the size of R. As *lifespan* of a record will affect the performance of a join to a large extent, two tests were conducted,

(1) The lifespan of each record is small with the mean value set to 100 time units. In this case, most of the records overlap one other bucket.

(2) The lifespan of each record is large with the mean value set to 800 time units, which means that on average, each record will overlap 4 buckets.

Figure 3 shows the results when the mean lifespan is 100 time unit. Generally, all the algorithms perform in a similar manner for large memory. However, when the memory is small,

we observed several interesting points. First, we observe that strategy SO performs poorly compared to the other strategies. This is because though the buckets are sorted in descending order of the total size of the matching partitions of S, it has two "weaknesses" – (1) it does not mean that the partitions to be read for bucket R_i is a superset of bucket R_{i+1}, (2) it does not mean that the partitions to be read overlapped. On the other hand, strategies BO and FO almost always ensure that the set of partitions of S for the next partition of R to be read in memory overlapped those that correspond to the partitions already in memory. Second, we find that strategy PO is not effective. When the mean record lifespan is short, most of the buckets at the higher end of the spatial rendition is empty, while those at the lower half of the spatial rendition are occupied. In other words, there are fewer non-empty partitions with each partition containing larger number of records than average. Moreover, because of the same data distribution, whenever a partition of R is read, the corresponding partition of S will be the next to be read. With small memory size and large partition size, this frequent toggling leads to the poor performance.

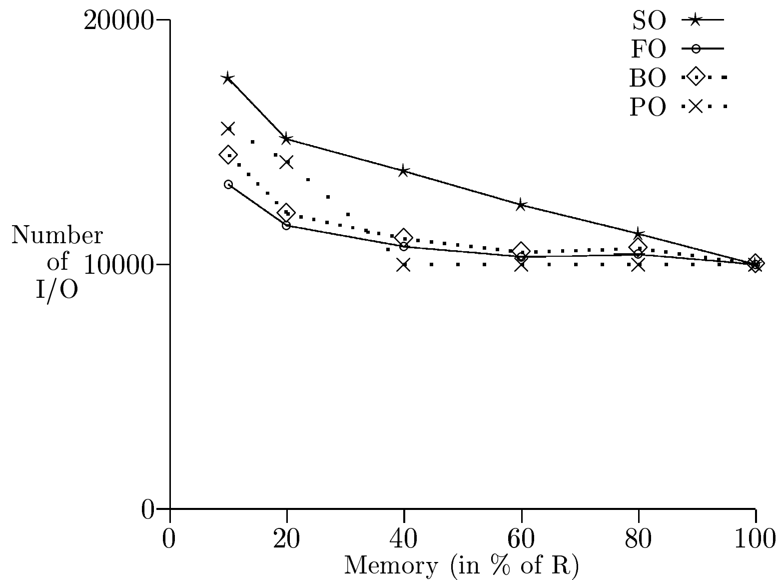


Figure 3: Effects of varying memory (mean lifespan = 100 units).

When the mean lifespan is 800 units, the relative performance of the algorithms is almost the same except for strategy PO. The result is shown in Figure 4. Again we see that SO performs poorly. Howsever, we see that strategy PO is superior to all strategies at small memory. This is because with a longer mean lifespan, more records stored at partitions that are higher up in the spatial rendition. Moreover, the number of partitions that contains data increases and hence the average number of records per bucket also decreases. Thus, the benefits of toggling between partitions of R and S increases.

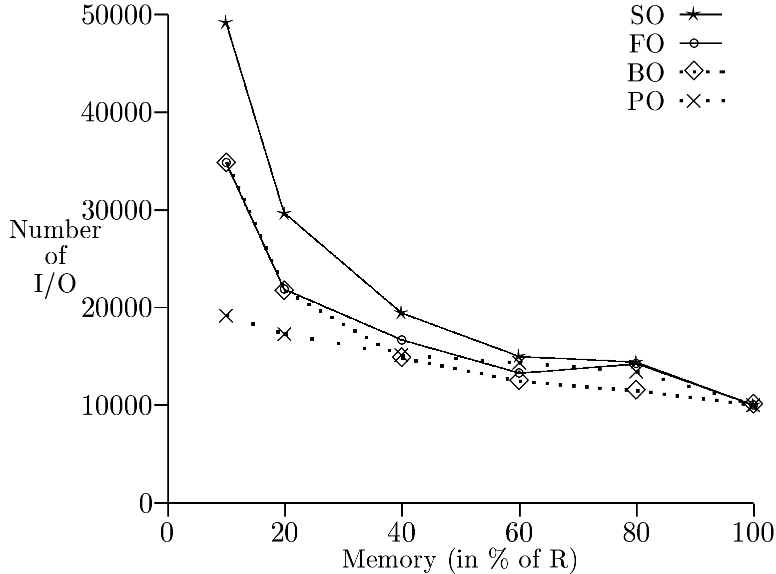


Figure 4: Effects of varying memory (mean lifespan = 800 units).

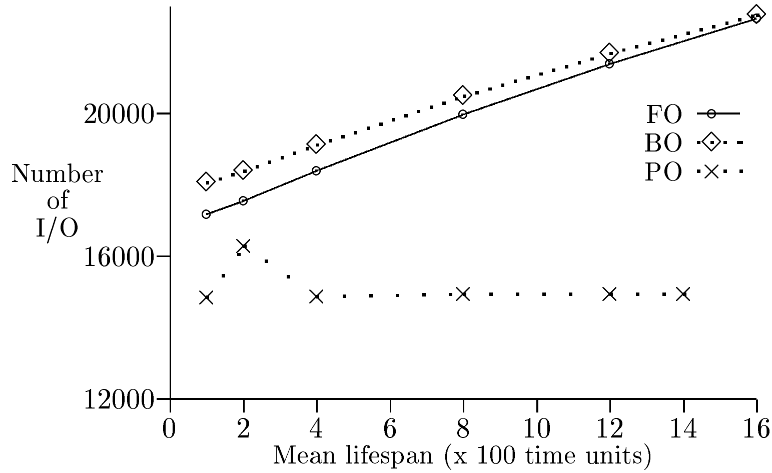
4.2. Experiment 2: Effect of Varying Lifespan

Since strategy SO performs poorly, we will focus our attention on the other three strategies for this and subsequent studies. In this experiment, we vary the mean lifespan of records in relation S from 100 time units to 1600 time units. The mean lifespan of records in relation R is fixed at 200 time units. Figure 5 shows the results under different memory availability.

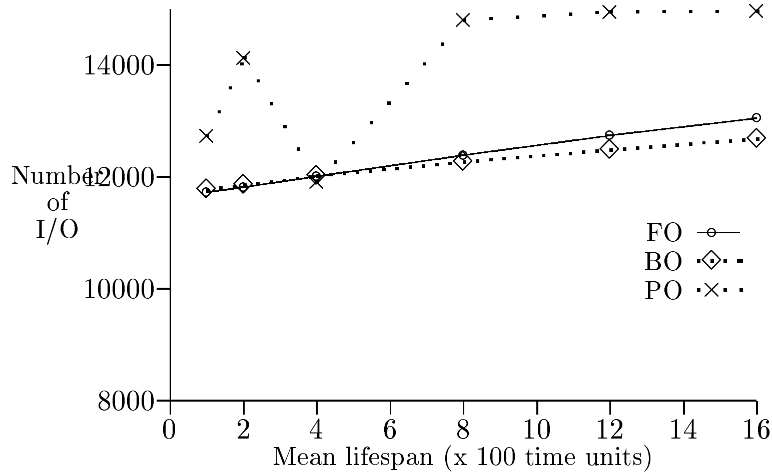
As expected, the number of disk I/O's required for the various scheduling strategies increases as the mean lifespan increases. This is because with increasing lifespan, each partition overlaps more partitions and hence the amount of work to be done for the join processing increases. As in the earlier experiments, PO is superior at small and large memory size but performed poorly at medium memory size. It is also interesting to note that when both relations have the same mean record lifespan, PO did not perform well. This is because of the effect of toggling as described earlier. With different mean record lifespan, the effect of toggling becomes beneficial.

4.3. Experiment 3: Vary number of buckets

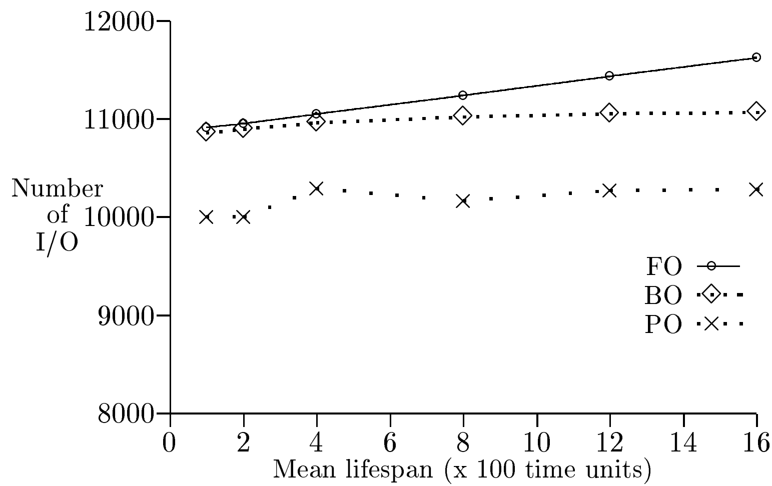
For partition-based algorithms, the granularity of the bucket may affect the performance of the algorithm. Recall that when the partitioning interval is t, we have $N = T_{now}/t$ diagonal strips, and there are a total of $N \cdot (N+1)/2$ buckets. With finer partitions, the number of S buckets to be read for each R bucket is controlled more closely. That is, fewer of the S buckets read are wasteful or unnecessary. In this experiment, we study this tradeoff by varying the number of buckets. Both the relations have the same number of buckets.



(a) Small Memory (10% R)

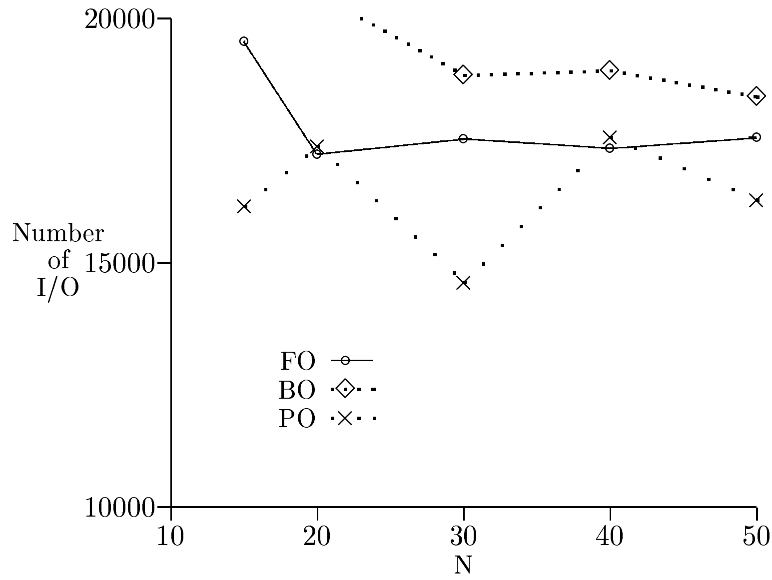


(b) Medium Memory (40% R)

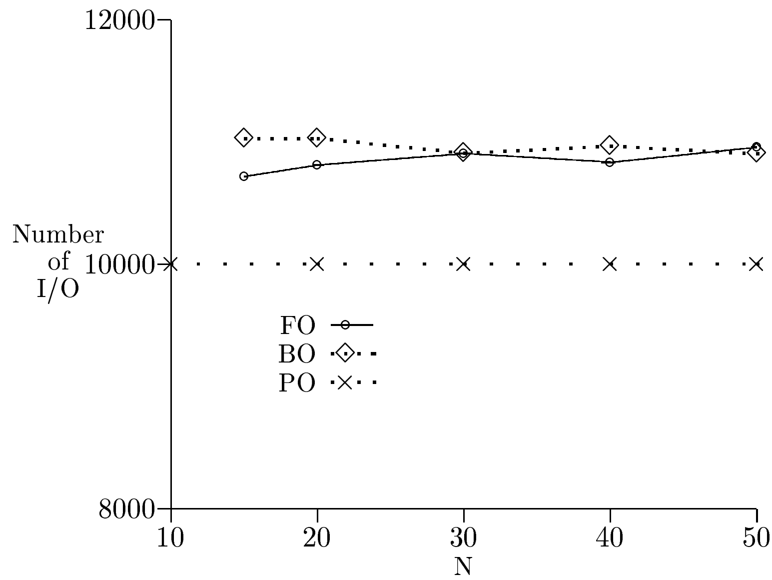


(c) Large Memory (80% R)

Figure 5: Vary the life span of records in one relation.



(a) Number of partitions = N(N+1)/2 (memory = 10% R)



(b) Number of partitions = N(N+1)/2 (memory = 80% R)

Figure 6: Effect of number of partitions.

Figure 6 shows the results of the experiments with mean lifespan 200 units for small and large memory. We observe that for strategies BO and FO, increasing the number of buckets generally reduces the I/O cost. For PO, there is a certain optimal number of partitions that will lead to superior performance (N = 30 in this experiment). Finally, as expected, for large memory size, the various strategies are not sensitive to the number of partitions.

5. Conclusions

In this paper, we have discussed scheduling issues in partition-based temporal join algorithms. Depending on the orders in which partitions are read, the number of I/Os incurred varies. We have studied the problem formally and proposed a three-phase scheduling framework to minimize the number of I/Os incurred. From the framework, a large number of scheduling strategies can be derived. An experimental study was also used to evaluate the performance of some representative strategies. While our results showed that no single strategy dominates performance, it also demonstrated that the choice of strategy is critical to performance (i.e. the difference in performance between strategies could be significant). We plan to extend this work in the following ways. First, we plan to automate the process of choosing the appropriate strategy at runtime. Second, we plan to study how the strategies proposed be used in other scheduling problems, for example scheduling of operations from multiple queries.

References

[1] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data*, Boston, NY, June 1984.

[2] H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 336–344, Kobe, Japan, April 1991.

[3] T. Leung and R. Muntz. Query processing for temporal databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 200–208, Los Angeles, CA, April 1990.

[4] T. Leung and R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 383–394, Vancouver, Canada, August 1992.

[5] H. Lu, R.-H. Luo, and B. Ooi. Spatial joins by precomputation of approximations. In *Proceedings of the 6th Australasian Database Conference*, pages 132–143, Glenelg, South Australia, January 1995.

[6] H. Lu, B. Ooi, and K. Tan. On spatially partitioned temporal joins. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 546–557, Santiago, Chile, August 1994.

[7] T. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page fetches in join operations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 488–497, Barcelona, Spain, August 1981.

[8] S. Pramanik and D. Ittner. Use of graph-theoretic models for optimal relational database accesses to perform join. *ACM Transactions on Database Systems*, 10(1):57–74, March 1985.

[9] S. Rana and F. Fotouhi. Efficient processing of time-joins in temporal data bases. In *Proceedings of the 3rd International Symposium on Database Systems for Advanced Applications*, pages 427–432, Taejon, Korea, April 1993.

[10] A. Segev and A. Shoshani. The representation of a temporal data model in the relational environment. *Lecture Notes in Computer Science*, 339:39–61, 1988.

[11] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[12] M. Soo, R. Snodgrass, and C. Jenson. Efficient evaluation of the valid-time natural join. In *Proceedings of the 10th International Conference on Data Engineering*, pages 282–292, February 1994.

[13] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.