Improving Temporal Joins Using Histograms

Inga Sitzmann and Peter J. Stuckey

Department of Computer Science and Software Engineering
The University of Melbourne
Parkville 3052, Australia
(inga, pjs)@cs.mu.oz.au

Abstract. Histograms are used in most commercial database systems to estimate query result sizes and evaluation plan costs. They can also be used to optimize join algorithms. In this paper we consider how to use histograms to improve the join processing in temporal databases. We define histograms for temporal data and a temporal join algorithm that makes use of this histogram information. The join algorithm is a temporal partition-join with dynamic buffer allocation. Histogram information is used to determine partition boundaries that maximize overall buffer usage. We compare the performance of this join algorithm to temporal join evaluation strategies that do not use histograms, such as a partition-based algorithm based on sampling and a partition-join using the Time Index, an index structure for temporal data. The results demonstrate that the temporal partition-join is substantially improved through the incorporation of histogram information, showing significantly better performance than the sampling-based algorithm and achieving equivalent performance to the Time Index join without requiring an index.

1 Introduction

The increasing need to store time-related data has challenged conventional database technologies. Temporal information usually leads to a larger amount of data to be stored and processed. Query evaluation has to deal with different operators and different properties of temporal data. As in relational databases, efficient evaluation of the join operator is crucial for the overall performance of the system.

Several temporal join algorithms have been presented in the literature. Temporal joins are categorised in [GS91] taking the physical organisation and indexing of the data into account. A sort-merge-based algorithm for time-oriented data is presented in [PJ99]. An incremental approach is used to minimize the cost for updating temporal joins in append-only databases. In [RF93], several refinements of the temporal nested-loop join are proposed that minimize comparisons and buffer space required to reduce disk I/O. A partition-based temporal join is presented in [SSJ94]. Sampling is used to approximate the distribution of the data and to compute the partitioning. The *Time Index* is used in [SE96] for a temporal join. The information stored in the *Time Index* is used for a partition-based join. In the paper [LM92], a partition-based join in a multi-processor environment is described.

We suggest extending the use of histograms to temporal joins and present a partition-based temporal join using histogram information. Histograms have become standard in most commercial database systems for estimating result sizes and optimizing evaluation plans. Making use of information already available in the system allows us to improve temporal joins significantly at no additional cost. We have also investigated a histogram-based merge-join for append-only databases [SS00] which improved on a greedy temporal merge-join in some cases and showed more robustness overall.

2 Temporal Data and Joins

Time is assumed to be a sequence of discrete time instances which differ by exactly one minimal time unit (chronon). An entity in a temporal relation consists of a surrogate attribute S, time-varying attributes and time-invariant attributes. For each object in the database, different versions with different time intervals can exist. Surrogate attributes are used in temporal databases to relate different versions of an entity to the entity. Various temporal data models have been presented in the literature [SS88]. We use the time-interval representation where each tuple t is stamped with a time interval $[t.t_s,\ t.t_e]$ indicating the range of time where the tuple is valid.

The time attributes associated with temporal relations can refer to either the *valid time* or the *transaction time*. A classification of temporal joins is given in [GS91]. In this paper, we focus on evaluation of the valid-time time join. The *time* join finds tuples with intersecting time intervals without specifying a join predicate on non-time attributes. The *time-equi* join also requires equality on non-time attributes. Our approach can also be used for *time-equi* joins if joining on the time attribute first leads to more efficient evaluation [SS00].

3 Histograms

Histograms are statistical approximations to data distributions. They are used in database systems to estimate the size of a query and to find efficient access plans. Histograms are created by partitioning the distribution of attribute values into buckets. In each bucket, information about the estimated frequency of bucket values is maintained. To update histograms, samples of the relations are drawn.

Extensive research has been performed on creating and maintaining accurate histograms. Various classes of histograms have been presented. Two well-known classes of histograms are *equi-width* and *equi-depth* histograms. In an equi-width histogram, the widths of all buckets are the same. In equi-depth histograms, the number of tuples with the value represented by the bucket are the same. The width of the buckets varies. Examples for more advanced classes of histograms are *spline-based*, *V-optimal* and *end-biased* histograms. An extensive investigation of histogram classes can be found in [Poo97], [PIHS96].

3.1 Temporal Histograms

To estimate the number of tuples qualifying for a temporal join, we need to know the number of tuples active during a time interval. We distinguish between two kinds of tuples active in a time interval: (i) tuples starting in the interval and (ii) tuples that started in a previous interval but are still active in this interval. We call the latter "long-lived" tuples.

Temporal histograms show the number of tuples that start (resp. are still active) in each histogram bucket. In our experiments we use simple equi-width histograms. The use of more advanced histogram classes leads to higher accuracy for skewed data and should improve the performance of our evaluation methods even further.

Temporal histograms divide the time range of the relation P into m buckets where bucket i denotes time range $[H_P[i], H_P[i+1][$ [1] (assuming $H_P[m+1] =$ end_time is greater than all possible times). A temporal tuple t with start time $t.t_s$ and end time $t.t_e$ is active in histogram bucket i if one of the following conditions holds:

(1) $H_P[i] \leq t.t_s < H_P[i+1]$
(2) $t.t_s < H_P[i] \leq t.t_e$

The two histograms used in the algorithms represent:

1. the number of tuples $START_R[i]$ of R that start during bucket i (active due to the condition (1) above); and
2. the number of tuples $ACT_R[i]$ of R that started in a previous bucket j with $j < i$ but are still active in bucket i (active due to the condition (2) above).

Notice that the two histograms for the same relation have the same bucket boundaries.

Throughout we assume that the histogram heights are normalized to the size of the actual relation. For example, if 2% of the tuples of relation P are sampled to create the histogram, the resulting start and active numbers are multiplied by 50 to give the histograms for the relation P.

Figures 1 (a) and (b) show histograms for the start times for two relations R and S. The histogram for R is equi-width, while that for S is equi-height. For example, 7 tuples in R start in the range [20,40[while 4 tuples in S start in the range [16,24[.

Information about "long-lived" tuples is summarized by giving the number of tuples still active from previous intervals at the start point of the histogram interval as shown in Figures 1 (c) and (d). For example, 4 tuples in R are still active at time 60 and possibly stay active till the end of the interval.[2]

[1] $[a, b[$ denotes the interval $\{x \mid a \leq x < b\}$
[2] Although they might have become inactive during the interval, they will be treated as if they are active during the whole duration of the interval (the worst case).

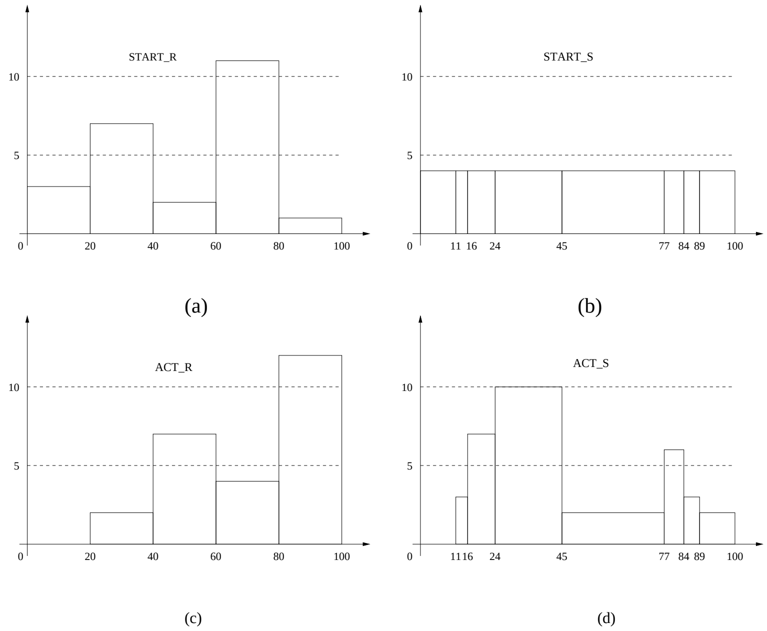


Fig. 1. Start times and active tuple histograms for an (a) equi-width histogram for relation R and (b) (start time) equi-height histogram for relation S

4 Temporal Partition-based Joins

Partition-based joins often have reduced I/O-costs in comparison with other join algorithms, e.g. the nested-loop join. Partition-based joins try to cluster tuples with similar values of the join attribute. The relation is then split into partitions that fit into main memory. During join evaluation, only tuples in corresponding partitions have to be compared.

A partition-based join consists of three phases. The first phase calculates the boundaries of the partitions. In the second phase, the relations are physically partitioned. This involves reading the relations once and writing the tuples back in the order of the partitions they belong to. The third phase finally reads the corresponding partitions of both relations and joins them in main memory.

The performance of a partition-based join strongly depends on the quality of the partitioning. A good partitioning avoids buffer overflow and requires the partitions to be read only once. Common techniques use sampling or an index to approximate the distribution of the attribute values used for calculating the partitioning. The disadvantage is that either additional block accesses for sampling are necessary or the data has to be stored in a particular index structure.

4.1 Partition-based Valid-time Join

The partition-based valid-time join (PBVTJ) was proposed in [SSJ94]. Its partitioning algorithm uses sampling to determine the partition boundaries. Their experiments showed that this join outperforms nested-loop and sort-merge-joins.

The buffer allocation scheme used in the approach of [SSJ94] divides the available buffer into an inner partition area of one block, a tuple cache page of one block for long-lived tuples of the inner relation and an outer partition area (the remaining blocks). The sizes of all three parts are fixed. In what follows, we refer to this buffer organization as "static allocation".

Partition boundaries are then determined by drawing samples from the larger of the two relations to be joined. The samples are used to estimate the distribution of data. Depending on the sample rate, a part of the outer relation buffer is reserved for potential overflow tuples. The actual sampling percentage is determined by estimating the cost for the join process. The optimal total join cost is a trade-off between the join cost and the sampling cost.

Long-lived tuples have to be kept for each partition they overlap to find all possible join partners. To avoid physically duplicating these tuples in each relation, a certain part of the buffer is reserved for them, and the partitions must be visited in time order. After each join step, only tuples not valid in the next partition are removed. Long-lived tuples are held in buffer. Long-lived tuples of the outer relation are kept in the outer relation buffer, long-lived tuples of the inner relation are stored in the tuple cache page and flushed to disk if necessary.

4.2 The *Time-Index* join

The join algorithm of [RF93] makes use of the *Time Index* to compute a perfect partitioning, where each partition is guaranteed to fit into the allocated buffer space. The Time Index is an index structure for temporal data that stores pointers to tuples in order of indexing points. An indexing point is created when a version of an object starts or ends at this time point. Indexing points can be stored in a B^+-tree like structure. Tuple pointers can be split into three buckets for an indexing point: (i) tuples that started at that point, (ii) tuples ending at that point, and (iii) tuples still active from a previous indexing point. In a index node, only the first entry contains pointers to all active tuples. The other entries contain only incremental changes. Together with the pointers, the number of tuples in each bucket is maintained. For the join, each relation is allocated a fixed amount of buffer space (by default, half) and the partitioning process scans the time index to find the largest starting time for the next partition.

The paper [RF93] also considers joining append-only relations, and in this case advocates a partition-join with dynamic buffer allocation. The partitioning process estimates the arrival rate of new tuples for each relation using the *Time Index* and allocates buffer space to each relation based on this estimate, and then follows the partitioning process above. It is not clear why the authors restrict dynamic partitioning to append-only relations.

5 Partition-Join Using Histograms

Although the partition-based natural join has proved ([SSJ94]) to be more I/O cost efficient than the nested-loop and sort-merge-join, several aspects can still be significantly improved. The I/O accesses for sampling can be completely avoided if information about the data distribution, such as a histogram, is already available in the system. For the Time-Index join, using an index not only involves additional block accesses during the join process but also requires building and maintaining an index on the join attribute in the system. We present a partition-based temporal join algorithm that computes an optimal dynamic partitioning using temporal histograms. This algorithm uses dynamic buffer allocation and does not require the data to be stored in any particular way.

5.1 Dynamic buffer allocation

Investigation of the cost of join algorithms has shown that algorithms should try to maximize the number of blocks read from disk for both relations instead of reading one relation in large parts and the other relation pagewise [HR96]. For the temporal histogram partition-join, we use this strategy, which also leads to better buffer utilization. The sizes of the input buffers of both relations vary in size from partition to partition depending on the ratio of active tuples in each relation in the current partition. The sizes of the input buffers are also determined during the partitioning computation step.

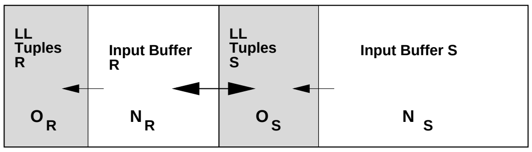


Fig. 2. Storage organization for the temporal partition-join

Figure 2 illustrates this memory organization. The buffer is divided into a smaller part for relation R and a larger part for relation S. Within the input buffers, a certain amount of space (marked grey in the figure) is occupied by long-lived tuples.

The histogram partition-join can also be executed using a static buffer allocation as described in Section 4.1.

5.2 Computing Partition Boundaries

As overflow of partitions leads to expensive additional disk accesses during the join, the quality of the partition boundaries is the crucial part for the performance of a partition-join. We use temporal histograms to calculate the partitioning.

As defined in Section 3, we assume we have histograms for the start times of tuples as well as for the number of active long-lived tuples. The algorithm scans through the time range of the relations. A partition is formed every time the number of blocks required to store the number of tuples still active and the number of tuples starting during the partition fills the available buffer space.

In the following algorithm, $Act_P(t)$ is the estimated number of tuples active for each relation P at time t. We can calculate $Act_P(t)$ from the histogram as

$$Act_P(t) = ACT_P[j] + START_P[j], \text{where } j = \max\{k \mid H_P[k] \leq t\}$$

We calculate $Start_P(t)$, the estimated number of tuples starting at time t, as

$$Start_P(t) = \begin{cases} START_P[j], & \exists j \; t = H_P[j] \\ \infty, & t = \mathsf{end\_time} \\ 0, & \text{otherwise} \end{cases}$$

Effectively, we treat each tuple in a histogram bucket i as starting at the start of that bucket ($H_P[i]$) and existing over the entire bucket time range.

B_{total} is the total buffer space (in blocks) available for the partitions of the two relations. The expression $block(n)$ is the number of blocks required to store n tuples. $PBounds[i]$ is the start time for the i^{th} partition. $T_R[i]$ and $T_S[i]$ denote the number of tuples in partition i calculated so far. $S_R[i]$ and $S_S[i]$ are the buffer sizes (in blocks) reserved for R and S in partition i. The algorithm makes no assumptions on the form (e.g. equi-width or equi-height) of the histograms, or that the histograms for R and S share common boundaries.

The time counter t is initialized to the earliest start time, and the number of partitions is set to zero. Until the whole of the time range of the relations has been covered, the algorithm tests if it can include the next time point in the current partition. If the tuples that start at this time point could be added to the partition without overflowing the buffer, the partition is extended to include this time point. Otherwise, the current partition is finished and a new partition is started. For the new partition, the tuple counter is again initialized with the number of long-lived tuples at the start of the partition.

```
calculate_partitioning(R,S)
  t := start_time; np:= 0;
  while t < end_time
    np++; PBounds[np] := t;
    T_R[np] := Act_R(t); T_S[np] := Act_S(t);
    t := t + 1;
    while (Start_R(t) = 0 and Start_S(t) = 0) or
              (block(T_R[np] + Start_R(t))) + (block(T_S[np] + Start_S(t))) < B_Total)
      T_R[np] = T_R[np] + Start_R(t); T_S[np] = T_S[np] + Start_S(t);
      t := t + 1;
    S_R[np] = block(T_R[np]); S_S[np] = block(T_S[np]);
  return PBounds;
```

This algorithm determines the partition boundaries and buffer sizes for each partition taking into account the space for both long-lived tuples and newly starting tuples. The resulting partition boundaries are used to physically partition the two relations using Grace partitioning [KTM83]. Note that indirectly partition boundaries will always be histogram boundaries.

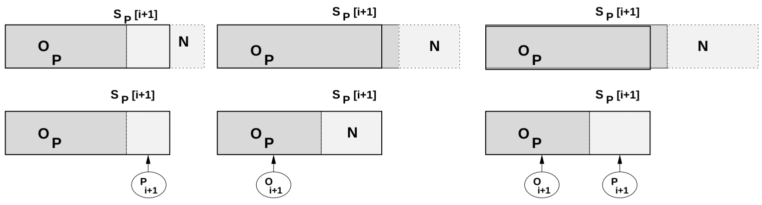
5.3 Joining the Partitions

The main join algorithms reads each partition and joins it in main memory using any kind of suitable join. Because long-lived tuples occur in more than one partition, the join is executed in three steps to avoid joining long-lived tuples again. For each partition, the tuples for each relation are read and joined with the other relation's active long-lived tuples (O_R, O_S), and with each other. After the join, the long-lived tuples for the next partition are determined and kept (in fact, with care, this can be folded into the join).

join_partitions(R,S)
  $O_R := \emptyset$; $O_S := \emptyset$;
  for $i = 1$ **to** np
    $N_R :=$ ReadPartition(R,i); $N_S :=$ ReadPartition(S,i);
    Join(N_R,O_S); Join(N_R,N_S); Join(O_R,N_S);
    $O_R :=$ KeepLonglivedTuples($N_R \cup O_R$);
    $O_S :=$ KeepLonglivedTuples($N_S \cup O_S$);

5.4 Overflow treatment

As histograms are only an estimation of the distribution of data, partition boundaries can be miscalculated and this may cause the partition to overflow. Buffers



(a) (b) (c)

Fig. 3. Overflow treatment for the partition-join

are internally separated into blocks for long-lived tuples and new tuples. When we are about to read in a next partition $i+1$ for relation P consisting of N blocks, we have $S_P[i+1]$ blocks of allocated buffer space and $block(|O_P|)$ blocks already holding long-lived tuples for P. If $block(|O_P|) < S_P[i+1] < N + block(|O_P|)$ then we cannot fit all the long-lived tuples and new tuples in the available buffer space. In this case $S_P[i+1] - block(|O_P|)$ blocks are used for holding new tuples of partition $i+1$. During the join we read partition $i+1$ in chunks of $S_P[i+1] - block(|O_P|)$ blocks (Figure 3 (a)).

If $block(|O_P|) > S_P[i+1]$ and $N \leq S_P[i+1]/2$ the long-lived tuples by themselves do not fit in the allocated buffer space. In this case we keep $S_P[i+1] - N$ blocks of long-lived tuples in the buffer and save the remaining tuples to a separate overflow file, similar to the tuple cache in the static buffer allocation.

The entire partition of new tuples is read in, and during the join we reread the flushed long-lived tuples (Figure 3 (b)).

If $block(|O_P|) > S_P[i+1]$ and $N > S_P[i+1]/2$ then conceivably too many long-lived tuples would be flushed to disk, so we allocate half of the available buffer space ($S_P[i+1]/2$ blocks) to long-lived tuples and half to new tuples, and save excess long-lived tuples to disk before performing the join (Figure 3 (c)).

6 Experimental Evaluation

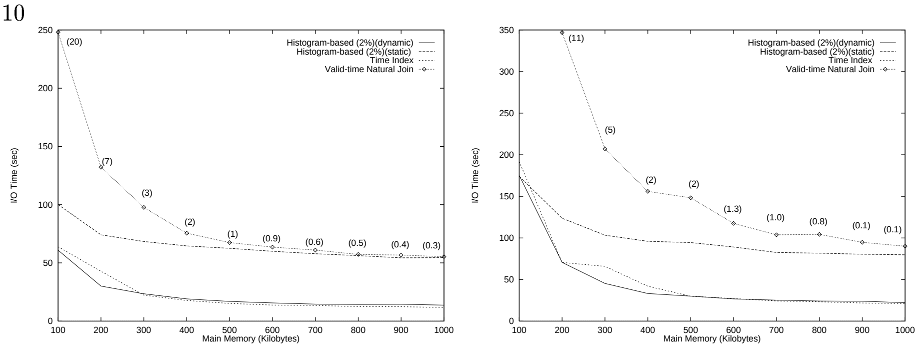
We created testdata with a uniform and a normal distribution of its active tuples. The size of the uniformly distributed relation is 50,000 tuples. The normally distributed relation contains 99,000 tuples. We used a blocksize of 4 Kbytes in the experiments. The size of a tuple is 128 bytes. To create histograms, 2 per cent of the tuples were used as samples (by default). We used total histograms (i.e. histograms with width 1) in the experiments (by default). More details can be found in [SS00].

We assume a random access every time a sequence of blocks is written or read from file. Each of these operations involves one random access followed by the number of sequential accesses. The average disk seek time used is $TS = 10$ms, while the transfer time is $TT = 1$ms/block. The ratio of seek to transfer time is conservative with respect to modern disk drives. All the experiments compare the I/O costs for processing the input, and do not include I/O costs for writing the result of the join. We have not included I/O for handling histograms which is completely overshadowed by the join cost [SS00].

Sensitivity to main memory buffer size: We first compare the performance of the histogram partition-join to the valid-time natural join and the Time Index. The Time Index join has been implemented using dynamic buffer allocation. We varied the size of the available buffer space from 100 Kbytes to 1000 Kbytes. We also measured the effect of using the static buffer allocation used by the valid-time natural join but computed the partitions using histograms. Figure 4 shows the input I/O times for the three algorithms for both distributions. The graph for the valid-time natural join is labeled with the percentage of tuples used for samples. Note that the costs for the valid-time natural join and histogram join do not include the I/O costs for sampling or reading histograms. The sampling time can be quite considerable. The costs for the Time-Index join include the index access costs during the partitioning process.

Both experiments show that the histogram partition-join using dynamic buffer allocation improves significantly on the valid-time natural join. Compared with the Time-Index join, the histogram partition-join demonstrates better performance for small buffer sizes and performance is approximately the same for large buffer sizes.

Static buffer allocation decreases performance significantly. I/O costs for the valid-time natural join are higher because of a less precise partitioning and a generally larger number of partitions due to the part of the buffer reserved for overflow. The Time-Index joins computed perfect partitions in most experiments



(a) Normal distribution (b) Uniform distribution

Fig. 4. Sensitivity to buffer size

and avoided overflow almost completely, but the positive effect is reduced by additional block accesses to the Time-Index during the partitioning process. We also compared the partition-join with histograms of higher accuracy ($>$ 40 per cent sample rate) to the Time-Index join. Using more accurate histograms, the histogram partition-join consistently improved on the Time-Index join by about 10 per cent.

Necessary number of samples: We also investigated the effect of histogram accuracy on the performance. Table 1 shows the I/O costs for sample rates ranging from 0.01 per cent to 100 per cent. The buffer size used in both experiments is 500 Kbytes.

Sample Rate (in %)	Uniform Distribution				Normal Distribution			
	HistoJ (Dynamic)	HistoJ (Static)	Time Index	VTNJ	HistoJ (Dynamic)	HistoJ (Static)	Time Index	VTNJ
0.01	85.48	244.70	15.21	63.53	62.04	301.26	29.94	117.47
0.10	30.18	66.60	15.21	63.53	33.14	96.18	29.94	117.47
1.00	16.93	62.99	15.21	63.53	30.04	91.29	29.94	117.47
10.00	16.79	59.87	15.21	63.53	30.09	89.80	29.94	117.47
100.00	13.97	55.75	15.21	63.53	26.24	83.22	29.94	117.47

Table 1. Sensitivity to histogram accuracy

The results show that histograms based on more than 0.1 per cent of tuples are still accurate enough to avoid significant overflow. Our experiments also showed that for the uniform distribution a sample rate of less than 40 per cent, for the normal distribution of less than 15 per cent, is enough to provide accurate enough histogram information to compute a perfect partitioning and achieve better performance than the Time-Index join. As the maintenance cost of histograms is extremely low compared to maintaining an index, this demonstrates that the histogram join offers better performance at a lower overall cost than the Time-Index join.

Effect of long-lived tuples: In a third experiment, we investigated the effect of a high percentage of long-lived tuples on the performance of the algorithms. Our experiments showed that a higher percentage of long-lived tuples has only little effect on the histogram partition-join. For details refer to [SS00].

Effect of histogram width: In the experiments, we used total histograms, i.e. histograms of width 1 chronon. We also examined the effect of histograms with wider buckets. Our experiments showed that histograms with greater width still provide accurate enough information to calculate a good partitioning. It appears that as long as histograms bars are not larger than partitions, they still provide enough useful information for a good partitioning.

Summary: Overall, our experiments showed that the histogram-based partition-join improves significantly on existing temporal partition-joins by demonstrating better performance than the valid-time natural join. Without requiring an index and thus saving maintainance cost in the system, the histogram-based partition-join shows better performance than the Time-Index join given the accuracy of the histogram information is above a certain threshold and similar performance otherwise.

Acknowledgements We thank Kotagiri Ramamohanarao for his inspiration and support.

References

[GS91] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proc. 7th ICDE*, pages 336–344, April 1991.

[HR96] E.P. Harris and K. Ramamohanarao. Join Algorithm Costs Revisited. *The VLDB Journal*, 5(1):64–84, 1996.

[KTM83] M. Kitsuregawa, H. Tanaka, and T. Motoka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[LM92] T.Y. Leung and R. Muntz. Temporal query processing amd optimization in multiprocessor database machines. In *Proc. of the 18th VLDB*, pages 383–394, 1992.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predi cates. In *Proc. of ACM SIGMOD Conf.*, pages 294–305, June 1996.

[PJ99] Dieter Pfoser and Christian S. Jensen. Incremental join of time-oriented data. In *Proc. of the 11th SSDBM*, pages 232–243, 1999.

[Poo97] Viswanath Poosala. *Histogram-based Estimation Techniques in Database Systems.* PhD thesis, University of Wisconsin – Madison, 1997.

[RF93] S. Rana and F. Fotouhi. Efficient Processing of Time-Joins in Temporal Data Bases. In *Proc. 3rd Int. Symp. on DB Systems for Advanced Applications*, pages 427–432, April 1993.

[SE96] D. Son and R. Elmasri. Efficient Temporal Join Processing Using Time Index. In *Proc. of the 8th SSDBM*, pages 252–261, June 1996.

[SS88] A. Segev and A. Shoshani. The Representation of a Temporal Data Model in the Relational Environment. In M. Rafanelli, J.C. Klensin, and P. Svensson, editors, *LCNS*, volume 339, pages 39–61. Springer-Verlag, 1988.

[SS00] Inga Sitzmann and Peter Stuckey. Histogram-based Temporal Joins Compared to the Time-Index Join. Technical Report ????, Department of Computer Science and Software Engineering, The University of Melbourne, 2000.

[SSJ94] M.D. Soo, R.T. Snodgrass, and C.S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proc. of the 10th ICDE*, 1994.