Efficient Evaluation of the Valid-Time Natural Join

Michael D. Soo[1]
Richard T. Snodgrass[1]
Christian S. Jensen[2]

TR 93-17

June 7, 1993

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 inequality predicates which are 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. The algorithm is then adapted to an incremental mode of operation, which is especially appropriate for temporal query evaluation. Performance comparisons between the recomputation 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]Department of Computer Science
University of Arizona
Tucson, AZ 85721
{soo,rts}@cs.arizona.edu

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

Efficient Evaluation of the Valid-Time Natural Join

Contents

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 [Soo91, Sno90, 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 is likely to 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 [JCG+92, SA86]. In this paper, we consider strategies for evaluating the *valid-time natural join* [LM92, CC87], which matches tuples with identical attribute values during coincident time intervals. 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 categories, nested-loop, sort-merge, or partition-based [ME92]. The majority of previous work in temporal join evaluation has concentrated on refinements of nested-loop [GS90, SG89] 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 [LM91b].

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 most situations, our algorithm allows an $O(n)$ evaluation cost, thereby improving on the $O(n^2)$ cost of nested-loop evaluation and the $O(nlog(n))$ cost of sort-merge evaluation. We then adapt the algorithm to an incremental mode of computation [Han87, Rou87, Hor86], where the join is materialized and updated incrementally as the base relations change. We motivate why incremental evaluation is an appropriate strategy for valid-time join evaluation.

The paper is organized as follows. Section 2 formally defines the valid-time natural join in the valid-time conceptual data model (VCDM), a special case of the Bitemporal Conceptual Data Model [JSS93]. 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. Section 5 adapts the partition-based algorithm to an incremental framework, and conclusions and future work are detailed in Section 6.

2 Valid-Time Natural Join

In this section, we define the valid-time natural join using the tuple relational calculus. Two definitions are provided. The first uses the valid-time conceptual data model [JSS93]. The second is an equivalent definition in a common representational model.

2.1 Valid-Time Conceptual Data Model

The valid-time conceptual data model employs tuple timestamping. Each tuple in a valid-time relation consists of a set of first normal form attribute values and a timestamp. The timestamp, termed a *valid-time element* and denoted V, is a set of chronons [DS93] representing a set of time intervals.

EXAMPLE: We define two valid-time relation schemas $EmpSal = (Emp, Sal \mid \text{V})$ and $EmpDep = (Emp, Dep \mid \text{V})$. Instances $empSal(EmpSal)$ and $empDep(EmpDep)$ follow.

empSal:

Emp	*Sal*	V
Al	10	{30,31,33,...,40}
Al	11	{32,41,42,...,48}

empDep:

Emp	*Dep*	V
Al	Ship	{30,...,35}
Al	Load	{36,...,48}

The relation *empSal* shows Al's salary history. Between times 30 to 31 inclusive, Al's salary was 10. This is shown by the first tuple in the relation. At time 32, Al received a raise. His new salary, 11, is shown in the second tuple. At time 33, Al's raise is revoked, perhaps due to the present economic unrest, and his salary remains at 10 until time 40. This is shown in the first tuple. Finally, at time 41, Al receives the raise once again.

Note that the timestamps are truly sets of chronons rather than contiguous intervals of time, and that the timestamps effectively represent sets of intervals. For example, the timestamp of the first tuple is equivalent to the two intervals [30,31] and [33,40]. □

Let R and S be valid-time relation schemas

$$\begin{aligned} R &= (A_1, \ldots, A_n, B_1, \ldots, B_m \mid \text{V}) \\ S &= (A_1, \ldots, A_n, C_1, \ldots, C_k \mid \text{V}) \end{aligned}$$

where the A_i represent explicit join attributes, the B_i and C_i are additional attributes, and V is the valid timestamp. Also, let r and s be instances of R and S, respectively.

In the valid-time natural join, two tuples x and y join if they match on the explicit join attributes and they have overlapping valid timestamps. The attribute values of the resulting tuple z are as in the snapshot natural join, with the addition that the valid timestamp is the intersection of the valid timestamps of x and y.

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

$$\begin{aligned} r \bowtie^{\text{V}} s = \{z^{(n+m+k+1)} \mid \exists x \in r\ \exists y \in s(& x[A] = y[A] \wedge x[\text{V}] \cap y[\text{V}] \neq \emptyset \wedge \\ & z[A] = x[A] \wedge z[B] = x[B] \wedge z[C] = y[C] \wedge \\ & z[\text{V}] = x[\text{V}] \cap y[\text{V}])\}. \end{aligned}$$ □

EXAMPLE: Define $emp = empSal \bowtie^{\text{V}} empDep$ with schema $Emp = (Emp, Sal, Dep \mid \text{V})$.

emp:

Emp	*Sal*	*Dep*	V
Al	10	Ship	{30,31,33,34,35}
Al	10	Load	{36,...,40}
Al	11	Ship	{32}
Al	11	Load	{41,...,48}

□

Other terms for the valid-time natural join include the *natural time-join* [CC87] and the *time-equijoin* (TE-join) [GS90].

2.2 A Tuple-Timestamped Representation

The valid-time conceptual data model provides an intuitive and simple way to express temporally varying information. However, it is not a convenient vehicle for implementation due to its non-first normal form timestamp. Therefore, we define a representation for valid-time relations more suited for query evaluation, and define the valid-time natural join over this representation. It can be shown that relations in this representation have snapshot equivalents in the VCDM [JSS93].

Like the VCDM, the representation we choose uses tuple-timestamping. The timestamp, rather than being a set of chronons, is restricted to be a single interval of time denoted by inclusive starting and ending chronons.

Let R and S be valid-time relation schemas

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

where the A_i represent the joining attributes, the B_i and C_i are additional attributes, and V_s and V_e are the valid time start and end attributes. We use V to denote the interval $[\mathrm{V}_s, \mathrm{V}_e]$. Also, we define r and s to be instances of R and S, respectively.

EXAMPLE: We define two valid-time relation schemas $EmpSal = (Emp, Sal, \mathrm{V}_s, \mathrm{V}_e)$ and $EmpDep = (Emp, Dep, \mathrm{V}_s, \mathrm{V}_e)$. Relations $empSal(EmpSal)$ and $empDep(EmpDep)$, which are snapshot equivalent to the VCDM relations of the previous example follow.

empSal:

Emp	*Sal*	V_s	V_e
Al	10	30	31
Al	11	32	32
Al	10	33	40
Al	11	41	48

empDep:

Emp	*Dep*	V_s	V_e
Al	Ship	30	35
Al	Load	36	48

This representation encodes the same information as the VCDM versions given earlier. □

In the valid-time natural join, two tuples x and y join if they meet 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 $overlap(x[\mathrm{V}], y[\mathrm{V}])$ returns the maximal interval contained in both intervals $x[\mathrm{V}]$ and $y[\mathrm{V}]$.

$overlap(x[\mathrm{V}], y[\mathrm{V}]) = [i,j]$ where $\forall t(x[\mathrm{V}_s] \leq t \leq x[\mathrm{V}_e] \wedge y[\mathrm{V}_s] \leq t \leq y[\mathrm{V}_e] \rightarrow i \leq t \leq j$ □

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

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

EXAMPLE: Let $empSal$ and $empDep$ be as in the previous example, and let $emp = empSal \bowtie^{\mathrm{V}} empDep$. Then emp, with schema $Emp = (Emp, Sal, Dep, \mathrm{V}_s, \mathrm{V}_e)$, is given by

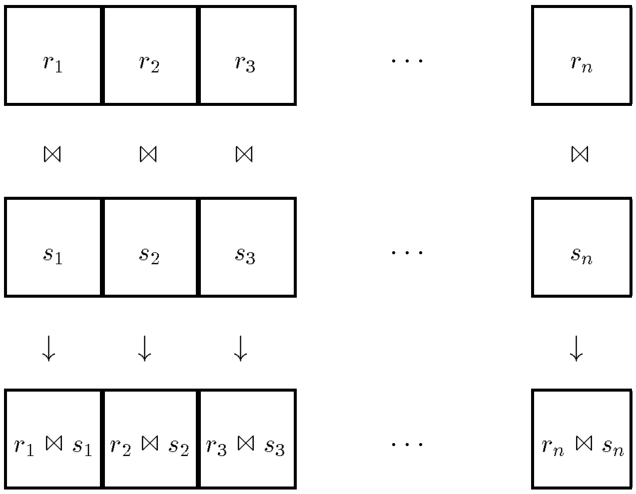

Figure 1: Partition join of $r \bowtie s$

emp:

Emp	*Sal*	*Dep*	V_s	V_e
Al	10	Ship	30	31
Al	10	Ship	33	35
Al	11	Ship	32	32
Al	10	Load	36	40
Al	11	Load	41	48

This clearly has the same information content as the previous VCDM instance. □

3 Valid-Time Partition Join

Partition joins cluster tuples with similar attribute values, thereby reducing the amount of unnecessary comparison needed to find matching tuples [ME92]. Both input relations are divided into partitions where tuples in a particular partition can only match with tuples in a corresponding partition of the other relation, and 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 cost 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 the 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 relatively inexpensive sequential reads. 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 $1 \leq i \leq n$ depending on their joining attribute values. The partitioning is performed to guarantee that, for any tuple $x \in r_i$, x can only join with tuples in s_i. The join $r \bowtie s$ is computed 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 $r_i \bowtie s_i$.

3.1 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 interval timestamped representation presented in Section 2.2.

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 [LM91b]. However, the disadvantages of this include additional secondary storage space to hold the replicated tuples and complications in update for incremental evaluation.

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 attributes values that determine partition boundaries are determined. This is performed by procedure *determinePartitions*. Next the relations are partitioned by procedure *createPartitions*, and lastly, the partitioned relations are joined by procedure *joinPartitions*.

$partitionJoin(r, s)$:
  $partitions \leftarrow determinePartitions(buffSize, |r|, |s|)$
  $r \leftarrow createPartitions(r, partitions)$
  $s \leftarrow createPartitions(s, partitions)$
  $joinPartitions(r, s, partitions)$

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

We assume that Grace partitioning [KTMo83, ME92] is used in procedure *createPartitions*. The available buffer space is divided 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

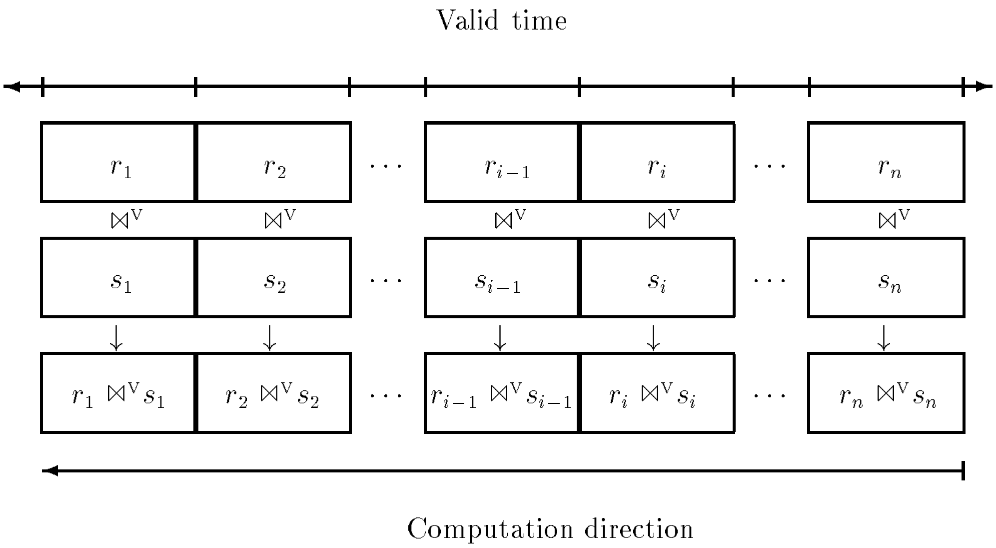

Figure 3: Partition-join of $r \bowtie^{\mathrm{V}} s$

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 3.3 the details of procedure *determinePartitions*.

3.2 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. We assume, for the purposes of this section, that each p_i has approximately the same number of tuples.

Figure 3 shows our approach. We assume that a tuple x is in the partition r_i if and only if $overlap(x[\mathrm{V}], p_i) \neq \emptyset$, 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 $overlap(x[\mathrm{V}], p_i) \neq \emptyset$ and $\neg\exists j$ such that $j > i$ and $overlap(x[\mathrm{V}], p_j) \neq \emptyset$.[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 r_{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_i, p_{i+1}, $\ldots$, p_j then x must be present in r_i, r_{i+1}, $\ldots$, r_j when their corresponding join is computed. Hence, no unnecessary comparisons are performed.

The buffer allocation strategy used in this algorithm is shown in Figure 4. 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.

Figure 5 provides the algorithm to compute $r \bowtie^{\mathrm{V}} s$. 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

[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 the incremental modifications to be described in Section 5.

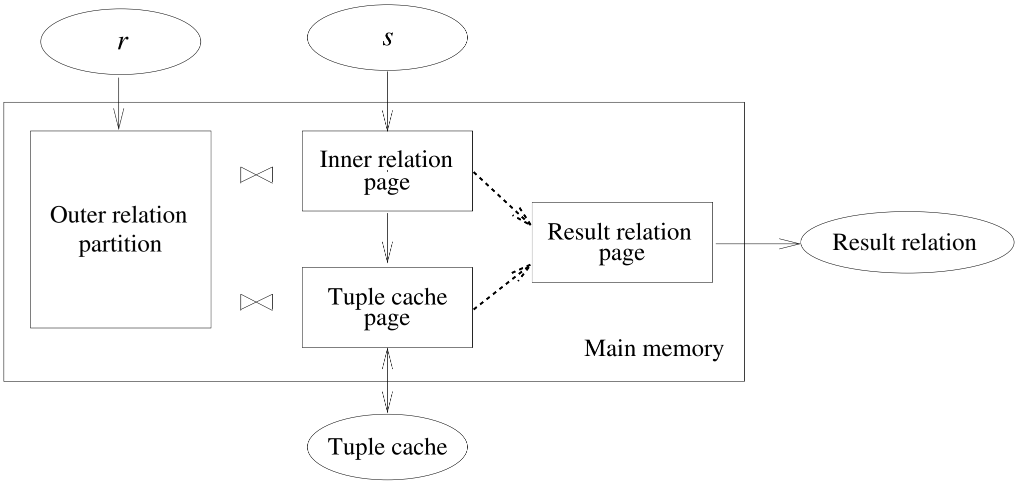

Figure 4: Buffer Allocation Strategy for $r \bowtie^{\mathrm{V}} s$ Evaluation

after r_i and the tuple cache are joined. We check this by comparing a tuple's validity interval with the partition boundary timestamps. 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 $r_{i-1} \bowtie^{\mathrm{V}} s_{i-1}$. 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 $r_{i-1} \bowtie^{\mathrm{V}} s_{i-1}$. 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 Figure 5 are possible, but either 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 $r_i \bowtie^{\mathrm{V}} s_i$ 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,partitions):
   cache_page ← ∅
   outer_partition ← ∅
   tuple_cache ← ∅
   for i from n to 1 do
      for each tuple x in outer_partition do
         if overlap(x,p_i) = ∅ then
            outer_partition ← outer_partition - {x}
      outer_partition ← outer_partition ∪ {read(r_i)}
      result_i ← result_i ∪ {outer_partition ⋈^V cache_page}
      for each tuple x ∈ cache_page do
         if overlap(x,p_{i-1}) ≠ ∅ then
            new_cache_page ← new_cache_page ∪ {x}
               if filled(new_cache_page) then
                  write(new_cache_page)
      for each flushed page c of tuple_cache do
         cache_page ← read(c)
         result_i ← result_i ∪ {outer_partition ⋈^V cache_page}
         for each tuple x ∈ cache_page do
            if overlap(x,p_{i-1}) ≠ ∅ then
               new_cache_page ← new_cache_page ∪ {x}
                  if filled(new_cache_page) then
                     write(new_cache_page)
         for each tuple x ∈ c do
            if overlap(x,p_{i-1}) ≠ ∅ then
               new_cache_page ← new_cache_page - {x}
                  if filled(new_cache_page) then
                     write(new_cache_page)
      for each page o of s_i do
         inner_page ← read(o)
         result_page ← result_page ∪ {outer_partition ⋈^V o}
         if filled(result_page) then
            write(result_page)
         for each tuple x ∈ o do
            if overlap(x,p_{i-1}) then
               cache_page ← cache_page ∪ {x}
```

Figure 5: Joining Partitions

3.3 Partitioning Strategies

In the previous section, we described how the join of two partitioned relations was computed, assuming that the partitions 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 relative small I/O cost. Our method is inspired by the partition size estimation technique originally developed for evaluation of *band-joins* [DNS91].

In Figure 4, 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 that this is the case.

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. A simple strategy would be to sort r and perform a linear scan of the sorted relation, choosing the chronons that delimit the partitioning intervals. While this allows us to exactly choose the partitioning intervals, it is expensive since it requires sorting.

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 for overflow, that is, to handle errors in the chronon choices that would likely result in overflow of the buffer space. 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. We make the simplifying assumption that the distribution of tuples over valid time is similar for both the inner and outer relations, and so sampling is required of only one relation. We arbitrarily choose to sample the outer relation.

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,

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

The choice of partitioning intervals affects both C_{sample} and C_{join}. We note that the cost of performing Grace partitioning is dependent only main memory buffer size, and is independent of partition size. Let $partSize \leq buffSize$ be the estimated size of an outer relation partition. Then $errorSize = buffSize - partSize$ is the amount of buffer space available to handle overflow if a partition exceeds the estimated size. If $partSize$ is large then the amount of error space is small, implying that the accuracy of our choice of partitioning intervals must be high to avoid overflow. A larger number of samples must be taken to ensure higher accuracy, and therefore C_{sample} increases as $partSize$ increases. However, tuples are less likely to overlap multiple partitions if the partitioning intervals are large, hence a large $partSize$ is likely to result in fewer long-lived tuples. Consequently, paging of the tuple cache is less likely to occur thereby decreasing C_{join}. In summary, a cost tradeoff is present between the amount of sampling performed on the outer relation and the amount of paging performed on the tuple cache. The optimal solution minimizes the sum $C_{sample} + C_{join}$.

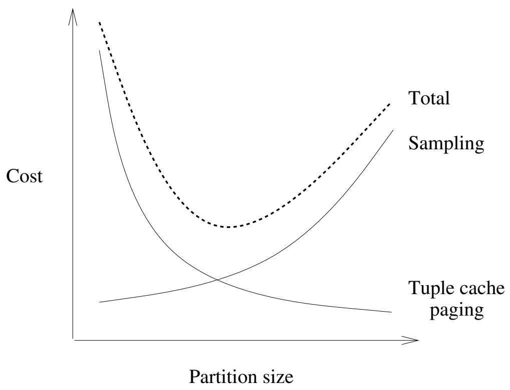

Figure 6: I/O Cost for Partition Size

Figure 6 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.

The algorithm of Figure 7 determines, for a given $buffSize$, the $partSize$ and $errorSize$ that minimize $C_{sample} + C_{join}$. The set of partitioning intervals is returned as its result.

The algorithm simply examines possible partition sizes $partSize$ given the buffer constraint $buffSize$ and estimates the evaluation cost for $partSize$. For a given $partSize$, and hence $errorSize$, the cost of sampling is computed using the Kolmogorov test statistic [Con71, DNS91]. The Kolmogorov test statistic 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 $errorSize \leq (1.63 \times |r|)/\sqrt{m}$ which implies that $m \geq ((1.63 \times |r|)/j)^2$ samples must be drawn.[2]

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. We note that, for a given partition size $partSize$, C_{sample} is fixed and independent of the contents of r. However, the tuple-cache paging costs are dependent on the contents of r. If few long-lived tuples are 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 than in the former case.

As the number of samples increases with partition size, we incrementally draw samples from r and add them to the sample set for increasing $partSize$. We assume that a random disk access is required for each sample, that is, only one sample tuple is drawn for each page randomly read from memory. We make the simplifying assumption that sampling is done without replacement, so that each tuple in the relation is equally likely to be drawn, and at most one time. The samples are used to determining the partitioning intervals and estimate the tuple cache size for each partition.

[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 $|r|$.

```
determinePartitions(buffSize, |r|, |s|):
    mincost ← ∞
    oldSampleCount ← 0
    chrononCount ← ∅
    samples ← ∅

    for each partSize from 1 to buffSize do
        errorSize ← buffSize - partSize
        newSampleCount ← (1.63*|r|/errorSize)^2
        C_sample ← newSampleCount * IO_random

        numPartitions ← |r|/partSize
        samples ← samples ∪ drawSamples(newSampleCount - oldSampleCount)
        partitions ← determinePartitioning(samples, partSize, |r|, numberOfPartitions)
        cachePagesPerPartition ← estimateCacheSizes(samples, |r|, partitions, numberOfPartitions)

        C_join ← numPartitions*IO_random + (partSize-1)*IO_seq + |s|*IO_seq +
            (numPartitions-1)*IO_random + numPartitions*IO_random
        for each m in cachePagesPerPartition do
            C_join ← C_join + 2*(IO_random + IO_seq*(m-1))

        cost ← C_sample + C_partition + C_join
        if cost ≤ cost_min
            cost_min ← cost
            result ← partitions

    return result
```

Figure 7: Determining Partitioning Chronons

These estimates are components of C_{join}, the cost of joining partitions. The partitioning cost, $C_{partition}$, includes sequentially reading r and s and randomly writing pages of r and s to disk as partition pages become full. The cost of writing the result relation is omitted since this cost is incurred by all evaluation algorithms.

With the set of sampled tuples, we are able to determine a set of partitioning intervals used in the partitioning of r and s. This is the function of procedure *determinePartitioning*, shown in Figure 8. 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 *numberOfPartitions* is the computed number of partitions then the chosen chronons are those that appear in a sorting of the sample set at every *numberOfPartitions* position. Adjacent pairs of the chosen chronons are then used to construct the partitioning intervals.

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 9. 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

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

```
determinePartitioning(samples,partSize,|r|,numberOfPartitions):
    chronons ← ∅
    for each tuple x ∈ samples do
        for each chronon t ∈ x[V] do
            chronons ← chronons ∪ t

    range ← max(chronons) - min(chronons)
    chronons ← sort(chronons)
    partitioningChronons ← ∅
    m ← range / numberOfPartitions

    while m ≤ range do
        partitioningChronons ← partitioningChronons ∪ chronon_m
        m ← m + (range / numberOfPartitions)

    partitioningIntervals ← ∅
    for partSize from 1 to |partitionChronons| - 1 do
        partitioningIntervals ← partitioningIntervals ∪ {[partitionChronons_i, partitioningChronons_{i+1}]}

    return partitioningIntervals
```

Figure 8: Determining Partition Intervals

overlap the given tuple.

We note that accurately estimating the amount of tuple cache paging is not as performance-critical as estimating the size of the outer relation partition. We assume that partitions are large; therefore, rereading partitions will incur a large expense. However, for any given partition, the size of its tuple cache is bounded by the number of tuples in the partition—for many applications the tuple cache will contain a relatively small percentage of the partition. Hence, rereading of the tuple cache will not impact performance as severely as rereading of a partition. This provides justification for the simple strategy of Figure 9, which does not employ a sophisticated technique such as the Kolmogorov test, or even incur the expense of sampling the inner relation itself.

3.4 Summary

We have presented a partition-join algorithm to compute $r \bowtie^{\mathrm{V}} s$, the valid-time natural join of two valid-time relations r and s. The key aspect of the algorithm is that tuples valid simultaneously are clustered into partitions, allowing, in most cases, an $O(n)$ evaluation cost.

Another advantage is that this algorithm is easily adapted to incrementally evaluate $r \bowtie^{\mathrm{V}} s$ as updates occur to r or s, as will be seen in Section 5.

4 Performance

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* [LM91a], 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*

```
estimateCacheSizes(samples, |r|, partitions, numberOfPartitions):
    for each partition p from 1 to numberOfPartitions do
        count_p ← 0

    for each tuple x ∈ samples do
        min ← earliestOverlap(partitions, x[V])
        max ← latestOverlap(partitions, x[V])
        for each partition p from p_min to p_max-1 do
            count_p ← count_p + 1

    for each partition p ∈ partitions do
        cachePages_p ← count_p * (|samples| / |r|)

    return cachePages
```

Figure 9: Tuple Cache Size Estimation

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. Leung and Muntz investigated partition-based valid-time joins 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 the partition join along with sort-merge join and conducted a series of experiments. In addition, analytical results were calculated 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 global parameters used in all of the experiments is shown in Figure 10.

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 10: 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 historical information are recorded for each real-world entity, then the database contains approximately 26,000 objects, a reasonably large number. 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.

4.1 Sensitivity to Main Memory Buffer Size

In Section 3.3, we analytically showed 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 performance of sort-merge join at equivalent main memory allocations.

The tuples in the database were randomly distributed over the lifetime 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. The presence of long-lived tuples causes paging of the tuple cache for the partition join algorithm and "backing-up" 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 cost ratio between random and sequential access was varied from 2:1, 5:1, and 10:1. The results of the experiments are shown in Figure 11. Each curve in the figure 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.

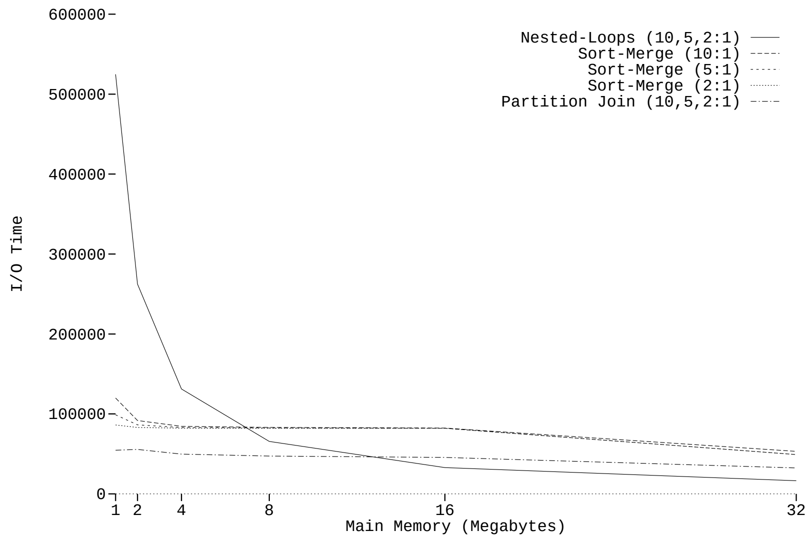

Figure 11: Performance Effects of Main Memory

The graph shows an interesting property of the 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 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 more random I/O is being performed during partitioning since less buffer space is available and the in-memory "buckets" must be flushed more often. However, the effects are not as drastic 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 sequentially 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.

4.2 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. These databases each contained 32 megabytes (262144 tuples), but varied the number of long-lived tuples in each 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 12.

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

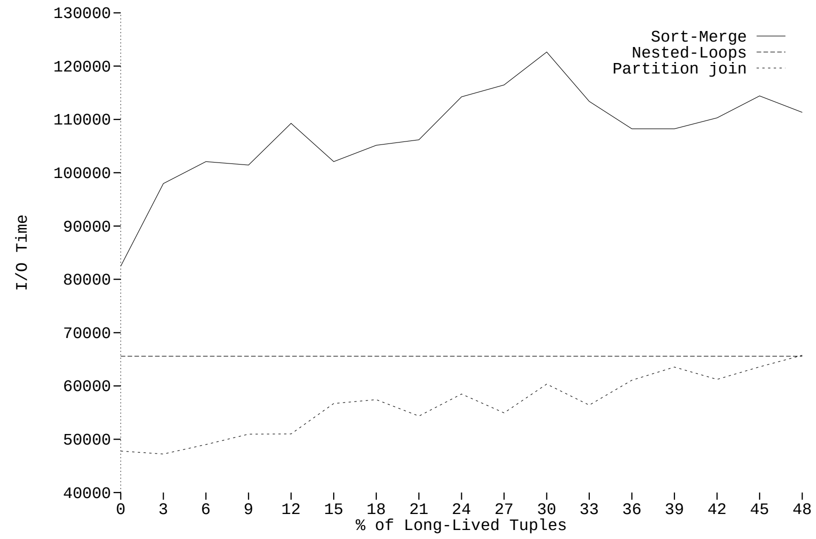

Figure 12: Performance Effects of Long-Lived Tuples

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 number 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.

4.3 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 13.

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 fairly large difference between the evaluation costs over the different databases. This was to be expected also. 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 paritition 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.

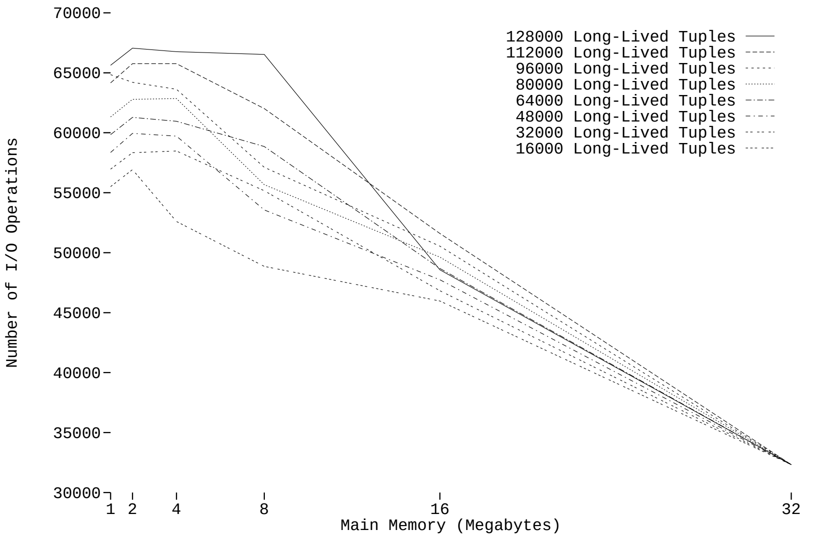

Figure 13: Relative Effects of Main Memory Size and Tuple Caching

4.4 Summary

We expected that the partition join algorithm would be sensitive to the amount of main memory available during evaluation. These experiments bear out this hypothesis. Relative to the sort-merge algorithm, the partition join algorithm compares favorably. The partition join outperforms sort merge significantly, especially in the presence of long-lived tuples. Tuple caching in the partition join incurs a relatively low cost relative to the high cost of backing-up in sort-merge. Finally, when comparing the cost of tuple caching versus the amount of main memory available, we conclude that the density of long-lived tuples does not greatly increase the evaluation cost when sufficient main memory is available. 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. Of the three algorithms, the partition join shows the best performance over all memory buffer sizes and in the presence of long-lived tuples.

5 Incremental Evaluation

Incremental query evaluation attempts to reduce the amount of recomputation performed by the DBMS. In this strategy, a query is materialized as a view, and once computed, it is stored back in the database for later use [Rou87, BCL86, SRMF87, Han87]. When an update occurs to a base relation, differentials are propagated to the views that depend on that base relation, incrementally ensuring the consistency of the view. Applications of incremental query evaluation have been found in the areas of smart editing [HT86], mainframe-workstation database architectures [RK86b, RK86a, Rou91], and monitoring [Sno88].

Hanson and Roussopolous independently showed that incremental view materialization offers superior performance over recomputation if five conditions are simultaneously satisfied [Han87, Han88, Rou87].

1. The number of queries on the view is sufficiently higher than the number of updates to the underlying base relations.

2. The sizes of the underlying base relations are sufficiently large.

3. The selectivity factor of the view predicate is sufficiently low.

4. The percentage of the view retrieved by queries is sufficiently high.

5. The volatility of the underlying base relations is sufficiently low.

The justification for incremental query processing in the valid-time setting depends, to a large extent, on the semantics of valid time. As real-world objects evolve, their changes are appended to the valid-time database with the result that the eventual size of valid-time relations will be quite large as historical information accumulates. Furthermore, it is reasonable to expect most queries to involve current data, that is, the same queries that would be posed to a snapshot database are likely to be asked of the valid-time database. Furthermore, as a percentage of the size of the database, we expect the current states to be relatively small, especially after some time has passed and historical information has accumulated in the database. Finally, one may assume a query mix, that is, updates vs. retrievals, that approximates that of a snapshot database since, for current data, identical ratios exist independent of the temporal support provided and corrections to past states of the database are expected to be infrequent.

Hence, valid-time databases satisfy all five conditions, and thus are excellent candidates for incremental evaluation. This section addresses a small but critical aspect of this problem, the incremental evaluation of the valid-time natural join.

5.1 Overview

We assume that the outer relation r and the inner relation s have been partitioned and $r \bowtie^{\mathrm{V}} s$ has been computed by joining the corresponding partitions of r and s. Furthermore, we assume that $r \bowtie^{\mathrm{V}} s$ is physically stored as the collection of partitioned results $r_i \bowtie^{\mathrm{V}} s_i$, $1 \leq i \leq n$, and pages of a partition r_i, $1 \leq i \leq n$, are stored contiguously on disk, but separate partitions r_i, r_j, $i \neq j$, are not. Similar assumptions are made for s and $r \bowtie^{\mathrm{V}} s$.

We examine the problem of reflecting updates to r or s into incremental changes in the materialized result, $r \bowtie^{\mathrm{V}} s$. As shown in Figure 14, there are six cases to consider depending on the type of update and whether the update occurs to r or s.

$$\left\{ \begin{array}{c} \texttt{insert} \\ \texttt{delete} \\ \texttt{update} \end{array} \right\} \times \left\{ \begin{array}{l} \text{outer relation } r \\ \text{inner relation } s \end{array} \right\}$$

Figure 14: Cases for Incremental Evaluation

As shown in the figure, three types of changes can occur to the base relations. A tuple can either be inserted, deleted, or updated. Each of these changes can be applied to either the outer relation r or the inner relation s resulting in six different situations where the result may require updating.

Note that an update operation is logically equivalent to deleting an existing tuple and inserting a new one. As such, for either the outer or inner relations, the update operation can be implemented by performing a deletion followed by a subsequent insertion, or more efficiently, an algorithm that performs both operations concurrently. For this reason, we omit further discussion of the update operation and concentrate on the insertion and deletion operations.

5.2 Outer Relation Operations

As described in Section 3.2, when a tuple x is inserted into the outer relation r, it is physically stored in the last partition that it overlaps. New tuples may result in $r \bowtie^V s$ due to the insertion of this tuple. Clearly, these new result tuples can only be produced by joins involving x.

The tuples $y \in s$ that can join with x are those that can overlap x in valid time. Let the given partitioning be $p_1, p_2, \ldots, p_n$, and suppose that $x[V_s] \in p_{min}$ and $x[V_e] \in p_{max}$. Then the only tuples in s that can overlap x must belong to one of the partitions $s_{min}, \ldots, s_{max}$. As tuples are physically stored in their last overlapping partition, it is possible that long-lived tuples overlapping $p_{min}, \ldots, p_{max}$ are stored in s_j, $max < j \leq n$. We therefore must scan the partitions $s_{max+1}, \ldots, s_n$ to collect long-lived tuples overlapping p_{max}. The result is then brought up to date by simply joining x with the tuple cache and each partition s_i, $min \leq i \leq max$, with the resulting tuples placed in the partitioned results $r_i \bowtie^V s_i$ as appropriate. We note that only three pages of main memory are required—one to hold a page of s, one for the tuple cache, and one to hold result tuples. The algorithm is shown in Figure 15.

```
insertOuter(x, s, partitions):
    min ← earliestOverlap(partitions, x[V])
    max ← latestOverlap(partitions, x[V])

    for each i from n to max do
        for each page p of s_i do
            inner_page ← read(p)
            for each tuple y ∈ inner_page do
                if overlap(y, x[V]) ≠ ∅ then
                    cache_page ← cache_page ∪ {y}

    for each page c of tuple_cache do
        cache_page ← read(c)
        result_i ← result_i ∪ {{x} ⋈^V cache_page}

    for each i from max to min do
        for each page p of s_i do
            inner_page ← read(p)
            result_i ← result_i ∪ {{x} ⋈^V inner_page}
```

Figure 15: Incremental Evaluation on Outer Relation Insert

Deletion of a tuple x from r causes result tuples in $r \bowtie^V s$ to be deleted, if there were any tuples $y \in s$ that joined with x. Deletion therefore involves determining the tuples in $r \bowtie^V s$ produced by x and removing them from the result.

The deletion algorithm is shown in Figure 16. As in algorithm *insertOuter* we compute the set of result tuples by first collecting all long-lived tuples of s overlapping $x[V]$ into the tuple cache. The result tuples for each partition of s overlapped by $x[V]$ is then computed and deleted from the result.

5.3 Inner Relation Operations

As with insertion into the outer relation, the insertion of a tuple y into the inner relation s may generate additional result tuples, and the tuples $x \in r$ that can join with y are those that overlap y in valid time. Again assume that the given partitioning is $p_1, p_2, \ldots, p_n$, and suppose that

```
deleteOuter(x, s, partitions):
    min ← earliestOverlap(partitions, x[V])
    max ← latestOverlap(partitions, x[V])

    for each i from n to max do
        for each page o of s_i do
            inner_page ← read(o)
            for each tuple y ∈ inner_page do
                if overlap(y, x[V]) ≠ ∅ then
                    cache_page ← cache_page ∪ {y}

    for each page c of tuple_cache do
        cache_page ← read(c)
        delete_page ← delete_page ∪ {{x} ⋈^V c}

    for each i from max to min do
        for each page o of s_i do
            inner_page ← read(o)
            delete_page ← delete_page ∪ {{x} ⋈^V inner_page}
        for each page q of result_i do
            result_page ← read(q)
            for each tuple d ∈ delete_page do
                if d ∈ result_page then
                    result_page ← result_page - {d}
```

Figure 16: Incremental Evaluation on Outer Relation Delete

$y[V_s] \in p_{min}$ and $y[V_e] \in p_{max}$. We must join y with all tuples x in r_{min}, ..., r_{max}, and in particular, we must ensure that long-lived tuples overlapping p_{min}, ..., p_{max} are present in the outer relation partition buffer when the join of that partition with y is computed.

We can ensure this by scanning all partitions r_i, $max < i \leq n$, and retaining all long-lived tuples in the outer relation partition buffer. Since the outer relation partition size has been estimated to fit in the available buffer space, we are assured that the partitions r_{min}, ..., r_{max} will not overflow the buffer space. The join can then proceed as in the manner of Figure 5. Buffer space is required for the outer relation partition and a page to hold result tuples. The actual algorithm is shown in Figure 17.

Deletion of a tuple y from s causes result tuples in $r \bowtie^V s$ to be deleted, if there were any tuples $x \in s$ that joined with y. Deletion therefore involves determining the tuples in $r \bowtie^V s$ produced by y and removing them from the result.

The deletion algorithm is shown in Figure 18. The set of result tuples is computed by joining y with all partitions of r that overlap in valid time. The result tuples for each partition are then deleted from the result.

5.4 Summary

We expect incremental query evaluation to be an important feature of temporal database management systems. A partition-based approach is especially well-suited to incremental evaluation since partitioning provides a natural mechanism for limiting the amount of recomputation necessary when base relation updates occur. Also, the particular evaluation algorithm we have presented does not require replication of tuples on secondary storage, thereby simplifying base relation up-

$insertInner(x, s, partitions)$:
    $min \leftarrow earliestOverlap(partitions, x[\mathrm{V}])$
    $max \leftarrow latestOverlap(partitions, x[\mathrm{V}])$

    for each i **from** n **to** max **do**
        $outer\_partition \leftarrow outer\_partition \cup \{read(r_i)\}$
        for each tuple $x \in outer\_partition$ **do**
            if $overlap(x, y[\mathrm{V}]) \neq \emptyset$ **then**
                $outer\_partition \leftarrow outer\_partition - \{x\}$
        $result_{max} \leftarrow result_{max} \cup \{\{x\} \bowtie^{\mathrm{V}} outer\_partition$

    for each i **from** max **to** min **do**
        $outer\_partition \leftarrow \mathtt{read}(r_i)$
        $result_i \leftarrow result_i \cup \{\{x\} \bowtie^{\mathrm{V}} outer\_partition$

Figure 17: Incremental Evaluation on Inner Relation Insert

date.

The storage of long-lived tuples in their latest overlapping partition leads to an interesting performance related observation. If we assume that corrections to information already present in the database are rare, then most updates to r and s will be the insertion of tuples into the currently valid partition, that is, the partition overlapping NOW. In such situations, only the last result partition must be recomputed. Of course, this is an application dependent situation, but it is expected that application semantics such as these are commonly found and can be exploited during query evaluation [JS92].

Furthermore, as another performance relation observation, we note most disk reads will be sequential rather than random. By organizing all pages of a partition contigously on disk, we are able to read a partition by performing a single random seek to the start of the partition then sequentially reading the remaining pages. Further performance improvement could be obtained by storing the partitions themselves contiguously, though this may increase the amount of reorganization needed by the database.

6 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 linear time algorithm for valid-time join evaluation, improving on the $O(n^2)$ cost of nested loop join while avoiding the $O(nlog(n))$ 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 our algorithm with the nested loop and sort-merge algorithms, and showed that with adequate main memory, especially when long-lived tuples are present, our algorithm exhibits uniformly better performance.

$deleteInner(x, s, partitions):$

```
    min ← earliestOverlap(partitions, x[V])
    max ← latestOverlap(partitions, x[V])
    for each i from n to max do
        outer_partition ← outer_partition ∪ {read(r_i)}
        for each tuple x ∈ outer_partition do
            if overlap(x, y[V]) ≠ ∅ then
                outer_partition ← outer_partition − {x}
            delete_page ← delete_page ∪ {{x} ⋈^V outer_partition

    for each i from max to min do
        outer_partition ← read(r_i)
        delete_page ← delete_page ∪ {{x} ⋈^V outer_partition
        for each page q of result_i do
            result_page ← read(q)
            for each tuple d ∈ delete_page do
                if d ∈ result_page then
                    result_page ← result_page - {d}
```

Figure 18: Incremental Evaluation on Inner Relation Delete

- We motivated the importance of incremental evaluation to temporal database management systems and showed how our partition-based approach is easily adapted to incremental evaluation.

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 3.3 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; gross mis-estimation of tuple caching costs may result. It is therefore necessary to also sample the inner relation to more accurately estimate evaluation costs. 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 [JCG+92, SA86]. Our colleagues previously defined incremental semantics for both valid-time databases [McK88, MS91] and transaction time databases [JMRS92]. This work did not address evaluation-level issues, rather they concentrated on the semantics of incremental valid-time and transaction-time operators. In this paper, we presented the underlying implementation of a single valid-time operator, the valid-time natural join; future work will address the implementation of the remaining incremental valid-time operators. With techniques in hand for the incremental evaluation of valid-time databases, we plan to adapt techniques for incremental transaction-time query evaluation resulting in an incremental bitemporal query evaluation system.

Acknowledgements

Support for this work was provided the IBM Corporation through Contract #1124, the National Science Foundation through grant IRI-8902707, the Danish Natural Science Research Council through grant no. 11-9675-1 SE, and Christian and Otilia Brorsons Mindelegat. 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.

[BCL86] J. A. Blakeley, N. Coburn, and P.-A. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In *Proceedings of the Conference on Very Large Databases*, pages 457–466, Kyoto, Japan, August 1986.

[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.

[Han87] E. N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of ACM SIGMOD*, pages 440–453, San Francisco, CA, May 1987.

[Han88] E. N. Hanson. Processing Queries Against Database Procedures: A Performance Analysis. In *Proceedings of ACM SIGMOD*, pages 295–302, Chicago, IL, June 1988.

[Hor86] S. B. Horwitz. Adding Relational Databases to Existing Software Systems: Implicit Relations and a New Relational Query Evaluation Method. Technical Report 674, Computer Science Department, University of Wisconsin, Madison, WI, November 1986.

[HT86] S. B. Horwitz and T. Teitelbaum. Generating Editing Environments Based on Relations and Attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[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, April 1993.

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. 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.

[LM91a] T. Y. Leung and R. Muntz. Stream Processing: Temporal Query Processing and Optimization. Technical report, University of California, Los Angeles, December 1991.

[LM91b] T. Y. Leung and R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. Technical Report CSD-910077, Computer Science Department, University of California, Los Angeles, November 1991.

[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.

[McK88] E. McKenzie. *An Algebraic Language for Query and Update of Temporal Databases.* PhD thesis, Department of Computer Science, University of North Carolina, September 1988.

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

[MS91] E. McKenzie and R. Snodgrass. An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–543, December 1991.

[RK86a] N. Roussopoulos and H. Kang. Preliminary Design of ADMS: A Workstation-Mainframe Integrated Architecture for Database Management Systems. In *Proceeding of the Conference on Very Large Databases*, pages 355–364, Kyoto, Japan, August 1986.

[RK86b] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of ADMS. *IEEE Computer*, 19(12):19–25, December 1986.

[Rou87] N. Roussopoulos. Overview of ADMS: A High Performance Database Management System. Technical report, University of Maryland, 1987.

[Rou91] N. Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[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.

[Sno88] R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Database Systems*, 6(2):157–196, May 1988.

[Sno90] R. 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.

[SRMF87] T. Sellis, N. Roussopoulos, L. Mark, and C. Faloutsos. High Performance Expert Database Systems: Efficient Support for Engineering Environments. Technical report, Department of Computer Science, University of Maryland, April 1987.