©1994 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

Efficient Evaluation of the Valid-Time Natural Join

Michael D. Soo Richard T. Snodgrass

Department of Computer Science
University of Arizona
Tucson, AZ 85721

{soo,rts}@cs.arizona.edu

Christian S. Jensen

Department of Mathematics and Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg Ø, DENMARK
csj@iesd.auc.dk

Abstract

Joins are arguably the most important relational operators. Poor implementations are tantamount to computing the Cartesian product of the input relations. In a temporal database, the problem is more acute for two reasons. First, conventional techniques are designed for the optimization of joins with equality predicates, rather than the inequality predicates prevalent in valid-time queries. Second, the presence of temporally-varying data dramatically increases the size of the database. These factors require new techniques to efficiently evaluate valid-time joins.

We address this need for efficient join evaluation in databases supporting valid-time. A new temporal-join algorithm based on tuple partitioning is introduced. This algorithm avoids the quadratic cost of nested-loop evaluation methods; it also avoids sorting. Performance comparisons between the partition-based algorithm and other evaluation methods are provided. While we focus on the important valid-time natural join, the techniques presented are also applicable to other valid-time joins.

1 Introduction

Time is an attribute of all real-world phenomena. Consequently, efforts to incorporate the temporal domain into database management systems (DBMSs) have been on-going for more than a decade [Sno90, Soo91, Sno92]. The potential benefits of this research include enhanced data modeling capabilities, and more conveniently expressed and efficiently processed queries over time.

Whereas past work in temporal databases has concentrated on conceptual issues such as data modeling and query languages, recent attention has focused on implementation-related issues, most notably indexing and query processing strategies. We consider in this paper an important subproblem of temporal query processing, the evaluation of temporal join operations.

Joins are arguably the most important relational operators. They occur frequently due to database normalization and are potentially expensive to compute. Poor implementations are tantamount to computing the Cartesian product of the input relations. In a temporal database, the problem is more acute. Conventional techniques are aimed towards the optimization of joins with equality predicates, rather than the inequality predicates prevalent in temporal queries [LM90]. Secondly, the introduction of a time dimension may significantly increase the size of the database. These factors require new techniques to efficiently evaluate valid-time joins.

Valid-time databases support *valid-time*, the time when facts were true in the real-world [SA86, JCG+92]. In this paper, we consider strategies for evaluating the *valid-time natural join* [CC87, LM92], which matches tuples with identical attribute values during coincident time intervals. Other terms for the valid-time natural join include the *natural time-join* [CC87] and the *time-equijoin* (TE-join) [GS90]. Like its snapshot counterpart, the valid-time natural join supports the reconstruction of normalized data [JSS92a]. Efficient processing of this operation can greatly improve the performance of a database management system.

Join evaluation algorithms fall into three basic categories, nested-loop, sort-merge, or partition-based [ME92]. The majority of previous work in temporal join evaluation has concentrated on refinements of the nested-loop [SG89, GS90] and sort-merge algorithms [LM90]. Comparatively little attention has been paid to partition-based evaluation of temporal joins, the notable exception being Leung and Muntz who considered such algorithms in a multiprocessor setting [LM92b].

In this paper, we present a partition-based evaluation algorithm for valid-time joins that clusters tuples with similar validity intervals. If the number of disk pages occupied by the input relations is n then, in terms of I/O costs, our algorithm allows an $O(n)$ evaluation cost in many situations, thereby improving on the $O(n^2)$ cost of nested-loop evaluation and the $O(n \cdot log(n))$ cost of sort-merge evaluation. In addition, our approach does not require sort orderings or auxiliary access paths, each with additional update costs, and adapts easily to an incremental evaluation framework [SSJ93].

The paper is organized as follows. Section 2 formally defines the valid-time natural join in a common representational data model that is well-suited for query evaluation. A new, partition-based algorithm for computing the valid-time natural join is presented in Section 3. Performance comparisons between the partition-based algorithm and previous valid-time join evaluation algorithms are made in Section 4. Conclusions and future work are detailed in Section 5.

2 Valid-Time Natural Join

In this section, we define the valid-time natural join using the tuple relational calculus. The definition we provide is for a 1NF tuple-timestamped data model.

1063-6382/94 $3.00 © 1994 IEEE

An equivalent definition for a conceptual-level data model [JSS93, JSS93a] is given elsewhere [SSJ93].

In the data model, tuples are stamped with intervals denoting their time of validity. We assume that the time-line is partitioned into minimal-duration intervals, termed *chronons* [DS93]. Timestamps are therefore single intervals denoted by inclusive starting and ending chronons.

Let R and S be valid-time relation schemas

$$\begin{aligned} R &= (A_1, \ldots, A_n, B_1, \ldots, B_k, \mathrm{V}_s, \mathrm{V}_e) \\ S &= (A_1, \ldots, A_n, C_1, \ldots, C_m, \mathrm{V}_s, \mathrm{V}_e) \end{aligned}$$

where the A_i, $1 \le i \le n$, are the explicit join attributes, the B_i, $1 \le i \le k$, and C_i, $1 \le i \le m$, are additional, non-joining attributes, and V_s and V_e are the valid-time start and end attributes. We use V as a shorthand for the interval $[\mathrm{V}_s, \mathrm{V}_e]$. Also, we define r and s to be instances of R and S, respectively.

In the valid-time natural join, two tuples x and y join if they satisfy the snapshot equi-join condition (i.e., they match on the explicit join attributes), and if they have overlapping valid-time intervals. The attribute values of the resulting tuple z are as in the snapshot natural join, with the addition that the valid-time interval is the maximal overlap of the valid-time intervals of x and y. We formalize this with the following definitions.

DEFINITION: The function *overlap*(U, V) returns the maximal interval contained in both of the intervals U and V. We provide a procedural definition of *overlap*. The auxiliary functions *min* and *max* return the smallest chronon and largest chronons, respectively, in their argument sets.

```
overlap(U,V):
    common ← ∅;
    for each chronon t from U_s to U_e
        if V_s ≤ t ≤ V_e
            common ← common ∪ {t};
    if common = ∅
        result ←⊥;
    else
        result ← [min(common), max(common)];
    return result;
```
□

DEFINITION: The valid-time natural join of r and s, $r \bowtie^{\mathrm{V}} s$, is defined as follows.

$$\begin{aligned} r \bowtie^{\mathrm{V}} s = \{z^{(n+m+k+2)} \mid\ & \exists x \in r\ \exists y \in s \\ & (x[A] = y[A] \wedge z[A] = x[A] \wedge \\ & z[B] = x[B] \wedge z[C] = y[C] \wedge \\ & z[\mathrm{V}] = overlap(x[\mathrm{V}], y[\mathrm{V}]) \wedge z[\mathrm{V}] \ne \perp)\} \end{aligned}$$ □

3 Valid-Time Partition Join

In this section, we show how partitioning can be used to evaluate the valid-time natural join. We begin by describing, in Section 3.1, the general characteristics of partition joins. In Sections 3.2 to 3.4, we show how valid-time can be supported in a partition-based framework.

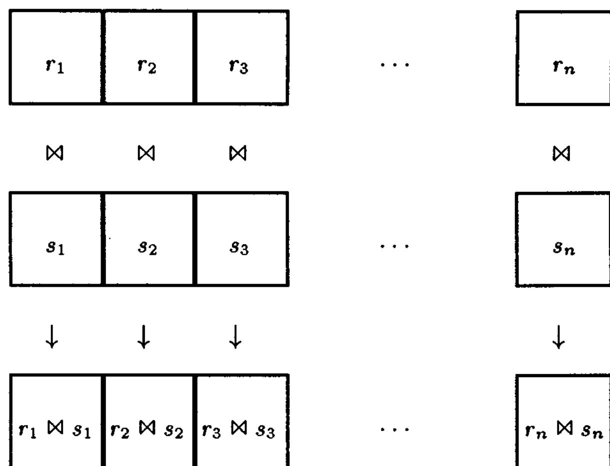

Figure 1: Partition Join of $r \bowtie s$

3.1 Overview of Partition Joins

Partition joins cluster tuples with similar join attribute values, thereby reducing the amount of unnecessary comparison needed to find matching tuples [ME92]. Both input relations are partitioned so that tuples in a particular partition of one relation can only match with tuples in a corresponding partition of the other relation. A primary goal is to perform the partitioning so that joins between corresponding partitions can be efficiently evaluated.

Partition-join evaluation consists of three phases. First, the attribute values delimiting partition boundaries are determined. The partition boundaries are chosen to minimize the evaluation cost—disk I/O is usually the dominant factor. Second, these attribute values are used to physically partition the input relations. In the ideal case, this involves linearly scanning both input relations and placing tuples into the appropriate partition. Lastly, the joins of corresponding partitions of the input relations are computed. In the ideal case, the partitions are small enough to fit in the available main memory and can be accessed with a single random disk seek followed by relatively inexpensive sequential reads. Ignoring the cost of operations performed in main memory, any simple evaluation algorithm, such as nested-loops or sort-merge, can be used to join the partitions once in memory. If the partitioning satisfies the given buffer constraints, the join can be computed with a linear I/O cost, thereby avoiding the quadratic complexity of the brute force implementation.

Figure 1 shows how partitioning is used to compute $r \bowtie s$ for two snapshot relations r and s. Relations r and s are initially scanned and tuples are placed into partitions r_i and s_i, $1 \le i \le n$, depending on their joining attribute values. The partitioning guarantees that, for any tuple $x \in r_i$, x can only join with tuples in s_i. The result, $r \bowtie s$, is produced by unioning the joins $r_i \bowtie s_i$.

Suppose that *buffSize* pages of buffer space are available in main memory. If a partition r_i occupies *buffSize*−2 pages or less then it is possible to compute

$r_i \bowtie s_i$ by reading r_i into main memory and joining it with each page of s_i one at a time. (The remaining page of main memory is used to hold result tuples.) Therefore, a single linear scan of r and s suffices to compute $r \bowtie s$. Also, the partitioning provides a natural clustering mechanism on tuples with similar attribute values. If partitions are stored on consecutive disk pages then, after an initial disk seek to the first page of a partition, its remaining pages are read sequentially. Last, it is easy to see how the algorithm can be adapted to an incremental mode of operation. For example, suppose that $r \bowtie s$ is materialized as a view, and an update happens to r in partition r_i. As tuples in r_i can only join with tuples in s_i, the consistency of the view is insured by recomputing only $r_i \bowtie s_i$.

3.2 Supporting Valid-Time

We now present a partition-join algorithm to compute the valid-time natural join $r \bowtie^{\mathrm{V}} s$ of two valid time relations r and s in the tuple-timestamped representation of Section 2.

Our approach is to partition the input relations using a tuple's interval of validity. For the corresponding partitions r_i and s_i, the partitioning guarantees that for each $x \in r_i$, x can only join with tuples in s_i, and, similarly, $y \in s_i$ can join only with tuples in r_i.

Tuple timestamping with intervals adds an interesting complication to the partitioning problem. Since tuples can conceivably overlap multiple partitions, these tuples, termed *long-lived tuples*, must be present in each partition they overlap when the join of that partition is being computed. That is, the tuple must be present in main memory when the join of an overlapping partition is being computed. Notice that this problem does not occur in the partition join of snapshot relations since, in general, the joining attributes are not range values such as intervals.

A straightforward solution to this problem simply replicates the tuple across all overlapping partitions [LM92b]. However, replication requires additional secondary storage space and complicates update operations.

We propose a different solution that guarantees the presence of the tuple in each overlapping partition when the join of that partition is computed, while avoiding replication of the tuple in secondary storage. Simply, we choose a single overlapping partition to contain the tuple on disk and dynamically migrate the tuple to the remaining partitions as the join is being evaluated.

The evaluation algorithm is shown in Figure 2. As with partition-join algorithms for conventional databases, three steps are performed. First, the attribute values that determine partition boundaries are determined. This is performed by procedure *determinePartIntervals.* Next the relations are partitioned by procedure *doPartitioning*, and lastly, the partitioned relations are joined by procedure *joinPartitions.*

We assume that Grace partitioning [KTMo83, ME92] is used in procedure *doPartitioning.* We reserve a single buffer page to hold a page of the input relation, and divide the remaining buffer space evenly among the partitions. Each tuple in r and s is examined and placed in a page belonging to the appropriate partition; when the pages for a given partition become filled they are flushed to disk. We assume that the number of partitions is small, and therefore, that sufficient main memory is available to perform the partitioning. This assumption held true for all experiments we performed. As partitioning is straightforward, we concentrate on the remaining algorithms. The following section describes how two partitioned relations are joined in procedure *joinPartitions.* For the time being, we assume that r and s are divided into n equal sized partitions and postpone until Section 3.4 the details of procedure *determinePartIntervals.*

```
partitionJoin(r, s):
    partIntervals ←
            determinePartIntervals(buffSize, |r|, |s|);
    r ← doPartitioning(r, partIntervals);
    s ← doPartitioning(s, partIntervals);
    return joinPartitions(r, s, partIntervals);
```

Figure 2: Evaluation of $r \bowtie^{\mathrm{V}} s$

3.3 Joining Partitions

Let P be a partitioning of valid time, i.e., P is a set of n non-overlapping intervals p_i, $1 \leq i \leq n$, that completely covers the valid-time line. Associated with each p_i is a partition r_i of r. We assume, for the purposes of this section, that each r_i has approximately the same number of tuples.

We assume that a tuple x is in the partition r_i if and only if $overlap(x[\mathrm{V}], p_i) \neq \perp$, and similarly for $y \in s_i$. Tuples are physically stored in the last partition they overlap, that is, a tuple x is physically stored in partition r_i if $overlap(x[\mathrm{V}], p_i) \neq \perp$ and $\neg\exists j$ such that $j > i$ and $overlap(x[\mathrm{V}], p_j) \neq \perp$.[1] The computation proceeds from $r_n \bowtie^{\mathrm{V}} s_n$ to $r_1 \bowtie^{\mathrm{V}} s_1$. For a given r_i, all tuples $x \in r_i$ that overlap p_{i-1} are retained and added to r_{i-1} prior to computing $r_{i-1} \bowtie^{\mathrm{V}} s_{i-1}$, and similarly for s_{i-1}. As tuples are initially placed in their last overlapping partition, this algorithm ensures that tuples are present in each partition they overlap, and does so without introducing unnecessary redundancy in secondary storage. Notice also that if a given tuple x overlaps partitions $p_j, \ldots, p_{i-1}, p_i$ then x must be present in $r_j, \ldots, r_{i-1}, r_i$ when their corresponding join is computed; therefore, no unnecessary comparisons are performed.

The buffer allocation strategy used in this algorithm is shown in Figure 3. Space is allocated to hold an entire partition r_i of the outer relation r, a page of the corresponding partition s_i of the inner relation, a page, the *tuple cache*, to hold the long-lived tuples of s, and a page to hold the result tuples. For a detailed description of the movement of tuples between the buffers, see Appendix A.1.

[1] An equivalent strategy is to place tuples in their first partition and propagate long-lived tuples towards the last partition during evaluation. We chose the given strategy with consideration for incremental adaptations described elsewhere [SSJ93].

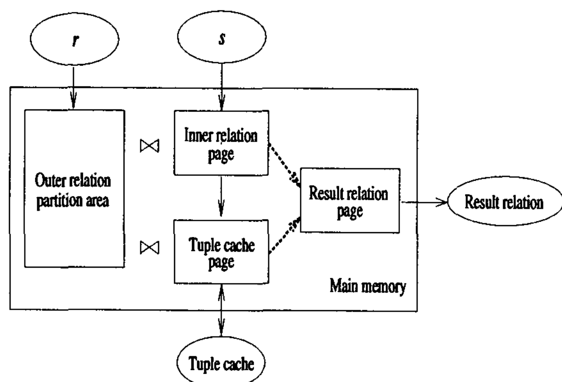

Figure 3: Buffer Allocation for $r \bowtie^{\mathrm{v}} s$ Evaluation

3.4 Partitioning Strategies

In the previous section, we described how the join of two previously partitioned relations was computed, assuming that each partition of the outer relation r, contained approximately equal numbers of tuples. We show in this section how to determine a partitioning of the input relations that satisfies this property with relatively small I/O cost. Our method is inspired by the partition-size estimation technique originally developed for the evaluation of *band-joins* [DNS91].

In Figure 3, a single buffer page is allocated to each of the inner relation buffer, tuple cache, and result relation, and *buffSize* pages are allocated to hold a partition of the outer relation. Our goal is to ensure that each r_i fits in the available *buffSize* pages with high probability, while minimizing the I/O cost of ensuring this important property.

The task at hand is to construct a set of *partitioning intervals* that covers the valid-time line. Tuples belong to a partition if they overlap, in valid time, the corresponding partitioning interval. Note that the length of a partitioning interval p_i determines the cardinality of the resulting partition r_i.

A simple strategy to construct the p_i is to sort r on V_e or V_s, and then choose the partitioning chronons in a subsequent linear scan. While this yields an optimal solution, it is prohibitively expensive.

A better solution is to choose partitioning intervals that with high probability are close to those that would have been chosen with the exact method. To do this, we randomly sample tuples from r, and, based on this sample set, choose a set of partitioning chronons, from which the partitioning intervals are constructed. As our partitioning is only approximate, some portion of the *buffSize* pages must be reserved to accommodate errors in the chronon choices that would likely result in overflow of the outer relation partition area. We note that should such errors occur, that is, a partition is created that is bigger than *buffSize* pages, the correctness of the join algorithm is not affected—only performance will suffer due to buffer thrashing.

Samples drawn from the outer relation are used to determine the intervals used to partition both the outer and inner relations. In addition, this same sample set is used to estimate the caching costs associated with long-lived tuples in the inner relation. We make the implicit assumption that the distribution, over valid time, of tuples in the outer and inner relations is similar, thereby allowing us to use a single sample set for both purposes. We provide justification for this assumption later.

The cost of evaluating $r \bowtie^{\mathrm{v}} s$ has the following three components (c.f., Figure 2).

- C_{sample}—the cost of sampling r,
- $C_{partition}$—the cost of creating the partitions r_i and s_i, $1 \leq i \leq n$, and
- C_{join}—the cost of joining the partitions r_i and s_i, $1 \leq i \leq n$.

The total I/O cost C_{total} is the sum of these,

$$C_{total} = C_{sample} + C_{partition} + C_{join}.$$

Our goal, then, is to choose a set of partitioning intervals so that the estimated evaluation cost, C_{total}, is minimized. Since the cost of Grace partitioning is not affected by the chosen partition size (it is dependent only on the amount of available main memory), we need only consider the sum $C_{sample} + C_{join}$. In the following, we show how to compute the set of partitioning intervals that minimizes $C_{sample} + C_{join}$.

Let *partSize* $\leq$ *buffSize* be the estimated size of an outer relation partition. We want to find a *partSize* that minimizes $C_{sample} + C_{join}$. Let *errorSize* = *buffSize* − *partSize* be the amount of buffer space available to handle overflow if a partition exceeds the estimated size. If *partSize* is large then *errorSize* is small. The effect of a small *errorSize* is to increase C_{sample} since, in order to prevent overflowing the smaller error space, higher accuracy is needed when choosing the partitioning intervals. However, a large *partSize* decreases C_{join} since tuples are less likely to overlap many partitions when the partitioning intervals are large, resulting in a decrease in tuple-cache paging. Alternatively, consider the effects of a small *partSize*, and, hence, large *errorSize*. Since more overflow space is available, fewer samples need to be drawn, and C_{sample} decreases. However, the smaller *partSize* increases C_{join} since tuples are more likely to overlap multiple partitions, if the partitioning intervals are small.

In summary, a cost tradeoff occurs between the amount of sampling performed on the outer relation, a component of C_{sample}, and the amount of paging performed on the tuple cache, a component of C_{join}. The optimal solution minimizes the sum $C_{sample} + C_{join}$.

Figure 4 plots sampling and tuple-cache paging costs for increasing partition sizes. As seen from the figure, as the expected partition size *partSize* increases, sampling costs (C_{sample}) increase monotonically and tuple-cache paging costs (and hence C_{join}) decrease monotonically. In order to minimize the evaluation cost, the sum of the sampling cost and the tuple-cache paging cost (shown as a dotted line in the figure) must be minimized.

This minimal sum is determined by computing, given the buffer constraint *buffSize*, $C_{sample} + C_{join}$ for each possible *partSize*. If few long-lived tuples are

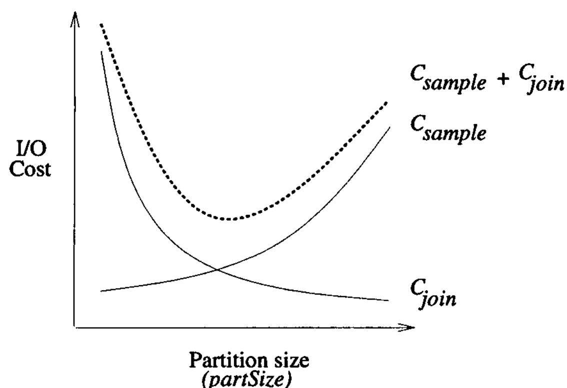

Figure 4: I/O Cost for Partition Size

present in r then the tuple-cache paging cost will decrease very quickly, and the minimal cost will be obtained at a larger partition size. Conversely, if many long-lived tuples are present in r then the tuple-cache paging cost will decrease slowly, and the minimal cost will be obtained at smaller partition sizes.

The number of samples to draw is determined using the Kolmogorov test statistic [Con71, DNS91]. The Kolmogorov test is a non-parametric test which makes no assumptions about the underlying distribution of tuples. With 99% certainty, the percentile of each chosen partitioning chronon will differ from an exactly chosen partitioning chronon by at most $1.63/\sqrt{m}$, where m is the number of samples drawn from r [Con71]. Since $1.63/\sqrt{m}$ represents a percentage difference from an exact partitioning, we know that $(1.63 \times |r|)/\sqrt{m}$ is the number of necessary error pages should a partition overflow the allotted *partSize* pages. Hence, we must have $(1.63 \times |r|)/\sqrt{m} \leq errorSize$ which implies that $m \geq ((1.63 \times |r|)/errorSize)^2$ samples must be drawn.[2]

The algorithm *determinePartIntervals*, shown in Appendix A.2, determines, for a given *buffSize*, the *partSize* that minimizes $C_{sample} + C_{join}$. The corresponding set of partitioning intervals is returned as its result. It uses an additional procedure *chooseIntervals*, shown in Appendix A.3 that chooses partitioning intervals from a set of partitioning chronons.

From the sample set, and its derived partitioning, the tuple cache paging costs are estimated. For a given partition, the estimated size of its tuple cache is the number of sampled tuples that overlap its partitioning interval scaled by the percentage of the relation sampled. This simple strategy suffices since accurately estimating the amount of tuple cache paging is not as critical as estimating the size of the outer relation partition. Partitions are large; therefore, rereading of partitions will incur a large expense. However, for any given partition, the size of its tuple cache is small, being bounded by the partition size; for many applications the tuple cache will contain a relatively small percentage of the partition. The expense of applying a sophisticated technique such as the Kolmogorov test, or directly sampling the inner relation, is not justified.

The algorithm *estimateCacheSizes*, shown in Appendix A.4, performs the tuple cache size estimation described here.

[2] It is interesting to note that the number of samples required is independent of $|r|$. Since *errorSize* is some number of pages we can express *errorSize* as a percentage of $|r|$, *errorSize*= $|r|/l$, where l is some integer. Substituting this expression for *errorSize* into the formula for m yields an expression independent of $|r|$.

4 Performance

In this section, we describe performance experiments involving the partition-join algorithm. We first describe, in Section 4.1, previous work in valid-time join evaluation and the general setting for the experiments. Sections 4.2 to 4.4 describe in detail the experiments we performed. Conclusions are offered in Section 4.5.

4.1 General Considerations

A wide variety of valid-time joins have been defined, including the *time-join*, *event-join*, *TE-outerjoin* [SG89], *contain-join*, *contain-semijoin*, *intersect-join*, *overlap-join* [LM92a], and *contain-semijoin* [LM92]. Refinements to the nested-loops algorithm were proposed by Gunadhi and Segev to evaluate several temporal join variants [SG89, GS91]. This work assumed that temporal data was "append-only," i.e., tuples are inserted in timestamp order into a relation, and once inserted into a relation are never deleted. With the append-only assumption, a new access path, the *append-only tree*, was developed that provides a temporal index on the relation. Simple extensions to sort-merge were also considered where, again, tuples were assumed to be inserted into a relation in timestamp order [SG89, GS91]. Leung and Muntz extended this work to accommodate additional temporal join predicates, mainly those defined by Allen [All83], and to incorporate various ascending and descending sort orders on either valid-start or valid-end time [LM90].

Simply stated, our work differs from most previous work in that we adapt the third and remaining join evaluation strategy, partitioning, to the evaluation of valid-time joins. Our approach does not require sort orderings or auxiliary access paths, each with additional update costs, and it adapts easily to an incremental evaluation framework. Partition-based evaluation of valid-time joins was investigated by Leung and Muntz in the context of parallel join evaluation, but their strategy required the replication of tuples across processors. We avoided replication for two reasons: to save on secondary storage costs and to easily adapt the algorithm to an incremental framework.

In order to evaluate the relative performance of our algorithm, we implemented main-memory simulations of partition join and sort-merge join, and calculated analytical results for nested-loops join. To obtain a fair comparison, we made the weakest assumptions possible about the input relations. That is, we do not assume any sort ordering of input tuples, nor the presence of additional data structures or access paths, where the incremental cost of maintaining a sort order or an access path is hidden from the query evaluation. However, the sort-merge algorithm was optimized to make

best use of the available main memory size, and similar remarks apply to the analytical results generated for nested-loops. We measured cost as the number of I/O operations performed by an algorithm, distinguishing between the higher cost of random access and the lower cost of sequential access. The parameters used in all of the experiments are shown in Figure 5.

Parameter	Value
Page size	4K bytes
Tuple size	128 bytes
Tuples per relation	262,144 tuples
Size of inner relation $\|r\|$	8192 pages (32 Mb)
Size of outer relation $\|s\|$	8192 pages (32 Mb)
Relation lifespan	1 million chronons

Figure 5: Global Parameter Values

We have attempted to choose realistic values for the example databases. If ten tuples are present for each object in the database, that is, ten pieces of information are recorded for each real-world entity, then the database contains approximately 26,000 objects. For most of the experiments, we are concerned more with ratios of certain parameters as opposed to their absolute values, and so choosing realistic values is less critical.

4.2 Sensitivity to Main Memory Buffer Size

In Section 3.4, we argued that the performance of the partition-join algorithm was dependent on the ratio of main memory buffer size to database size. That is, we expected that with larger memory sizes, the performance of the partition-join algorithm would improve. We designed an experiment to empirically investigate this tradeoff, and to simultaneously compare the partition-join algorithm with sort-merge join, at varying main memory allocations.

The tuples in the database were randomly distributed over the lifespan of the relation. In order to evaluate only the effect of memory size on the join evaluation, we eliminated the possibility of long-lived tuples by having each tuple's valid-time interval be exactly one chronon long. Long-lived tuples cause paging of the tuple cache in the partition-join algorithm and "backing-up" during the matching phase of the sort-merge algorithm. In addition, we were interested in the relative cost of random access versus sequential access since this varies among different hardware devices.

The allotted main memory was varied from 1 megabyte to 32 megabytes, and three trials were run for each of the join algorithms, where the random to sequential access cost was varied as 2:1, 5:1, and 10:1. The results of the experiments are shown in Figure 6. Note that the x-axis in the figure is log-scaled. Each curve represents the evaluation cost of an algorithm, either sort-merge, partition join, or nested-loops, for a given random/sequential cost ratio over varying main memory sizes.

The graph shows an interesting property of the partition-join algorithm. In contrast to nested-loops and sort-merge, the partition-join algorithm shows relatively good performance at all memory sizes, and, as expected, the performance of the algorithm improves

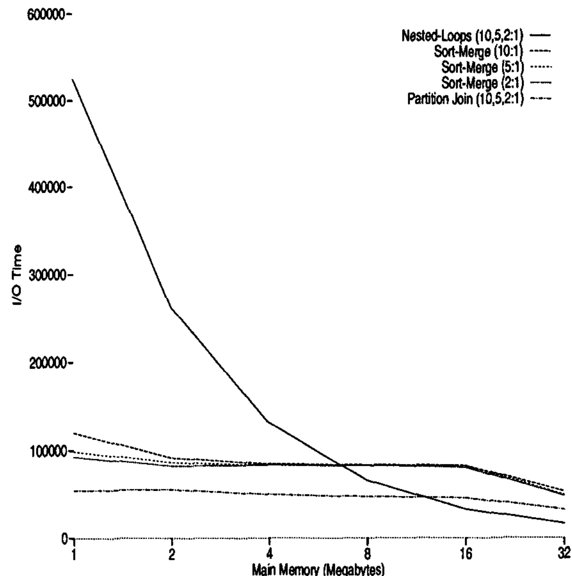

Figure 6: Performance Effects of Main Memory

as memory increases. Nested loops performs quite poorly at small memory allocations since few pages of the outer relation can be stored in memory, requiring many scans of the inner relation. At large memory allocations, e.g., 32 megabytes, the performance of nested-loops is quite good since a large portion of the outer relation remains resident in memory reducing the number of scans of the inner relation. We note also that the cost of reading the outer relation is quite low since if i pages of the outer relation are read, this requires a single random read followed by a $i-1$ sequential reads.

Comparing the partition join to sort-merge, we see that the partition join is approximately twice as fast as sort-merge at all memory sizes. As no backing up is performed by the sort-merge algorithm, we attribute this to the cost of sorting. At small memory sizes, the sort-merge algorithm must use more runs with fewer pages in each run, with a random access required by each run.

Similarly, when little main memory is available, partition sizes are necessarily small, and higher random access cost is incurred by the partition-join algorithm during both the sampling and partitioning phases. That is, not only are more samples required when the partitioning intervals are being determined, but, since less buffer space is available, the in-memory "buckets" must be flushed more often, requiring more random I/O. However, the effect on performance is not as drastic as for sort-merge since the partitioning phase requires only one pass through the relations, and we discovered an optimization that can reduce sampling costs.

We initially assumed that a random access is required for each sample. At large partition sizes, the effect is to perform a large number of random accesses during sampling, sometimes exceeding the number of pages in the outer relation. The algorithm instead se-

quentially scans the outer relation, drawing samples randomly when a page of the relation is brought into main memory. For example, at a random/sequential cost ratio of 10:1, only 819 random samples (3% of the relation) must be drawn before the entire outer relation can be scanned for the same cost. This requires only a single random access to read first page of the relation, followed by sequential reads of the remaining pages of the relation. The sampling cost is therefore proportional to the number of pages of the outer relation, as opposed to the number of sampled tuples which may be quite large.

4.3 Effects of Long-Lived Tuples

The presence of long-lived tuples adds another cost dimension to both the partition-join and sort-merge algorithms. The partition-join algorithm may incur paging of the tuple cache when many long-lived tuples are present, and the sort-merge algorithm may back-up to previously processed pages of the input relations to match overlapping tuples. Long-lived tuples do not affect the performance of the nested-loops algorithm, but it is included here for completeness.

We designed an experiment to empirically investigate the cost effect that long-lived tuples have on both strategies. A series of databases were generated with increasing numbers of long-lived tuples. Each database contained 32 megabytes (262144 tuples); we varied the number of long-lived tuples from 8000 to 128,000 in 8000 tuple steps. Non-long-lived tuples were randomly distributed throughout the relation lifespan with a one chronon long validity interval. Long-lived tuples had their starting chronon randomly distributed over the first 1/2 of the relation lifespan, and their ending chronon equal to the starting chronon plus 1/2 of the relation lifespan. To not influence the performance of the algorithms via main memory effects, we fixed the main memory allocation at 8 megabytes, the memory size at which all three algorithms performed most closely in the previous experiment. Additionally, the random to sequential I/O cost ratio was fixed at 5:1. The results of the experiment are shown in Figure 7.

As can be seen from the figure, the partition-join algorithm outperformed the sort-merge algorithm at all long-lived tuple densities. We expected this result. The tuple caching cost incurred by the partition-join algorithm is relatively low—the tuple cache size is small (it cannot exceed the size of a partition), and it is fairly inexpensive to read or write (a random access for the first page followed by sequential accesses for the remaining pages). Furthermore, many long-lived tuples do not significantly increase this cost since they merely cause additional pages to be appended to the tuple cache, and these pages incur an inexpensive sequential I/O cost.

In contrast, the presence of long-lived tuples greatly increases the cost of the sort-merge algorithm. To see this, consider what happens when a long-lived tuple is encountered during the matching phase. The tuple must be joined with all tuples that overlap it, some of these tuples may, unfortunately, have already been read, requiring the algorithm to re-read these pages. For tuples with lifespans of 1/2 the relation lifespan, this incurs a significant cost. Furthermore, the percentage of long-lived tuples is less significant to the sort-merge algorithm. While a higher density of long-lived tuples may require the algorithm to back-up more often, the presence of only a single long-lived tuple will still cause the sort-merge algorithm to back-up.

Figure 7: Performance Effects of Long-Lived Tuples

4.4 Main Memory vs. Long-Lived Tuples

The previous two experiments showed that the partition join exhibits better performance when more main memory is available, and incurs a performance penalty at increasing densities of long-lived tuples.

We desired to determine whether the allotted main memory size or the density of long-lived tuples played a larger effect on the performance of the partition-join algorithm, and designed an experiment to investigate this. Eight 262,144 tuple databases were generated with increasing numbers of long-lived tuples, from 16,000 to 128,000 in 16,000 tuple steps. A trial was run for each database at 1, 2, 4, 16, and 32 megabyte main memory allocations. The results are shown in Figure 8. (The x-axis is log-scaled in the figure.)

The graph shows that at large memory sizes (16 and 32 megabytes) the evaluation cost for all databases becomes fairly equal, hence the relative cost of tuple caching is small due to the large memory size. At smaller memory sizes, there is a more pronounced difference between the evaluation costs over the different databases. This was expected. When the allotted memory sizes are small the cost of tuple caching is significant since partition sizes are necessarily smaller and more tuples are likely to overlap multiple partitions. Again, the conclusion to be drawn is that main memory availability is necessary for the partition join to be efficient. When sufficient main memory is available, the effects of tuple caching become insignificant, but when insufficient main memory is available, the performance impact of tuple caching is significant.

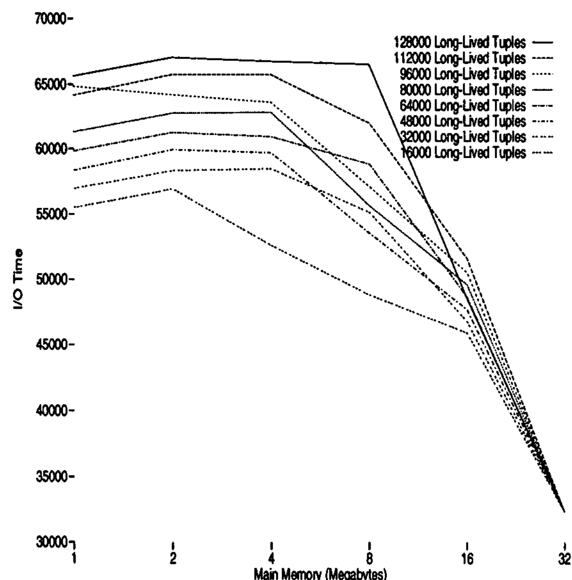

Figure 8: Relative Effects of Main Memory Size and Tuple Caching

4.5 Summary

We expected that the partition-join algorithm would be sensitive to the amount of main memory available during evaluation. The experiment of Section 4.2 confirms this hypothesis. The algorithm performed better at larger memory sizes, mainly due to the decreased random I/O during partitioning and the fewer samples required to determine the partitioning intervals. Furthermore, the partition join shows uniformly good performance at all memory sizes, unlike nested-loops which performs well at large memory sizes, but quite poorly at small memory sizes.

Relative to sort merge, the partition-join algorithm compares favorably. When long-lived tuples are present, the partition join outperforms sort-merge significantly, as shown in Section 4.3. Tuple caching in the partition join incurs a low cost relative to the high cost of backing-up in sort-merge.

Finally, in Section 4.4, we compared the relative costs of tuple caching and main memory availability. For the partition join, the density of long-lived tuples did not greatly increase the evaluation cost when sufficient main memory was available. Given that sufficient main memory is available, our conclusion is that the partition-join algorithm performs well relative to both nested-loops and sort-merge, both in the presence, and absence, of long-lived tuples.

5 Conclusions and Future Work

The contributions of this work are summarized as follows.

- We formally defined the valid-time natural join, the operator used to reconstruct normalized valid-time databases.
- We presented a new algorithm for valid-time join evaluation, improving on the $O(n^2)$ cost of nested-loop join while avoiding the $O(n \cdot log(n))$ cost of sorting.
- Our approach is based on tuple partitioning, but still avoids replication of tuples in multiple partitions, thereby allowing simple base relation updates.
- We compared the performance of the partition-join algorithm with both nested-loop and sort-merge, and showed that with adequate main memory our algorithm exhibits almost uniformly better performance, especially in the presence of long-lived tuples.

As relatively little work has appeared on temporal query evaluation, there are many directions in which this work can be expanded. First, many important problems remain to be solved with valid-time natural join evaluation. We made the simplifying assumption in Section 3.4 that the distribution of tuples over valid time was approximately the same for both the inner and outer relations. Obviously, this assumption may not be valid for many applications since gross mis-estimation of tuple caching costs may result. Secondly, while tuple caching is a relatively inexpensive operation, the paging cost associated with it can be reduced if sufficient buffer space is allocated to retain, with high probability, the entire tuple cache in main memory. Trading off outer relation partition space for tuple cache space is a possible solution to this problem. Lastly, while we have distinguished between the higher cost of random access and the lower cost of sequential access, we have ignored the cost of main-memory operations. Incorporating main-memory operations into the cost model would allow us to more accurately choose partitioning intervals through better estimates of evaluation costs.

More globally, this work can be considered as the first step towards the construction of an incremental evaluation system for a bitemporal database management system, that is, a DBMS that supports both valid and transaction time [SA86, JCG+92]. Elsewhere we motivate the importance of incremental evaluation to temporal database management systems and show how our partition-based approach is easily adapted to incremental evaluation [SSJ93].

Acknowledgements

Support for this work was provided the IBM Corporation through Contract #1124, the National Science Foundation through grants IRI-8902707 and IRI-9302244, and the Danish Natural Science Research Council through grant 11-9675-1 SE. We thank Nick Kline for providing the aggregation tree implementation used in the simulations.

References

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the Association for Computing Machinery*, 26(11):832–843, November 1983.

[CC87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537, Los Angeles, CA, February 1987.

[Con71] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, 1971.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proceedings of the Conference on Very Large Databases*, pages 443–452, 1991.

[DS93] C. E. Dyreson and R. T. Snodgrass. Timestamp Semantics and Representation. *Information Systems*, 18(3), September 1993.

[GS90] H. Gunadhi and A. Segev. A Framework for Query Optimization in Temporal Databases. In *Proceeding of the Fifth International Conference on Statistical and Scientific Database Management*, pages 131–147, Charlotte, NC, April 1990.

[GS91] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of the 7th International Conference on Data Engineering*, Kobe, Japan, 1991.

[JCG+92] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 21(3):35–43, September 1992.

[JMRS92] C. S. Jensen, L. Mark, N. Roussopoulos, and T. Sellis. Using Caching, Cache Indexing, and Differential Techniques to Efficiently Support Transaction Time. *VLDB Journal*, 2(1):75–111, 1992.

[JS92] C. S. Jensen and R. Snodgrass. Temporal Specialization. In *Proceedings of the International Conference on Data Engineering*, pages 594–603, Tempe, AZ, February 1992.

[JSS92a] C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Extending Normal Forms to Temporal Relations. TR 92-17, Computer Science Department, University of Arizona, July 1992.

[JSS93] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unification of Temporal Relations. In *Proceedings of the International Conference on Data Engineering*, Vienna, Austria, pages 262–271, April 1993.

[JSS93a] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. TR 93-31, Department of Computer Science, University of Arizona, September 1993.

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of Hash to Database Machine and its Architecture. *New Generation Computing*, 1(1), 1983.

[LM90] T. Y. Leung and R. Muntz. Query Processing for Temporal Databases. In *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, California, February 1990.

[LM92] T. Y. Leung and R. Muntz. Generalized Data Stream Indexing and Temporal Query Processing. In *Second International Workshop on Research Issues in Data Engineering: Transaction and Query Processing*, February 1992.

[LM92a] T. Y. Leung and R. Muntz. Stream Processing: Temporal Query Processing and Optimization. Chapter 14 of *Temporal Databases: Theory, Design, and Implementation*, Benjamim/Cummings, pp. 329–355, 1993.

[LM92b] T. Y. Leung and R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proceedings of the Conference on Very Large Databases*, August 1992.

[ME92] P. Mishra and M. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[SA86] R. T. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, September 1986.

[SG89] A. Segev and H. Gunadhi. Event-Join Optimization in Temporal Relational Databases. In *Proceedings of the Conference on Very Large Databases*, pages 205–215, August 1989.

[Sno90] R. T. Snodgrass. Temporal Databases: Status and Research Directions. *ACM SIGMOD Record*, 19(4):83–89, December 1990.

[Sno92] R. T. Snodgrass. *Temporal Databases*, Volume 639 of *Lecture Notes in Computer Science*, pages 22–64. Springer-Verlag, September 1992.

[Soo91] M. D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD Record*, 20(1):14–23, March 1991.

[SSJ93] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. TR 93-17, Department of Computer Science, University of Arizona, June 1993.

A Appendix

We describe in detail the algorithms used in Sections 3.3 and 3.4.

A.1 Algorithm *joinPartitions*

Algorithm *joinPartitions*, shown in Figure 9, computes $r \bowtie^V s$, assuming that r and s have been previously partitioned. For each i, $1 \leq i \leq n$, the algorithm constructs the next outer relation partition r_i by purging tuples in the outer relation partition buffer that do not overlap p_i and reading in the physical partition r_i from disk. r_i is then joined with the long-lived tuple cache. Tuples in the tuple cache that do not overlap p_{i-1} are purged after r_i and the tuple cache are joined. We check this by comparing a tuple's validity interval with the partitioning intervals. Finally, r_i is then joined with each page of s_i. Tuples in the current page of s_i that overlap p_{i-1} are inserted into the tuple cache to be available for the computation of $r_{i-1} \bowtie^V s_{i-1}$. In preparation for the next partition, tuples in r_i that overlap p_{i-1} are retained in the outer relation partition for the subsequent computation of $r_{i-1} \bowtie^V s_{i-1}$. We assume that the tuple cache is paged in and out of memory as necessary to compute the join.

The ordering of operations in algorithm *joinPartitions* attempts to minimize the amount of I/O, both random and sequential, performed during the evaluation. Each partition fetch of the outer relation requires a random seek, but subsequent pages are read with sequentially. Similarly, each page of the tuple cache and the inner partition are, after an initial seek, read nearly sequentially except when the result buffer requires flushing. The result buffer requires random writes in most cases. In all cases, reading of either the outer relation partition, inner relation partition, or the tuple cache normally requires only a single random seek followed by $i - 1$ sequential reads, where i is the number of pages in the item of interest.

Different orderings of the operations in algorithm *joinPartitions* are possible, but these alternatives result in higher evaluation cost through more random access, rereading of pages, or more complex bookkeeping. For example, prior to joining r_i with the tuple cache, we could join each r_i with each page of s_i, moving long-lived tuples in s_i to the tuple cache as pages of s_i are brought into main memory. Since $r_i \bowtie^V s_i$ is computed prior to the join of r_i and the tuple cache, the tuple cache contains tuples from s_i that have already been processed and, to prevent recomputation, more complex tuple management is required.

Other variations include migrating long-lived tuples from s_i to the tuple cache prior to performing any joins, and purging "dead" tuples from the tuple cache prior to joining it with the r_i. Both of these variants suffer from repeated reading of tuples. The former requires that s_i be read twice, first to migrate live tuples, then to join the remaining tuples with r_i. This requires an additional random access and $|s| - 1$ sequential reads. The latter requires that the tuple cache be read twice for each partition. While reading the tuple cache is not as expensive as reading a partition, this is unnecessary and should be avoided.

```
joinPartitions(r,s,partIntervals):
  cachePage ← ∅;
  outerPart ← ∅;
  tupleCache ← ∅;

  for i from n to 1
    for each tuple x ∈ outerPart
      if overlap(x[V],partIntervals_i) =⊥
        outerPart ← outerPart − {x};
    outerPart ← outerPart ∪ {read(r_i)};
    result_i ←
        result_i ∪ outerPart ⋈^V cachePage};

    for each tuple x ∈ cachePage
      if overlap(x[V],partIntervals_{i-1}) ≠⊥
        newCachePage ← newCachePage ∪ {x};
          if filled(newCachePage)
            write(newCachePage);

    for each flushed page c of tupleCache
      cachePage ← read(c);
      result_i ←
          result_i ∪ {outerPart ⋈^V cachePage};
      for each tuple x ∈ cachePage
        if overlap(x[V],partIntervals_{i-1}) ≠⊥
          newCachePage ← newCachePage ∪ {x};
            if filled(newCachePage)
              write(newCachePage);

    for each page o of s_i
      innerPage ← read(o);
      result_i ← result_i ∪ {outerPart ⋈^V o};
      for each tuple x ∈ o
        if overlap(x[V],partIntervals_{i-1}) ≠⊥
          newCachePage ← newCachePage ∪ {x};
          if filled(newCachePage)
            write(newCachePage);

  return result_1 ∪ ... ∪ result_n;
```

Figure 9: Algorithm *joinPartitions*

A.2 Algorithm *determinePartIntervals*

Algorithm *determinePartIntervals*, shown in Figure 10, determines the lowest cost partitioning of two input relations r and s given the buffer constraint *buffSize*. The algorithm differentiates between the higher cost of random disk access, as incurred during sampling, and sequential disk access, as incurred while reading the second to last pages of an outer relation partition.

C_{sample} is dependent only on *errorSize* = *buffSize* − *partSize*. For a given partition size *partSize*, C_{sample} is computed using the Kolmogorov statistic, and a sample set is drawn. Since the number of samples increases with partition size, we incrementally draw samples from r and add them to the sample set for increasing *partSize*. Sampling incurs a random I/O cost, and tuples are sampled without replacement; each tuple in the relation is equally likely to be drawn, and at most one time. The samples are used to determine the partitioning intervals, using procedure *chooseIntervals*, described in Appendix A.3, and estimate the tuple cache size for each partition, using procedure *estimateCache-*

Sizes, described in Appendix A.4. This estimate is a component of C_{join}, the cost of joining partitions. The cost of writing the result relation is omitted since this cost is incurred by all evaluation algorithms.

The set of partitioning intervals associated with the *partSize* minimizing the sum $C_{sample} + C_{join}$ is returned.

```
determinePartIntervals(buffSize,r,s):
    mincost ← ∞;
    oldSampleCount ← 0;
    samples ← ∅;

    for each partSize from 1 to buffSize
        errorSize ← buffSize - partSize;
        newSampleCount ← (1.63 × |r|/errorSize)²;
        C_sample ← newSampleCount × IO_ran;

        numPartitions ← |r|/partSize;
        samples ← samples∪
            drawSamples(r, newSampleCount−
                        oldSampleCount);
        partIntervals ←
            chooseIntervals(samples, numPartitions);
        cachePagesPerPartition ←
            estimateCacheSizes(samples, |r|,
                        partIntervals, numPartitions);

        C_join ← 2 × (numPartitions × IO_ran+
            (partSize − 1) × numPartitions × IO_seq);
        for each m in cachePagesPerPartition
            C_join ←
                C_join + 2 × (IO_ran + IO_seq × (m − 1));

        cost ← C_sample + C_join;
        if cost ≤ mincost
            mincost ← cost;
            result ← partIntervals;

    return result;
```

Figure 10: Algorithm *determinePartIntervals*

A.3 Algorithm *chooseIntervals*

Using the set of sampled tuples and the desired number of partitions, we can derive a set of partitioning intervals. This is the function of algorithm *chooseIntervals*, shown in Figure 11. For a given sample set, the chronons covered by any tuple in the sample set are collected,[3] and the range of time covered by the sample set is computed. If *numPartitions* is the computed number of partitions then the chosen chronons are those that appear in a sorting of the sample set at every *numPartitions* position. Adjacent pairs of the chosen chronons are then used to construct the partitioning intervals.

A.4 Algorithm *estimateCacheSizes*

Having determined the partitioning of the input relations, we are able to estimate the size of the tuple cache for each partition s_i of s. This is the function of procedure *estimateCacheSizes*, shown in Figure 12. Using the sampled tuples and the set of partitioning chronons, we can determine how many of these tuples overlap the given partition boundaries. For any partition, its estimated tuple cache size is simply the number of sampled tuples that overlap that partition with a scaling factor to account for the percentage of the relation sampled. The functions *earliestOverlap* and *latestOverlap* simply return the indexes of the earliest and latest partitions, respectively, that overlap the given tuple.

```
chooseIntervals(samples,numPartitions):
    chronons ← ∅;
    for each tuple x ∈ samples
        for each chronon t ∈ x[V]
            chronons ← chronons ∪ t;

    lifespan ← max(chronons) − min(chronons);
    chronons ← sort(chronons);
    partChronons ← ∅;
    m ← lifespan/numPartitions;

    while m ≤ lifespan
        partChronons ← partChronons ∪ chronon_m;
        m ← m + (lifespan/numPartitions);

    partIntervals ← ∅;
    for i from 1 to |partitionChronons| − 1
        partIntervals ← partIntervals∪
            {[partChronons_i, partChronons_{i+1}]};

    return partIntervals;
```

Figure 11: Algorithm *chooseIntervals*

```
estimateCacheSizes(samples,|r|,partIntervals,
                   numPartitions):
    for each interval p ∈ partIntervals
        cnt_p ← 0;

    for each tuple x ∈ samples
        min ← earliestOverlap(partIntervals, x[V]);
        max ← latestOverlap(partIntervals, x[V]);
        for each interval p from p_min to p_max − 1
            cnt_p ← cnt_p + 1;

    for each interval p ∈ partIntervals
        cachePages_p ← cnt_p × (|samples|/|r|);

    return cachePages;
```

Figure 12: Algorithm *estimateCacheSizes*

[3] In the algorithm, *chronons* is a multiset. Hence the union operation used to add chronons to the multiset is not strict set union.