Parallel Temporal Joins (Kurzbeitrag)

Thomas Zurek

Department of Computer Science
University of Edinburgh
Scotland / UK
Email: tz@dcs.ed.ac.uk

Abstract: We present a framework for parallel temporal joins. The temporal join is a key operator for temporal processing. Efficient implementations are required in order to make temporal database features attractive and applicable for the many applications that are amenable.

We focus on the temporal intersection as the supertype of temporal joins. In contrast to traditional equi-joins, parallel temporal join processing suffers from tuples being replicated between data fragments. This causes a significant overhead.

A basic parallel temporal join strategy – derived from traditional approaches – is refined by two optimisations. The quantitative impacts of the optimisations are evaluated. It is shown that both optimisations together decrease the basic costs significantly.

Keywords: temporal join, temporal databases, parallel databases, data partitioning.

1 Introduction

Recent years have seen increasing research efforts on temporal databases. Temporal data models, temporal query languages and temporal index structures have been the focus of a lot of papers [14]. Only a few proposals have come up discussing algorithms for temporal operations although it is often cited that temporal-specific algorithms are required for performance reasons.

The temporal join is one of the key temporal relational operators. Intuitively, it is used whenever we want to retrieve data that is valid at the same time. Efficient implementations are required in order to make temporal database features attractive and applicable for the many applications that are amenable to temporal data processing. The performance of temporal join processing, however, suffers from the higher size of temporal relations because tuples are *logically* deleted rather than *physically* removed, and from the high selectivity factor of temporal join conditions.

Various authors argue that the semantics of time can be used for optimisations. A popular example is the 'append-only' characteristic of transaction time applications: when new tuples are inserted they are appended to the collection of existing ones. This results in a natural sort order on the transaction time attribute. The sort order itself can then be exploited in several ways.
Parallelism is another possibility of improving temporal processing performance. It has already been successfully incorporated in traditional database technology and helped to overcome certain performance deficits. However, it has been widely neglected in the context of temporal databases. To our knowledge, there has been only one paper that discusses parallel temporal issues [9]. Its optimisations, however, are bound to special cases of temporal joins and there is no quantitative evaluation.
In this paper, we want to overcome some of these deficits and present a framework for parallel temporal join processing. We thereby concentrate on features that are temporal-specific; considerations on data skew or optimisations based on non-temporal parts of the join condition are left out as these have been discussed already by other papers and in more detail. Section 2 discusses types of temporal joins and sequential algorithms that were proposed in the past. Section 3 shows a simple example that should make the reader aware of the temporal-specific problem of parallel join processing. In section 4, two optimisations for parallel temporal joins are presented. Section 5 evaluates these results quantitatively. Finally, the paper is summarised and concluded in section 6.

2 Temporal Joins

2.1 Types of Temporal Joins

Temporal joins combine (at least) two temporal relations using a temporal join condition over the two timestamps. The latter are usually represented as intervals. Temporal join conditions therefore consist of expressions that define relationships between timestamps, i.e. time intervals. Allen identified seven possible relationships[1] between two intervals [1].
An interval $[t_s, t_e]$ consists of a start point t_s and an end point t_e and includes all time points in between and including these. We adopt the most frequently used temporal data model in which each tuple holds an interval timestamp [7].
The literature has mainly concentrated on a 'supertype' of the joins that arise from Allen's relationships, namely the temporal *intersection* that includes the relationships *equals, overlaps, during, starts* and *finishes*. We will focus on intersection joins as all the other temporal joins can be considered as intersection joins with additional constraints.

[1]Actually there are 13 if one takes the six possible reversed relationships into account.

Furthermore we can draw a clear line between the optimisations that are possible for the temporal intersection join and those that are specific to the other temporal joins.

2.2 Join Algorithms

The four principal algorithmic join techniques are brute force nested-loop, sort-merge, hash-joins and index based joins. In general, most join conditions involve an equality predicate. Such joins are called *equi-joins* and a lot of effort has been spent on algorithms that exploit the low selectivity factor imposed by the equality predicate.
Basic nested-loops for equi-joins usually perform badly because they test every tuple combination given by the cartesian product of the participating relations. Of these combinations only few qualify for the result. Other algorithms avoid this by partitioning the data prior to the joining stage. The sort-merge approach is based on implicit partitioning given by a sort order on a join attribute. The hash join approach requires explicit partitioning prior to its joining stage whereas index based joins use precomputed partitioning [11].
In the literature, several algorithms based on these approaches have also been discussed in the context of temporal joining. Sort orderings on the temporal attributes – these are either achieved through explicit sorting or implied by the 'append-only' characteristic of transaction-time relations – allow various optimisations. Discussions can be found in [3], [8], [4], [12].
In the case of equi-joins, explicit partitioning is the basis for very efficient joining. Applying those techniques to temporal intersection of interval data, however, has the problem that intervals cannot be reduced to a discrete value that allows the grouping of the tuples whose timestamps intersect in one single partition. One way to get around this problem is to group tuples by either one of their temporal interval boundaries (start or end point) or a combination of these (e.g. in [10]). Tuples must then be either replicated and put into those partition fragments that hold other tuples that possibly join, i.e. have intersecting timestamps (as in [9] and [13]) or each partition fragment has to be joined with various others that hold possibly joining tuples (as in [10]). The algorithm proposed in [13] processes fragments sequentially and keeps tuples that have to be present in the following fragment in a cache. This makes it an inherently sequential algorithm.
Replication cannot be avoided in the case of parallel temporal join processing. Such an approach is discussed in [9]. The so called *asymmetry* property of relations – that appears e.g. in contain- or overlap-joins – can be used for reducing the number of tuples that have to be replicated. Asymmetry, however, does not occur in the more general intersection join. Therefore, most optimisations that are suggested by Leung and Muntz cannot be applied in the case of the intersection join.

3 An Example

Before digging deeper into the parallel temporal join processing we want to illustrate the problems by an example. For this purpose, the two simple temporal relations of figure 1 are used: one holds the names of members of staff of some university department and the dates when they began and finished to work in that department. Similarly, there is a table holding the names of students and the dates when they began and finished to study in that department.

Name	Start	End
Alex	3	8
Fred	4	9
Kim	2	10
Mary	6	10
Rob	1	8
Vic	1	10

(a) Relation **Staff**

Name	Start	End
Cris	7	9
Fred	1	4
Mary	2	5
Olga	1	3
Sam	7	10
Tim	1	4

(b) Relation **Students**

Figure 1: An example of two temporal relations.

Assuming that the names are unique we can get all persons who studied and worked in the department by computing an equi-join *Staff* $\bowtie_C$ *Students* with the join condition $C \equiv$ *Staff.Name = Students.Name*. To compute this join in parallel we can partition the two tables by using the values of the name attributes. Figure 2 shows an example for partitioning the tables into three fragments respectively and thus into three smaller and independent joins. Please note that the partitioning process produced *disjoint* fragments respectively. Each of the three joins can be processed on different nodes in parallel.

Staff			Students		
Alex	3	8	Cris	7	9
Fred	4	9	Fred	1	4
Kim	2	10			

(N1) Names beginning with A – K

Staff			Students		
Mary	6	10	Mary	2	5
Rob	1	8	Olga	1	3

(N2) Names beginning with L – O

Staff			Students		
Vic	1	10	Sam	7	10
			Tim	1	4

(N3) Names beginning with N – Z

Figure 2: Example of processing an equi-join in parallel.

If we want to relate staff members and students who worked and studied at the same time in the department then we need a temporal intersection join between the two relations. The join condition is $C \equiv$ *TIMESTAMP(Staff) intersects TIMESTAMP(Students)* in this case. Similarly to the equi-join above, the temporal join can be processed in parallel. This time, however, the tables have to be partitioned over the interval timestamps. Figure 3 shows an example of partitioning the tables into three fragments respectively. Please note that the fragments are *not disjoint* in this case and tuples are replicated. This causes an overhead not only because of the effort spent on the replication itself but also because of the additional work imposed on the joining of the fragments.

Staff			Students		
Alex	3	8	Tim	1	4
Kim	2	10	Fred	1	4
Rob	1	8	Mary	2	5
Vic	1	10	Olga	1	3

(N1) Timestamps intersecting with [1,3]

Staff			Students		
Alex	3	8	Tim	1	4
Kim	2	10	Fred	1	4
Fred	4	9	Mary	2	5
Rob	1	8			
Vic	1	10			

(N2) Timestamps intersecting with [4,5]

Staff			Students		
Alex	3	8	Cris	7	9
Kim	2	10	Sam	7	10
Fred	4	9			
Rob	1	8			
Mary	6	10			
Vic	1	10			

(N3) Timestamps intersecting with [6,10]

Figure 3: Example of processing a temporal join in parallel

4 Parallel Temporal Joins

4.1 Replication & Consequences

The algebraic notion behind parallelising joins is the following[2]:

$$R \bowtie S \;=\; R_1 \bowtie S_1 \cup \cdots \cup R_m \bowtie S_m \tag{1}$$

where the R_k and S_k are referred to as *fragments* of R and S respectively. There are several ways for creating the R_k and S_k: section 3 showed an example of *range-partitioning*, i.e. the partition attribute's domain is divided into several ranges and a tuple is placed into the fragment that corresponds to the range that its partition attribute's value falls in. In the case of an atomic attribute domain, this is just a special type of *hash-partitioning* where a hash function is used to assign a tuple to a fragment. This naturally leads to disjoint fragments as there is only one hash value for each attribute value.

Hash partitioning, however, does not work for interval data: assume that there was a hash function f such that intersecting intervals are assigned to the same fragment with at least two different fragments to be created. Let $[x_s, x_e]$ and $[y_s, y_e]$ be two non-intersecting intervals, i.e. $x_s \leq x_e < y_s \leq y_e$, assigned to two different fragments i and

[2]We omit the reference to a join condition in the remainder of the paper and use $\bowtie$ instead of $\bowtie_C$.

j by f, i.e. $f(x_s, x_e) = i \neq j = f(y_s, y_e)$. Now consider the interval $[x_e, y_s]$ which should fall into fragment i because it intersects with $[x_s, x_e]$ but also into fragment j because it intersects with $[y_s, y_e]$. Thus f would have to assign two different values i and j for $[x_e, y_s]$ which contradicts its notion of being a function. Thus, in general, it is impossible to create disjoint fragments for interval data and it is necessary to replicate tuples. In sections 4.2 and 4.3 two measures are presented to reduce the overhead imposed by tuple replication.

4.2 Optimisation 1

Looking at figure 3 we observe the following: tuples (Alex,3,8) of *Staff* and (Tim,1,4) of *Students* are in the first and the second fragments respectively. Thus they are joined twice: in the first partial join at processing node N1 and in the second one at node N2. This not only produces duplicates in the result but is unnecessary. Naturally, we can avoid the duplicate by checking if they are to be joined in one of the preceding partial joins. This is the case if both have the starting point of their interval timestamp in one of the preceding partition ranges. This fact, however, can also be exploited by avoiding them being brought together more than once.

For that purpose, we can divide a fragment R_k into two subfragments R'_k and R''_k. R'_k holds the tuples whose timestamp startpoint falls into the k-th range – these are called the *primary tuples* – and R''_k holds tuples that are already in some fragment R_j with $j < k$ – these are called the *replicated tuples*. A partial join then looks like this

$$R_k \bowtie S_k = (R'_k \cup R''_k) \bowtie (S'_k \cup S''_k) = R'_k \bowtie S'_k \cup R'_k \bowtie S''_k \cup R''_k \bowtie S'_k \cup R''_k \bowtie S''_k \quad (2)$$

However, the join $R''_k \bowtie S''_k$ comprises exactly those 'unnecessary' tuple joins that we have identified above. Processing can therefore be restricted to the first three joins in (2).

4.3 Optimisation 2

The second optimisation is based on the following observation: if the remaining three joins in (2) are processed sequentially in one of the orders

$$\text{(a)} \quad R'_k \bowtie S''_k \,,\; R'_k \bowtie S'_k \,,\; R''_k \bowtie S'_k \quad \text{or} \quad \text{(b)} \quad R''_k \bowtie S'_k \,,\; R'_k \bowtie S'_k \,,\; R'_k \bowtie S''_k$$

and R'_k and S'_k fit, respectively, into the local main memory of the processing node then we can avoid unnecessary accesses to secondary storage. We can, however, enforce this situation because the sizes of the R'_k and S'_k are much easier to predict and to control than those of the R''_k and S''_k (and consequently those of R_k and S_k): each tuple of R and S appears exactly in one R'_k and S'_k but might appear several times in an R''_k or an S''_k respectively. The assignment of a tuple to a certain R'_k (S'_k resp.) only depends on the value of the startpoint of the timestamp. This, however, is nothing else than partitioning atomic values as in the equi-join example in section 3. In the simplest case, the number of fragments m can be chosen high enough to fit the R'_k and S'_k into main memory. More sophisticated techniques might be necessary if the startpoint values are heavily skewed, see e.g. [5] or [2].

One might argue that optimal join ordering and avoiding I/O accesses by optimally using main memory is no special feature of temporal join processing. However, there are two significant issues about this second optimisation:

- A partial join $R_k \bowtie S_k$ is decomposed into three subjoins, one of them being $R'_k \bowtie S'_k$. R'_k and S'_k have predictable sizes as each tuple of R and S appears in only one of these fragments respectively. In contrast, the sizes of R''_k and S''_k, and therefore also of R_k and S_k, are rather difficult to predict because the rate of tuple replication is difficult to estimate. Furthermore, the negative effects of data skew are higher than for R'_k and S'_k: it is not only the data skew on the timestamp start point values, but also skew on the timestamp interval lengths, that influence the sizes of R''_k and S''_k.

- The optimisation is based on the patterns (a) and (b) that occur *regularly*, namely in each of the local joins of the parallel temporal join execution. It is therefore an *integral* part of the algorithm rather than a feature that an optimiser might exploit whenever a qualifying situation is detected.

5 Evaluation

We created a performance model for parallel temporal joins similar to the one presented in [6] for parallel hash joins. Due to the space restrictions for this paper we focus on the basics of the evaluation. The interested reader can find more details in [15].

On a uniform workload[3], three different joins strategies were tested: *Join A*: traditional approach as in (1), *Join B*: join A plus optimisation 1, *Join C*: join A plus optimisations 1 and 2. We compared the performance of the three joins varying the number N of processing nodes. Figure 4 shows the result. As it can be expected join C performs better than join B which itself is better than join A. Interesting facts, however, are the quantitative effects of the optimisations: Optimisation 1 (join A $\rightarrow$ join B) improves performance by between 20% ($N = 10$) and 65% ($N = 50$); Optimisation 2 (join B $\rightarrow$ join C) decreases costs by 90%; Optimisations 1 and 2 (join A $\rightarrow$ join C) give a composite improvement of around 95%. The experiments showed that the costs are dominated by the costs for the partial joins; partitioning costs and therefore communication costs

[3]This enables us to concentrate purely on the effects of the optimisations and avoids any implications given by the delicate choice of a partition for non-uniform data.

can almost be neglected. The costs of joins A and B mainly comprise I/O costs whereas join C's costs consist of CPU and memory access costs. Increasing N implies a higher total I/O bandwidth, more memory and more CPU power. This explains the ideal speedup in figure 4.

Figure 5 shows the split of costs for join C. Graphs for the other two joins look the same; only the respective cost values on the vertical axis are higher but the ratio between the partial cost components is the same. The *replication join costs* are the costs for performing the joins that are due to tuple replication, i.e. the joins involving R''_k and S''_k. The *primary join costs* are the costs for performing the join $R'_k \bowtie S'_k$.

The overhead costs imposed by tuple replication have a share of 65% to 75% of the total costs. This suggests that any optimisations that reduce tuple replication should translate nicely into a total cost reduction.

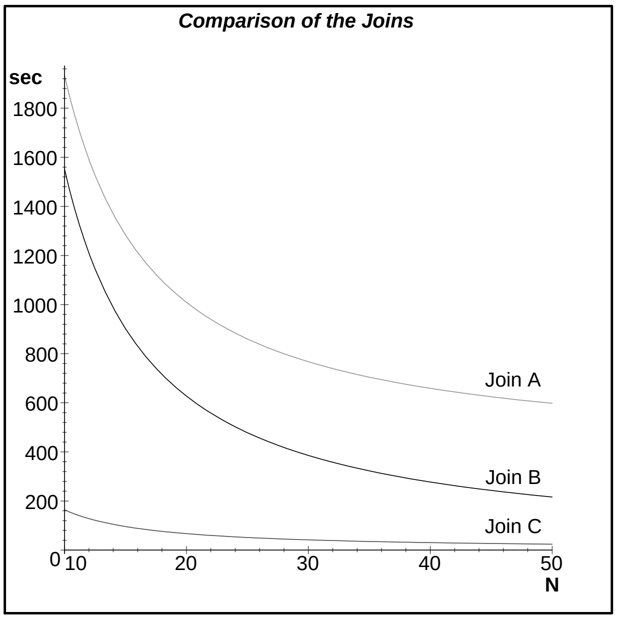

Figure 4: Performances of the three join strategies

6 Summary

In this paper, we discussed the parallel processing of temporal joins. We focused on temporal intersection as other types of temporal joins can be considered as special cases to temporal intersection.

Parallelising temporal joins is not trivial. The significant difference with respect to traditional equi-joins is based on the fact that timestamps are usually represented as intervals. The intersection conditions consist of non-equality predicates. Data partitioning

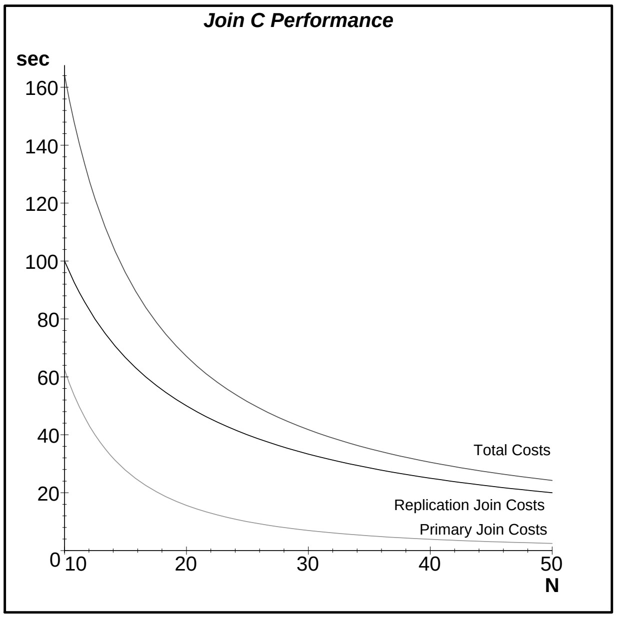

Figure 5: The components of the join C costs

over time intervals is therefore not straightforward: tuples have to be replicated and to be put into several fragments of a range-partition of the temporal relations. This causes a considerable overhead.

We showed that this overhead can be reduced significantly if one divides a data fragment into *primary* and *replicated* tuples. This allows to avoid that replicated tuples of one relation are joined with replicated tuples of another relation as this is not necessary (optimisation 1). Furthermore this division enables us to choose a certain number m of fragments such that subfragments that contain the primary tuples fit into main memory. This allows to reduce I/O accesses significantly (optimisation 2).

Finally, the effects of the optimisations were evaluated quantitatively. Both optimisations reduced costs by around 95%. A further conclusion was that the overhead caused by tuple replication made up around 70% of the total costs. Research efforts on reduction of tuple replication should therefore translate nicely into performance improvement.

An open issue is the choice of an appropriate partition for the temporal relations. This choice is very delicate as it has to cope with data skew and also determines the rate of replication. We plan to investigate this problem in the near future.

References

[1] J. Allen. Maintaining Knowledge about Temporal Intervals. *CACM*, 26(11):832–

843, Nov. 1983.

[2] D. DeWitt, J. Naughton, D. Schneider, and S. Seshradi. Practical Skew Handling in Parallel Joins. In *Proc. 18th VLDB*, pp. 27–40, Aug. 1992.

[3] H. Gunadhi and A. Segev. A Framework for Query Optimization in Temporal Databases. In *Proc. 5th Int. Conf. on Stat. and Scient. Database Management*, pp. 131–147, Apr. 1990.

[4] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proc. 7th Int. Conf. on Data Eng.*, pp. 336–344, Apr. 1991.

[5] K. Hua and C. Lee. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. In *Proc. 17th VLDB*, Sept. 1991.

[6] K. Hua, C. Lee, and J.-K. Peir. Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing. In *Proc. 1st Int. Conf. on Parallel Distr. Inf. Syst.*, pp. 262–270, Dec. 1991.

[7] C. Jensen, M. Soo, and R. Snodgrass. Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7):513–547, 1994.

[8] T. Leung and R. Muntz. Query Processing for Temporal Databases. In *Proc. 6th Int. Conf. on Data Eng.*, pp. 200–208, Feb. 1990.

[9] T. Leung and R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proc. 18th VLDB*, pp. 383–394, Aug. 1992.

[10] H. Lu, B.-C. Ooi, and K.-L. Tan. On Spatially Partitioned Temporal Join. In *Proc. 20th VLDB*, pp. 546–557, Sept. 1994.

[11] P. Mishra and M. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, pp. 63–113, Mar. 1992.

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

[13] M. Soo, R. Snodgrass, and C. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proc. 10th Int. Conf. on Data Eng.*, pp. 282–292, Feb. 1994.

[14] V. Tsotras and A. Kumar. Temporal Database Bibliography Update. *SIGMOD Record*, 25(1), Mar. 1996.

[15] T. Zurek. Parallel Temporal Nested-Loop Joins. Technical Report ECS-CSG-20-96, Dept. of Computer Science, Edinburgh University, Jan. 1996.