Parallel Temporal Nested-Loop Joins

Technical Report

ECS-CSG-20-96

Thomas Zurek
Department of Computer Science
Edinburgh University
King's Buildings
Edinburgh EH9 3JZ
United Kingdom
Email: tz@dcs.ed.ac.uk

January 1996

Abstract

In this report we present a framework for parallel temporal joins. We focus on temporal intersection as the most general temporal join condition. A basic algorithm is given that consists of a partition and a joining stage. In the partition stage tuples have to be replicated if they intersect with more than one partition range. This causes a significant overhead – around 70% in the case of our modest workload. The joining stage can employ any sequential join technique. The basic algorithm is enhanced through two possible optimisations that reduce the overhead imposed by replication.

The algorithm and its optimisations are evaluated on top of a performance model. We describe the model and give details in the appendix. The evaluation shows that both optimisations together decrease the basic costs significantly. Furthermore we can give an idea of the quantitative impact of replication overhead in parallel temporal join processing. Our (modest) workload caused a share of around 70% of the total costs; higher values can be expected in reality.

Contents

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. 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 operators. Intuitively, it is required whenever we want to retrieve data that is valid at the same time. Performance of temporal join processing suffers from

- the higher size of temporal relations due to the fact that tuples are *logically* deleted rather than *physically* removed,
- the high selectivity factor [Piatetsky-Shapiro and Connell, 1984] 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 [Leung and Muntz, 1992]. Its optimisations, however, are bound to special cases of temporal joins and there is no quantitative evaluation and no considerations on architectural issues.

In this report we want to overcome some of these deficits and present a framework for parallel temporal join processing. Section 2 discusses types of temporal joins and sequential algorithms that were proposed in the past. In section 3, a basic parallel temporal join algorithm is given. This algorithm is improved through two optimisations. Section 5 evaluates these results quantitatively. We modeled the performance of the three algorithms on top of a general-purpose hardware architecture. This makes the results useful for a wide range of parallel environments. Details of the performance model are given in the appendix. Finally, the report is 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 the relationships between timestamps, i.e. the time intervals. [Allen, 1983] identified seven possible relationships[1] between two intervals. These are shown in table 1.

We adopt the following notational conventions: an interval x consists of a start point, denoted by $x.t_s$, and an end point, denoted by $x.t_e$. When speaking of a timestamped tuple r we also refer to the respective interval boundaries by $r.t_s$ and $r.t_e$. Intervals are represented by its start and end point being enclosed in brackets: [or] means that the respective boundary is included whereas (and) mean that the respective boundary is excluded:

$$
\begin{aligned}
[x.t_s, x.t_e] &= \{t : x.t_s \leq t \leq x.t_e\} \\
(x.t_s, x.t_e) &= \{t : x.t_s < t < x.t_e\} \\
[x.t_s, x.t_e) &= \{t : x.t_s \leq t < x.t_e\}
\end{aligned}
$$

Relationship	**Example**	**Condition**
x equals y	`xxxx` `yyyy`	$x.t_s = y.t_s \ \wedge \ x.t_e = y.t_e$
x before y	`xxxx yyyy`	$x.t_e < y.t_e$
x meets y	`xxxxyyyy`	$x.t_e = y.t_s$
x overlaps y	`xxxx` `  yyyy`	$x.t_s < y.t_s \ \wedge \ x.t_e > y.t_s \ \wedge \ x.t_e < y.t_e$
x during y	`  xxx` `yyyyyyy`	$x.t_s > y.t_s \ \wedge \ x.t_e < y.t_e$
x starts y	`xxx` `yyyyyy`	$x.t_s = y.t_s \ \wedge \ x.t_e < y.t_e$
x finishes y	`   xxx` `yyyyyy`	$x.t_s > y.t_s \ \wedge \ x.t_e = y.t_e$

Addtional constraints are: $x.t_s \leq x.t_e \ \wedge \ y.t_s \leq y.t_e$

Table 1: The possible relationships between two intervals [Allen, 1983]

Temporal joins can be classified according to the relationship that its join condition is based on: there are temporal contain-, meet-, overlap-, ... joins. Obviously a temporal join condition may contain any arbitrary combination of these relationships. We can consider the corresponding join to be of any type that is involved. In reality it is more likely that there is only one.

The literature has mainly concentrated on a 'supertype' of the joins that arise from table 1, namely the temporal *intersection* that includes the relationships *equals, overlaps, during, starts*

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

and *finishes*. We will concentrate on intersection joins as all the other temporal joins can be considered as intersection joins with additional constraints. 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.

Usually the tuples that satisfy the join condition are concatenated. In the case of temporal joins this concatenation is not trivial as the value of the timestamp for the resulting tuple has to be defined. This definition depends on the type of the temporal join; assuming temporal intersection the resulting timestamp is defined to be the intersection of the individual timestamps of the participating tuples. For example in the case of the two tuples r and s the resulting timestamp is

$$[\max\{r.t_s, s.t_s\}, \min\{r.t_e, s.t_e\}] \tag{1}$$

2.2 Algorithms for Temporal Joins

The four principal algorithmic join techniques are nested-loop, sort-merge, hash-joins and index based joins. 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 imposed by the equality predicate.

Basic nested-loops for equi-joins usually perform badly due to the lack of any preceding partitioning of the data. The sort-merge approach is based on implicit partitioning given by a sort order on a join attribute. This allows to reduce the number of unnecessary comparisons. The hash join approach requires explicit partitioning prior to its joining stage whereas index based joins use precomputed partitioning [Mishra and Eich, 1992].

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 [Gunadhi and Segev, 1990], [Leung and Muntz, 1990], [Gunadhi and Segev, 1991], [Rana and Fotouhi, 1993].

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 [Lu et al., 1994]). 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 [Leung and Muntz, 1992] and [Soo et al., 1994]) or each partition fragment has to be joined with various others that hold possibly joining tuples (as in [Lu et al., 1994]). The algorithm proposed in [Soo et al., 1994] processes fragments in a sequential order and keeps tuples that have to be present in the following fragment in a cache.

Replication cannot be avoided in the case of parallel temporal join processing. Such an approach is discussed in [Leung and Muntz, 1992]. 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, optimisations that are suggested by Leung and Muntz and that originate in the asymmetry property cannot be applied in the case of the intersection join. Basic nested-loops for equi-joins usually perform badly due to the lack of any preceding partitioning. The sort-merge approach is based on implicit partitioning given by a sort order on a join attribute. This allows to reduce the number of unnecessary comparisons. The hash join approach requires explicit partitioning prior to its joining stage whereas index based joins use precomputed partitioning [Mishra and Eich, 1992].

3 Parallel Temporal Joins

In this section we first describe the general structure of parallel temporal joins. We focus on temporal intersection join. None the less what we state can also be applied to the more specific cases like contain- or overlap-joins that allow more specific optimisations due to their increased selectivity. E.g. the optimisations proposed in [Leung and Muntz, 1992] can be considered as an enhancement of the techniques we propose.

After an introduction of the basic architectural and notational issues in sections 3.1 and 3.2 a basic parallel intersection join is presented in section 3.3. Finally, two essential optimisations of the basic algorithm are discussed in section 3.4.

3.1 The Basic Structure

We assume a hybrid parallel architecture as it was described in [Hua et al., 1991]. This architecture has two levels: the inner or node-level is based on a shared-memory approach, whereas the outer level adopts shared-nothing. Put in another way: it is a shared-nothing combination of SMP nodes. This type of architecture has proved to be the most general one and a recent survey of commercial parallel database systems [Norman and Thanisch, 1995] showed that most parallel database systems were optimised for running on this type of hardware. Figure 1 illustrates the architecture[2].

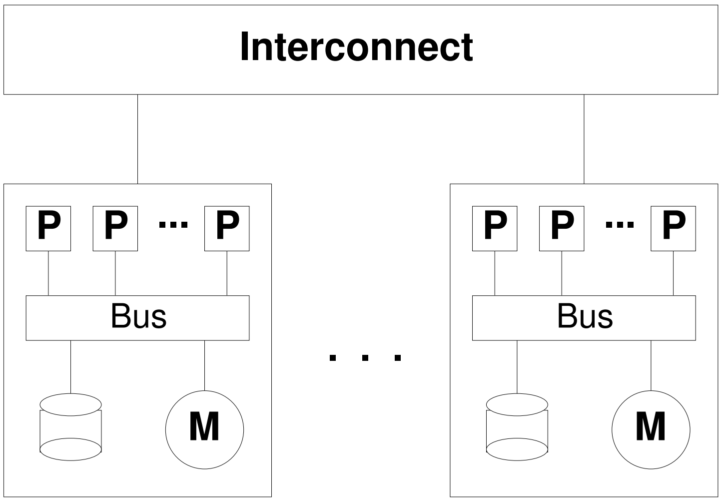

Figure 1: Hybrid parallel architecture

The general stages of a parallel (temporal) join of two (temporal) relations R and S are the following:

1. Partition R into fragments $R_1, \ldots, R_m$;
   partition S into fragments $S_1, \ldots, S_m$.

2. Perform local temporal joins

$$R_1 \bowtie S_1, \ldots, R_m \bowtie S_m$$

[2] For redundancy reasons the disks are usually not only connected to one SMP node but to several. For the purpose of this report we do without this feature.

3. Merge the local results to form a global result.

The general join symbol $\bowtie$ was used here because there is no difference with respect to traditional parallel joins. In the remainder of this report, the temporal aspect is relevant and we will use the symbol $\bowtie^T$. Section 3.3 discusses stages 1 and 2 in more detail because it is there where the differences between a parallel temporal join and a traditional parallel join arise.

It is assumed that the partitions of R and S have to be created dynamically. It is unlikely that R and S will be partitioned over the timestamp attribute using the same partition points. It is even more unlikely that those partitions are colocated[3].

3.2 Preliminaries

Let T be the time span covered by the tuples of R and S. For the purpose of this report we assume T to be an interval over a discrete domain, e.g. an interval of integers $[t_{\min}, \ldots, t_{\max}]$. A temporal m-way partition P of T is a set of $m+1$ partition points

$$\{p_0, p_1, \ldots, p_{m-1}, p_m\}$$

with $p_0 = t_{\min}, p_m = t_{\max} + 1$ and $p_i \in T$ for $i = 0, \ldots, m-1$. P divides T into m partition intervals

$$[p_{i-1}, p_i) \;=\; \{t \in T \mid p_{i-1} \leq t < p_i\}$$

for $i = 1, \ldots, m$. The following function determines the number of the fragment, time range respectively, that a time point $t \in T$ belongs to with respect to partition P

$$\mathit{fragment}_P(t) \;\equiv\; k \;\text{ iff }\; t \in [p_{k-1}, p_k)$$

The parallel hardware has N nodes each with n processors. The nodes are numbered from 1 to N and the processors from 1 to nN such that all processors of node i have numbers from $n\,(i-1)+1$ to $n\,(i-1)+n = ni$. A function $\mathit{node}(k)$ gives the number of the node that processor k belongs to

$$\mathit{node}(k) \;\equiv\; (k \;\mathit{div}\; N) + 1$$

For the moment we assume that the number m of fragments matches the number of processors nN. For convenience we additionally use an N-way partition Q of T. Q is a subset of P and holds every n-th partition point of P, i.e. the partition points that coincide with the points used as node boundaries. This is helpful when describing the algorithm in the next section

$$Q = \{q_0, q_1, \ldots, q_{N-1}, q_N\}$$

with $q_i = p_{ni}$. Similarly there is a function $\mathit{fragment}_Q$ for Q.

3.3 The Algorithm

The algorithm adopts the structure presented in section 3.1. We concentrate on stages 1 and 2 only; stage 3 either leaves the result partitioned for further processing or saves the tuples on disk.

At the start, relations R and S are assumed to be physically partitioned into N fragments that are distributed over the disks of the N nodes.

[3] In the sense of colocated joins as discussed in [Baru et al., 1995].

Stage 1

Range-partitioning for R, i.e. creation of the fragments $R_1, \ldots, R_{nN}$ (remember that $m = nN$) in stage 1 is done in the following way; S is partitioned in a similar way:

(a) Each node reads its fragment of R; then each processor processes the n-th part of this fragment.

(b) Each processor has nN hash buffers – one for each processor of the machine. It hashes its tuples to the buffers in the following way: a tuple with timestamp $[t_s, t_e]$ is put into

  (i) hash buffer $k = \mathit{fragment}_P(t_s)$

  (ii) hash buffers with numbers $(k-1)n+1$ where

$$\mathit{fragment}_Q(t_s) < k \leq \mathit{fragment}_Q(t_e)$$

Remark: Step (i) puts the tuple in the fragment that covers the range in which the timestamp's start point t_s falls; step (ii) puts the tuples in the first fragments on nodes (other than that covered by step (i)) that hold ranges with which $[t_s, t_e]$ intersects.

When a hash buffer is full it is transmitted to the output buffer of the corresponding processor.

(c) A further replication step is performed when tuples (each with some time interval $[t_s, t_e]$) arrive, say at processor k:

if $(k < \mathit{fragment}_P(t_e))$ **then**
  for $l = k+1$ **to** $\min(n \cdot \mathit{node}(k), \mathit{fragment}_P(t_e))$
    send tuple to the output buffer of processor l

Remark: Alternatively, an index structure can be built (for each processor) that refers to those tuples in shared-memory that fall into the respective fragment. In the case of large tuples this is certainly faster than replicating the tuples in main memory. In most cases, in particular for the workload of our experiments (see section 5), the choice of strategy has only little impact on the overall performance. Therefore we only describe the simple copying strategy.

(d) When an output buffer is full then its tuples are flushed to disk.

The significant difference, in comparison to partitioning for a traditional parallel join, is the replication of tuples in steps (b).(ii) and (c). We chose a two-level replication: (b).(ii) replicates the tuples over the interconnect and positions the tuples on all nodes that hold ranges that intersect with the tuples' timestamps; this step can be seen as an *inter-node replication*. Step (c) replicates the tuples within the nodes and sends them to all processors that cope with a range that intersects with the tuples' timestamps. This *intra-node replication* is faster because it can be done via shared-memory rather than via communication over the interconnection network. If this step was incorporated into step (b).(ii) the advantage of fast communication via main memory would be lost.

Stage 2

We now focus on stage 2 of the algorithm. Actually, any sequential temporal join algorithm can be used for performing the local joins $R_1 \bowtie^T S_1, \ldots, R_{nN} \bowtie^T S_{nN}$. We adopt a nested-loop approach for the following reason:

The performance drawback of nested-loop in the case of a traditional equi-join is mainly due to the fact that the algorithm compares the two portions of tuples completely in order to compute the join result. Sort-merge equi-joins can decrease this overlap of the two relations nearly to a minimum as they take advantage of the relations being sorted on a join attribute. Hash-based equi-joins only do the necessary comparisons due to disjoint partitioning [Mishra and Eich, 1992].

In the case of a temporal intersection join, however, the degrees of overlap of sort-merge and hash joins have to be variable and are – as we argue – very close to a complete overlap: R and S are already range-partitioned in our case; so the degree of necessary overlap between the portions R_k and S_k is likely to be high. Therefore the advantages of sort-merge and hash are reduced. Furthermore they would require R_k and S_k to be sorted[4] or hashed beforehand which imposes an overhead and further reduces their advantages.

Stage 2 of the basic algorithm then works in the following way: each processor k copes with 'its' fragments R_k and S_k of R and S respectively. Without loss of generality we assume $|R_k| \leq |S_k|$ in the following. This only implies that – for efficiency reasons – R_k will be the outer and S_k the inner relation in the nested-loop computation of $R_k \bowtie^T S_k$.

We assume that the join condition is a temporal intersection and some boolean expression $C(r, s)$. The latter is supposed to be non-temporal and therefore amenable to the same optimisations that may be applied to non-temporal join evaluation. For performance modelling purposes we later assume that $C(r, s)$ evaluates to *true* so that we can neglect any implications given by this part of the join condition and concentrate on the essential temporal aspects. Stage 2 then looks like this:

```
for each tuple r in R_k do
   { for each tuple s in S_k do
        { if ([r.t_s, r.t_e] intersects [s.t_s, s.t_e]) and no_previous_join(r, s, k) and C(r, s) then
              { time-concatenate r and s
                place result in output buffer X_k
                if (X_k is full) then flush to disk
              }
        }
   }
```

To avoid the situation in which the global result contains duplicates that are a consequence of tuple replication, tuples r and s are only joined if at least one of them appears in no preceding fragment R_l or S_l with $l < k$, i.e. if at least one of their timestamps has its start point in the current range. This is determined by the boolean function *no_previous_join*(r, s, k) that can be defined as

$$\textit{no\_previous\_join}(r, s, k) \equiv (\textit{fragment}_P(r.t_s) = k) \text{ or } (\textit{fragment}_P(s.t_s) = k)$$

[4]In a message-passing environment it is difficult to preserve a sort order on R_k, S_k in the case that relations R, S are already sorted. At least it would slow the communication down and would impose an overhead in this way even though the sorting of R_k and S_k was avoided.

Here is a short formal proof for this intuitively derived restriction:

$$\begin{aligned}
& \textit{no_previous_join}(r, s, k) \\
\Leftrightarrow\ & (\textit{fragment}_P(r.t_s) = k) \text{ or } (\textit{fragment}_P(s.t_s) = k) \\
\Leftrightarrow\ & r.t_s \in [p_{k-1}, p_k) \text{ or } s.t_s \in [p_{k-1}, p_k) \\
\Rightarrow\ & \forall j < k : r.t_s \notin [p_{j-1}, p_j) \text{ or } s.t_s \notin [p_{j-1}, p_j) \\
\Rightarrow\ & \forall j < k : r \notin R_j \text{ or } s \notin S_j \\
\Rightarrow\ & \forall j < k : r \circ s \ \notin\ R_j \bowtie^T S_j
\end{aligned}$$

When two tuples satisfy the join condition they are concatenated. As we mentioned in section 2 the tuple that results from joining two tuples r and s holds the timestamp defined by (1). This process is called the *time-concatenation* of r and s, denoted above as $r \circ s$.

3.4 Optimisations

In this section we want to point to two possible optimisations of the algorithm presented in the previous section.

Optimisation 1

The function *no_previous_join*(r, s, k) was used to avoid replicated tuples being joined unnecessarily and causing duplicates in the result. Nevertheless these tuples are processed by the algorithm. We can avoid this by splitting up the fragments R_k and S_k into two components respectively:

- the set of tuples that have their timestamp start points in the range $[p_{k-1}, p_k)$ – these are called the *primary tuples* and are put into the sets R'_k and S'_k, respectively,
- the set of tuples that fall into the fragment because of replication (i.e. their timestamp start point is not in the range $[p_{k-1}, p_k)$) – these are denoted as the *replicated tuples* and are put into R''_k and S''_k.

Formally, we can describe the benefit of this procedure as follows: stage 2 in section 3.3 does all the processing for computing the local join

$$R_k \bowtie^T S_k \tag{2}$$

but is prevented from putting some tuples to the results through the *no_previous_join*(r, s, k) condition. With

$$\begin{aligned}
R_k &= R'_k \cup R''_k \\
S_k &= S'_k \cup S''_k
\end{aligned}$$

(2) evaluates to

$$R'_k \bowtie^T S'_k \ \cup\ R'_k \bowtie^T S''_k \ \cup\ R''_k \bowtie^T S'_k \ \cup\ R''_k \bowtie^T S''_k \tag{3}$$

which represents four individual joins. The latter one ($R''_k \bowtie^T S''_k$) defines exactly the set of tuples that is excluded from the result by the *no_previous_join*(r, s, k) condition. It is actually unnecessary and can be skipped. Computation can be reduced to the first three joins. Section 5 will show the benefit of this measure.

As a prerequisite for this optimisation we have to keep primary and replicated tuples separated during the partitioning stage 1: each processor had one hash buffer per processor that kept primary and replicated tuples. Now we use two: a primary hash buffer and a replication hash buffer. Similarly, each processor had one output buffer for collecting tuples of its fragment before they were flushed to disk. Now there are primary and replication output buffers. Steps (b) and (c) of stage 1 are modified like this:

(b) Each processor hashes its tuples to the buffers in the following way: a tuple with timestamp $[t_s, t_e]$ is put into

  (i) the primary hash buffer
  $k = \mathit{fragment}_P(t_s)$

  (ii) the replication hash buffers with numbers $(k-1)n+1$ where

$$\mathit{fragment}_Q(t_s) < k \leq \mathit{fragment}_Q(t_e)$$

When a primary hash buffer is full it is transmitted to the primary output buffer of the corresponding processor.
When a replication hash buffer is full it is transmitted to the replication output buffer of the corresponding processor.

(c) A further replication step is performed when tuples (each with some time interval $[t_s, t_e]$) arrive at the replication output of processor (say k):

if $(k < \mathit{fragment}_P(t_e))$ **then**
  for $l = k+1$ **to** $\min(n \cdot \mathit{node}(k), \mathit{fragment}_P(t_e))$
    send tuple to the replication output buffer of processor l

Optimisation 2

A second optimisation is based on two observations:

- The three remaining joins of (3) can be computed in the following orders (amongst others):

$$R'_k \bowtie^T S''_k\ ,\ R'_k \bowtie^T S'_k\ ,\ R''_k \bowtie^T S'_k \tag{4}$$

  or

$$R''_k \bowtie^T S'_k\ ,\ R'_k \bowtie^T S'_k\ ,\ R'_k \bowtie^T S''_k \tag{5}$$

- Until now we assumed that the number m of fragments matches the number nN of processors. Alternatively, we can choose m in a way such that R'_k and/or S'_k are small enough to fit into main memory of the node. This is possible because the sizes of R'_k and S'_k can be nearly *arbitrarily* cut down by increasing m; remember that they only hold tuples that have their timestamp start point in the range $[p_{k-1}, p_k)$. Thus each tuple of R (S respectively) belongs only to one R'_k (S'_k respectively) because the ranges are disjoint. Thus R (S) is partitioned into more and smaller fragments[5] by increasing m.

[5]This statement falls apart only in extreme cases of data skew. For the purpose of the report, however, we assume a uniform distribution of the data.

Assuming that the joins are computed in the order as in (4) or (5) and keeping R'_k and S'_k in main memory we can avoid unnecessary accesses to secondary storage: R'_k is loaded once into main memory for computing the first join in (4) and is then kept for computing the second join, and finally the third join is computed with S'_k in main memory. The procedure works for (5) accordingly.

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:

- The original join (2) is decomposed into three subjoins, one of them being $R'_k \bowtie^T S'_k$. 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 (4) and (5) 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 and not a feature that an optimiser might exploit whenever a qualifying situation is detected.

In section 5 the quantitative benefit will be shown.

4 Performance Model

In this section the performance of the parallel temporal joining techniques described in section 3 is modeled:

- the basic algorithm described in 3.3 (join A),
- the basic algorithm plus optimisation 1 (join B),
- the basic algorithm plus optimisations 1 and 2 (join C).

Section 4.1 describes the basic issues of the model; the following sections described the model for each of the stages:

- the partitionining stage / stage 1 in section 4.2,
- the joining stage / stage 2 of join A in section 4.3,
- the joining stage / stage 2 of join B in section 4.4,
- and the joining stage / stage 2 of join C in section 4.5.

4.1 The Basic Issues

The performance of the algorithms was modeled in a similar way as the parallel hash join in [Hua et al., 1991]. The major issues are:

- The total response time C_{total} of the algorithms depends on the times C_{part}, C_{join} spent in stages 1 and 2. In reality there might be an overlap between these two stages; thus

  $$\max\{C_{part}, C_{join}\} \leq C_{total} \leq C_{part} + C_{join}$$

  In our model, however, we assume that there is no overlap (e.g. enforced through a barrier type synchronisation). Thus we use the upper bound

  $$C_{total} = C_{part} + C_{join}$$

  The stages (a), (b) etc. within stages 1 and 2 are treated accordingly.

- Within each stage the overlap between the I/O, communication, CPU and memory access phases is perfect. This can be nearly achieved by separate I/O and communication processors. Put the other way: in tables 2 – 5 the maximum of each row is taken.

4.2 Partitioning Stage

The partitioning for join C differs from that for A and B – but only from the algorithmic point of view. The performance model can be the same as the amount data being moved by each processor and node is the same. It is only that there are more but smaller data fragments.

The stage – as described on page 8 – comprises disk accesses, communication, CPU time and memory accesses as the major cost factors. We now derive the these costs for each of the substages:

(a) **Loading fragments of R from disk**

This substage does not involve any communication or memory accesses. Only disk accesses and the CPU costs for initiating these accesses have to be modeled.

We assume a uniform distribution of R over the nodes, i.e. equally sized parts of R are stored on the disks of the nodes. Therefore each node has to move

$$\frac{|R|}{N}$$

tuples each of size r to the processors. The disk I/O bandwidth is w_D which leads to

$$\frac{|R|}{N} \cdot \frac{r}{w_D} \tag{6}$$

as the disk access costs whereby a portion of

$$\frac{|R|}{n \cdot N}$$

is loaded by each processor. We assume that tuples are moved blockwise[6] from disk to the processors. Thus a disk I/O has to be initiated only once per block. If b is the size of such a block then

$$\frac{|R|}{n \cdot N} \cdot \frac{r}{b}$$

is the number of blocks to be moved. The time spent on initiating one block movement is

$$\frac{I_{sio}}{\mu}$$

where I_{sio} is the number of microprocessor instructions necessary; μ is the number of instructions per second being performed by the processor. Thus

$$\frac{|R|}{n \cdot N} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu} \tag{7}$$

is the CPU time spent in substage 1 (a).

As mentioned in the previous section, we assume that disk I/O, communication, CPU and memory access phases have a perfect overlap. Therefore it is the maximum of the individual times that is finally relevant. The total time spent on substage 1 (a) is therefore the maximum of (6) and (7):

$$C_{1a} = \max\left\{\frac{|R|}{N} \cdot \frac{r}{w_D}\, ,\ \frac{|R|}{n \cdot N} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}\right\} \tag{8}$$

(b) **Redistribution of the data via the network, including inter-node replication**

Substage 1 (b) describes the distribution of the data between the nodes. It initiates an inter-node replication (via the communication network). Thus it comprises communication and CPU costs.

[6] or – in other terms – pagewise.

We assume that each of the N nodes has to send a portion of

$$\frac{N-1}{N}$$

of its data via the communication network to other nodes. The data comprises not only the primary *but also* the replicated tuples. We use the parameter δ_R to denote the average rate by which one tuple of R is replicated on base of an underlying partition P. Communication via the interconnect is only necessary for inter-node replication, i.e. replication across node boundaries. A node boundary appears every n-th fragment in P (each fragment of Q, respectively). Thus each tuple is replicated over node boundaries at an average rate of

$$\frac{\delta_R}{n}$$

Hence each node sends

$$\frac{N-1}{N} \cdot \frac{|R|}{N} \cdot \frac{\delta_R}{n} \cdot r$$

bytes. As each of the N node sends this amount the total communication costs are

$$\frac{N-1}{N} \cdot |R| \cdot \frac{\delta_R}{n} \cdot \frac{r}{w_C} \tag{9}$$

To initiate the communication each processor has to spend

$$\frac{I_{scomm}}{\mu}$$

seconds per block. Similarly, the CPU costs for computing the expressions $\mathit{fragment}_P(t_s)$, $\mathit{fragment}_Q(t_s)$ and $\mathit{fragment}_Q(t_e)$ in stage 1 (b) are

$$\frac{N-1}{N} \cdot \frac{|R|}{n \cdot N} \cdot \frac{\delta_R}{n} \cdot \frac{3 \cdot I_{exp}}{\mu}$$

The total CPU costs are therefore

$$\frac{N-1}{N} \cdot \frac{|R|}{n \cdot N} \cdot \frac{\delta_R}{n} \cdot \left(\frac{r}{b} \cdot \frac{I_{scomm}}{\mu} + \frac{3 \cdot I_{exp}}{\mu}\right) \tag{10}$$

The total time spent on stage 1 (b) is the maximum of (9) and (10):

$$C_{1b} = \max\left\{\frac{N-1}{N} \cdot |R| \cdot \frac{\delta_R}{n} \cdot \frac{r}{w_C}\,,\ \frac{N-1}{N} \cdot \frac{|R|}{n \cdot N} \cdot \frac{\delta_R}{n} \cdot \left(\frac{r}{b} \cdot \frac{I_{scomm}}{\mu} + \frac{3 \cdot I_{exp}}{\mu}\right)\right\} \tag{11}$$

(c) **Intra-node replication via main memory**

Stage 1 (c) replicates tuples between processors on the same node. Communication can therefore be done via main memory. Originally, there are

$$\frac{|R|}{N}$$

tuples per node. Each tuple is replicated δ_R times on average. If δ_R exceeds n (the number of processors per node) then most tuples are replicated over all processors of a node; otherwise just δ_R times. Writing one tuple to memory creates costs of

$$\frac{r}{w_M}$$

if w_M is the memory bandwidth in bytes per second. Thus the memory access costs for this stage are

$$C_{1c} = \frac{|R|}{N} \cdot \frac{r}{w_M} \cdot \max\{\delta_R, n\} \tag{12}$$

CPU costs for these memory accesses can be neglected as the comprise by far less instructions as computing expressions (I_{exp}) or processing two tuples (I_{proc}). Thus (12) states the total costs for stage 1 (c).

(d) **Writing new fragments of R to disk**

Stage 1 (d) writes the new fragments to disk. The latter contain on average δ_R times more tuples due to replication. The performance equations can be derived similarly to those of stage 1 (a): (6) and (7) only have to be multiplied by δ_R. Thus

$$C_{1d} = \max\left\{\frac{|R|}{N} \cdot \delta_R \cdot \frac{r}{w_D}\ ,\ \frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}\right\} \tag{13}$$

The performance model of stage 1 is summarised in table 2. Relation S has to be partitioned in the same way using the respective parameters $|S|$ and δ_S. It is assumed that the tuple size r is the same for R and S. This is a simplification which keeps the equations simple, especially for the performance modelling of the join stages.

With equations (8), (11), (12) and (13) we can derive the total costs of stage 1:

$$C_1 = C_{1a} + C_{1b} + C_{1c} + C_{1d} \tag{14}$$

The total partition costs C_{part} comprise costs C_1 for relation R plus C_1 for relation S.

Stage	Disk I/O	Communication	CPU	Memory
1 (a)	$\frac{\lvert R\rvert}{N} \cdot \frac{r}{w_D}$		$\frac{\lvert R\rvert}{n \cdot N} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$	
1 (b)		$\frac{N-1}{N} \cdot \lvert R\rvert \cdot \frac{\delta_R}{n} \cdot \frac{r}{w_C}$	$\frac{N-1}{N} \cdot \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{\delta_R}{n} \cdot \left(\frac{r}{b} \cdot \frac{I_{scomm}}{\mu} + \frac{3 \cdot I_{exp}}{\mu}\right)$	
1 (c)				$\frac{\lvert R\rvert}{N} \cdot \frac{r}{w_M} \cdot \max\{\delta_R, n\}$
1 (d)	$\frac{\lvert R\rvert}{N} \cdot \delta_R \cdot \frac{r}{w_D}$		$\frac{\lvert R\rvert}{n \cdot N} \cdot \delta_R \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$	

Table 2: Performance model for the partitioning stage (stage 1, section 3.3)

4.3 Joining Stage of Join A

In this section we derive the costs C_{join} for algorithm A. In the joining stage of this basic algorithm each processor k computes the partial join $R_k \bowtie^T S_k$. The average number of tuples in these fragments are

$$\frac{|R|}{n \cdot N} \cdot \delta_R \quad \text{and} \quad \frac{|S|}{n \cdot N} \cdot \delta_S$$

For efficiency reasons, the relation with the lower cardinality is chosen to be the outer relation. This can be easily seen by looking at the number of disk accesses for processing the join which is:

$$(1 + |\textit{inner relation}|) \cdot |\textit{outer relation}|$$

This reflects the fact that each tuple of the outer relation has to be read once whereas each tuple of the inner relation has to be read $|\textit{outer relation}|$ times. For the two existing alternatives this number is lower when

$$|\textit{inner relation}| \geq |\textit{outer relation}|$$

As stated in section 3.3 we assume $|R_k| \leq |S_k|$, i.e. R_k to be the outer and S_k to be the inner relation. Thus the disk accesses that are caused by each processor are

$$\left(1 + \frac{|S|}{n \cdot N} \cdot \delta_S\right) \cdot \frac{|R|}{n \cdot N} \cdot \delta_R$$

This is done by each processor of a node. Thus the total tuple accesses to the common disk system is

$$n \cdot \left(1 + \frac{|S|}{n \cdot N} \cdot \delta_S\right) \cdot \frac{|R|}{n \cdot N} \cdot \delta_R$$

The quotient

$$\frac{r}{w_D}$$

gives the time that is required per tuple access. Therefore the total disk I/O costs are

$$n \cdot \left(1 + \frac{|S|}{n \cdot N} \cdot \delta_S\right) \cdot \frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{r}{w_D} \tag{15}$$

As already discussed in section 4.2 these accesses cause CPU costs of

$$\left(1 + \frac{|S|}{n \cdot N} \cdot \delta_S\right) \cdot \frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{r}{b} \cdot \frac{I_{scomm}}{\mu} \tag{16}$$

per processor when a blockwise transfer is assumed. For computing the join result each tuple in R_k is compared with each tuple of S_k. If processing of such a pair of tuples requires I_{proc} instructions then further CPU costs are

$$\frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{|S|}{n \cdot N} \cdot \delta_S \cdot \frac{I_{proc}}{\mu} \tag{17}$$

(16) plus (17) give the total CPU costs of stage 2. As we assume a perfect overlap between disk I/O and CPU activities, only the maximum of the two cost expressions is relevant. Therefore the total costs of stage 2 are

$$\begin{aligned} C_2^A \;=\; & \max\left\{ n \cdot \left(1 + \frac{|S|}{n \cdot N} \cdot \delta_S\right) \cdot \frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{r}{w_D}, \right. \\ & \left. \left(1 + \frac{|S|}{n \cdot N} \cdot \delta_S\right) \cdot \frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{r}{b} \cdot \frac{I_{scomm}}{\mu} + \frac{|R|}{n \cdot N} \cdot \delta_R \cdot \frac{|S|}{n \cdot N} \cdot \delta_S \cdot \frac{I_{proc}}{\mu} \right\} \end{aligned} \tag{18}$$

The cost expressions are also shown in table 3.

Stage	Disk I/O	CPU
2	$n \cdot \left(1 + \frac{\|S\|}{n \cdot N}\delta_S\right) \cdot \frac{\|R\|}{n \cdot N}\delta_R \cdot \frac{r}{w_D}$	$\left(1 + \frac{\|S\|}{n \cdot N}\delta_S\right) \cdot \frac{\|R\|}{n \cdot N}\delta_R \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \frac{\|R\|}{n \cdot N}\delta_R \cdot \frac{\|S\|}{n \cdot N}\delta_S \cdot \frac{I_{proc}}{\mu}$

Table 3: Performance model for the joining stage of join A (stage 2)

4.4 Joining Stage of Join B

In this section we derive the costs C_{join} for algorithm B. Join B computes the three partial joins

$$\begin{array}{ll} (a) & R'_k \bowtie^T S'_k \\ (b) & R'_k \bowtie^T S''_k \\ (c) & R''_k \bowtie^T S'_k \end{array}$$

whereby

$$\begin{aligned} R_k &= R'_k \cup R''_k \\ S_k &= S'_k \cup S''_k \end{aligned}$$

The partial costs of these joins will be referred to by $C^B_{2a}, C^B_{2b}, C^B_{2c}$ respectively.

The parameters δ_R and δ_S give the average number of fragment ranges that a tuple timestamp intersects with. Thus tuples are replicated $\delta_R - 1$ and $\delta_S - 1$ times. These are the ratios $|R''_k| \,/\, |R'_k|$ and $|S''_k| \,/\, |S'_k|$ respectively. R'_k and S'_k theirselves only hold primary tuples which are – as we assume – uniformly distributed. Hence their sizes are

$$\begin{aligned} |R'_k| &= \frac{|R|}{n \cdot N} \\ |S'_k| &= \frac{|S|}{n \cdot N} \end{aligned}$$

which implies the following sizes of R''_k and S''_k:

$$\begin{aligned} |R''_k| &= \frac{|R|}{n \cdot N} \cdot (\delta_R - 1) \\ |S''_k| &= \frac{|S|}{n \cdot N} \cdot (\delta_S - 1) \end{aligned}$$

For deciding on which are the inner and which the outer relations we make the following assumptions:

$$\begin{aligned} |R'_k| &\leq |S'_k| \\ |R'_k| &\leq |S''_k| \\ |S'_k| &\leq |R''_k| \end{aligned}$$

The first assumption is arbitrary whereas the second and the third reasonable because we can expect the number of replication tuples to be significantly higher than that of the primary tuples. The left sides of these relationships give therefore the respective outer and the right sides the respective inner relations.

The performance costs of the three joins can now be derived similarly to the one in section 4.3. This leads to the equations shown in table 4. Because of the overlap between disk I/O and CPU processing only the maximum of each row is relevant, i.e.

$$\begin{aligned} C^B_{2a} &= \max\{C^B_{2a-io}, C^B_{2a-cpu}\} \\ C^B_{2b} &= \max\{C^B_{2b-io}, C^B_{2b-cpu}\} \\ C^B_{2c} &= \max\{C^B_{2c-io}, C^B_{2c-cpu}\} \end{aligned}$$

The joining costs C_{join} for join B are then

$$C^B_2 = C^B_{2a} + C^B_{2b} + C^B_{2c}$$

Stage	**Disk I/O**	**CPU**
2 (a)	$n \cdot \left(1 + \frac{\lvert S\rvert}{n \cdot N}\right) \cdot \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{r}{w_D}$	$\left(1 + \frac{\lvert S\rvert}{n \cdot N}\right) \cdot \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{\lvert S\rvert}{n \cdot N} \cdot \frac{I_{proc}}{\mu}$
2 (b)	$n \cdot \left(1 + \frac{\lvert S\rvert}{n \cdot N}(\delta_S - 1)\right) \cdot \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{r}{w_D}$	$\left(1 + \frac{\lvert S\rvert}{n \cdot N}(\delta_S - 1)\right) \cdot \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \frac{\lvert R\rvert}{n \cdot N} \cdot \frac{\lvert S\rvert}{n \cdot N}(\delta_S - 1) \cdot \frac{I_{proc}}{\mu}$
2 (c)	$n \cdot \left(1 + \frac{\lvert R\rvert}{n \cdot N}(\delta_R - 1)\right) \cdot \frac{\lvert S\rvert}{n \cdot N} \cdot \frac{r}{w_D}$	$\left(1 + \frac{\lvert R\rvert}{n \cdot N}(\delta_R - 1)\right) \cdot \frac{\lvert S\rvert}{n \cdot N} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \frac{\lvert R\rvert}{n \cdot N}(\delta_R - 1) \cdot \frac{\lvert S\rvert}{n \cdot N} \cdot \frac{I_{proc}}{\mu}$

Table 4: Performance model for the joining stage of join B (stage 2)

4.5 Joining Stage of Join C

In this section we derive the costs C_{join} for algorithm C. This algorithm differs from join B in the issues incorporated through optimisation 2 (see section 3.4):

- The sequence in which the joins are computed. Join C rearranges them to the following sequence:

$$\begin{array}{ll} (a) & R'_k \bowtie^T S''_k \\ (b) & R'_k \bowtie^T S'_k \\ (c) & R''_k \bowtie^T S'_k \end{array}$$

  The advantage is that R'_k needs to be loaded only once from disk for computation in steps (a) and (b).

- The fact that the sizes of the R'_k and S'_k are chosen to fit into main memory such that then do not have to be reloaded various times during computation.

These two issues essentially reduce the disk I/O costs at the expense of additional memory costs. As R'_K and S'_k reside in main meory we choose these as the respective inner relation in joins (a) and (c). In join (b) we will use R'_k as the inner relation as it is already kept in main memory from computation for join (a).

In order to fit the R'_k and S'_k into main memory the number m of fragments has to be chosen accordingly. If $m \leq n \cdot N$ then all joins $R_k \bowtie^T S_k$ can be computed in parallel. If $m > n \cdot N$ then the joins $R_1 \bowtie^T S_1, \ldots, R_{nN} \bowtie^T S_{nN}$ are computed in parallel first and then, in one or more subsequent rounds, the remaining ones. The number λ of such rounds that are to be performed is therefore

$$\lambda = \left\lceil \frac{m}{n \cdot N} \right\rceil$$

In section s:model:stage2B we have already determined the sizes $|R'_k|, |R''_k|, |S'_k|$ and $|S''_k|$. There, $m = n \cdot N$. In join C we have to return to using m, thus

$$\begin{aligned} |R'_k| &= \frac{|R|}{m} \\ |S'_k| &= \frac{|S|}{m} \\ |R''_k| &= \frac{|R|}{m} \cdot (\delta_R - 1) \\ |S''_k| &= \frac{|S|}{m} \cdot (\delta_S - 1) \end{aligned}$$

Now we look at the partial joins:

(a) $R'_k \bowtie^T S''_k$

We can load each tuple of S''_k once and join it with R'_k which resides in main memory. Thus, each tuple of R'_k and S''_k has to be loaded only once. This causes a data transfer of

$$\left(\frac{|S|}{m}(\delta_S - 1) + \frac{|R|}{m} \right) \cdot \frac{r}{w_D}$$

per processor. The total disk accesses per node are n times as high. As λ rounds have to be performed the total disk I/O costs C^C_{2a-io} per node are

$$\lambda \cdot n \cdot \left(\frac{|S|}{m}(\delta_S - 1) + \frac{|R|}{m} \right) \cdot \frac{r}{w_D} \tag{19}$$

The corresponding CPU costs for initiating this data transfer are

$$\lambda \cdot \left(\frac{|S|}{m}(\delta_S - 1) + \frac{|R|}{m} \right) \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu} \tag{20}$$

per processor. The CPU costs for join processing are – similarly as in join B –

$$\frac{|R|}{m} \cdot \frac{|S|}{m}(\delta_S - 1) \cdot \frac{I_{proc}}{\mu}$$

Again, as there are λ rounds, this has to be multiplied by λ:

$$\lambda \cdot \frac{|R|}{m} \cdot \frac{|S|}{m}(\delta_S - 1) \cdot \frac{I_{proc}}{\mu} \tag{21}$$

The total CPU costs C^C_{2a-cpu} for this partial join are (20) plus (21).

As mentioned at the beginning, the tuples of R'_k reside in main memory and are joined with each tuple in S''_k. This means $|R'_k| \cdot |S''_k|$ tuple accesses to main memory. The time costs per access are

$$\frac{r}{w_M}$$

As memory is shared on node level, between n processors, the total of n memory accesses of the joins that are processed on a node have to be considered. Thus the costs are

$$n \cdot \frac{|R|}{m} \cdot \frac{|S|}{m}(\delta_S - 1) \cdot \frac{r}{w_M}$$

For the λ rounds the total memory costs C^C_{2a-mem} are

$$\lambda \cdot n \cdot \frac{|R|}{m} \cdot \frac{|S|}{m}(\delta_S - 1) \cdot \frac{r}{w_M} \tag{22}$$

The total costs for the entire partial join is the maximum of (19), (20) plus (21) and (22):

$$C^C_{2a} = \max\left\{C^C_{2a-io}, C^C_{2a-cpu}, C^C_{2a-mem}\right\} \tag{23}$$

(b) $R'_k \bowtie^T S'_k$

As R'_k already resides in main memory from the previous join only the tuples of S'_k have to be loaded from disk. This causes I/O costs C^C_{2b-io} of

$$\lambda \cdot n \cdot \frac{|S|}{m} \cdot \frac{r}{w_D} \tag{24}$$

per node and CPU costs of

$$\lambda \cdot \frac{|S|}{m} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu} \tag{25}$$

per processor. Join processing requires further

$$\lambda \cdot \frac{|R|}{m} \cdot \frac{|S|}{m} \cdot \frac{I_{proc}}{\mu} \tag{26}$$

of CPU costs per processor. Finally accesses to tuples of R'_k in main memory – similar to the preceeding partial join – cause costs C^C_{2b-mem} of

$$\lambda \cdot n \cdot \frac{|R|}{m} \cdot \frac{|S|}{m} \cdot \frac{r}{w_M} \tag{27}$$

The total costs for the entire partial join is the maximum of (24), (25) plus (26) and (27):

$$C^C_{2b} = \max\left\{C^C_{2b-io}, C^C_{2b-cpu}, C^C_{2b-mem}\right\} \tag{28}$$

(c) $R''_k \bowtie^T S'_k$

Cost calculations for this join resemble very much that of join (a) with S'_k residing in main memory and the tuples of R''_k subsequently being loaded from disk. Table 5 gives the cost equations for this join and summarises the previously discussed ones.

The joining costs C_{join} for join C are the sum of the costs for the partial joins (a), (b) and (c) as described above.

Stage	**Disk I/O**	**CPU**	**Memory**
2 (a)	$\lambda \cdot n \cdot \left(\frac{\lvert S \rvert}{m}(\delta_S - 1) + \frac{\lvert R \rvert}{m} \right) \cdot \frac{r}{w_D}$	$\lambda \cdot \left(\frac{\lvert S \rvert}{m}(\delta_S - 1) + \frac{\lvert R \rvert}{m} \right) \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \ \lambda \cdot \frac{\lvert R \rvert}{m} \cdot \frac{\lvert S \rvert}{m}(\delta_S - 1) \cdot \frac{I_{proc}}{\mu}$	$\lambda \cdot n \cdot \frac{\lvert R \rvert}{m} \cdot \frac{\lvert S \rvert}{m}(\delta_S - 1) \cdot \frac{r}{w_M}$
2 (b)	$\lambda \cdot n \cdot \frac{\lvert S \rvert}{m} \cdot \frac{r}{w_D}$	$\lambda \cdot \frac{\lvert S \rvert}{m} \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \ \lambda \cdot \frac{\lvert R \rvert}{m} \cdot \frac{\lvert S \rvert}{m} \cdot \frac{I_{proc}}{\mu}$	$\lambda \cdot n \cdot \frac{\lvert R \rvert}{m} \cdot \frac{\lvert S \rvert}{m} \cdot \frac{r}{w_M}$
2 (c)	$\lambda \cdot n \cdot \left(\frac{\lvert R \rvert}{m}(\delta_R - 1) + \frac{\lvert S \rvert}{m} \right) \cdot \frac{r}{w_D}$	$\lambda \cdot \left(\frac{\lvert R \rvert}{m}(\delta_R - 1) + \frac{\lvert S \rvert}{m} \right) \cdot \frac{r}{b} \cdot \frac{I_{sio}}{\mu}$ $+ \ \lambda \cdot \frac{\lvert R \rvert}{m}(\delta_R - 1) \cdot \frac{\lvert S \rvert}{m} \cdot \frac{I_{proc}}{\mu}$	$\lambda \cdot n \cdot \frac{\lvert R \rvert}{m}(\delta_R - 1) \cdot \frac{\lvert S \rvert}{m} \cdot \frac{r}{w_M}$

Table 5: Performance model for the joining stage of join C (stage 2)

5 Evaluation

In this section we give a quantitative analysis of the parallel temporal joining techniques described in section 3 on top of the performance model that has been presented in the previous section. For this purpose we use an typical workload on an architecture with certain performance parameters. Both, workload and architectural parameters, are presented in section 5.1. In section 5.2 the analysis of the joins is presented using performance model, workload and architecture.

5.1 Workload and Architectural Parameters

Table 6 summarises the workload that we used for the experiments. It is quite modest with respect to number of tuples, tuplesize and lengths of timestamps. This has to be taken into account later as we will see that even such a modest workload can cause tremendous processing costs due to the high selectivity of a temporal intersection join.

Parameter	**Description**	**Value in the Experiments**
$\|R\|, \|S\|$	number of tuples in relations R, S	100000 tuples
r	size of a tuple in bytes	500 bytes
$\|T\|$	length of the joint relation spans $T = T_R \cup T_S$	5000 time units
τ_R, τ_S	average lengths of the tuple timestamps in R, S	100 time units
δ_R, δ_S	the average number of fragment-ranges that a tuple timestamp spans	derived from $\tau_R, \tau_S, \|T\|, m$

Table 6: The workload parameters

We assume a uniform distribution of the data over time, i.e. tuple lifespans do not vary much from their average values τ_R, τ_S and the timestamp start points are uniformly distributed over the relation lifespans T_R, T_S. This assumption is certainly ideal and in reality temporal data skew has to be considered. It allows, however, the avoidance of any (possibly unreal or application specific) assumptions about the nature of temporal data skew. Furthermore, the model can be kept simple.

The timespan T, covered by tuples from R and S, is divided into m equally sized ranges, i.e. two partition points p_i and p_{i+1} are on equal distances for $0 \leq i < m$. If $|T|$ is the length of T then

$$\frac{|T|}{m}$$

is the length of a fragment range. If τ is the average length of a tuple timestamp interval then this timestamp occupies a share of

$$\frac{\tau}{\frac{|T|}{m}} \tag{29}$$

of a fragment's range. As interval start points are distributed uniformly over the fragment range there will be some tuples whose timestamps start near the end of the range. This means that those tuples overlap the range borders (i.e. the partition points p_i). To get the average number

δ of fragment ranges that a tuple timestamp spans we have to add 1 to (29). Therefore δ_R, δ_S are given by

$$\delta_R = \frac{\tau_R}{|T|} \cdot m + 1$$
$$\delta_S = \frac{\tau_S}{|T|} \cdot m + 1$$

Remember that $m = n \cdot N$ for joins A and B.

In table 7 the parameters of the parallel architecture are described. A rate of 100 MIPS is pretty standard at the moment, e.g. it is more or less the rate of a Pentium processor. With respect to the memory size it should be emphasized that, naturally, each node has more memory than 8 MB (e.g. for the operating system kernel) but it is only these 8 MB that are available for join processing purposes.

Parameter	**Description**	**Value in the Experiments**
N	number of nodes	varied
n	number of processors per node	4
μ	processor speed in MIPS	100 MIPS
w_D	disk I/O bandwidth per node	20 MB/sec
w_C	communication bandwidth	100 MB/sec
w_M	memory bandwidth per node	400 MB/sec
I_{proc}	number of CPU instructions for processing a tuple in each step	1000
I_{exp}	number of CPU instructions for computing arithmetic expressions $fragment_P(t)$	$\frac{I_{proc}}{10} = 100$
I_{scomm}	number of CPU instructions for initiating a data transfer	500
I_{sio}	number of CPU instructions for initiating a disk I/O	500
mem	amount of shared memory per node avalaible for data structures	8 MB
b	page size	4 kB

Table 7: The parameters describing the parallel architecture

5.2 Analysis

The performance model was used to run several experiments. First of all, we compared the performance of the three joins with respect to the workload of table 6 and varying the number N of nodes. Figure 2 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 that were discussed in section 3.4:

- Optimisation 1 (join A $\rightarrow$ join B) improves performance between 20% ($N = 10$) and 65% ($N = 50$).

- Optimisation 2 (join B $\rightarrow$ join C) decreases costs furthermore by 90%.
- Optimisations 1 and 2 (join A $\rightarrow$ join C) give a composite improvement of around 95%.

All join costs are dominated by the costs for stage 2; partitioning costs and therefore communication costs can 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 I/O bandwidth, more memory and more CPU power. This explains the ideal scaleup in figure 2.

Figure 3 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'' in (4) and (5). The *primary join costs* are the costs for performing the join $R_k' \bowtie^T S_k'$.

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.

Up to now, the workload of table 6 did not require m to exceed nN in order to reduce the size of the R_k', S_k' such that they fit in main memory (see optimisation 2). In other words: it is $\lambda = 1$ in figures 2 and 3.

In figure 4 the costs of join C are shown for the tuple size r being varied. The graphs break off at around 1050 and 1678 bytes. The first breakpoint is caused by the fact that memory costs overtake CPU costs. The second one is due to the choice of m being a multiple of nN for simplicity Passing the '1678-bytes-point' λ changes from 1 to 2 (see section 4.5). The crease suggests that this choice is not optimal but not bad either. The share of the replication overhead remains in the 65% to 75% margin throughout the experiment[7].

[7] This margin, of course, certainly depends on the workload, in particular on the lengths of tuple timestamps. We consider the average timestamp / relation lifespan ratio $\tau/|T|$ of 2% in our example as low. Higher ratios can perfectly be expected. These would increase the 70% margin.

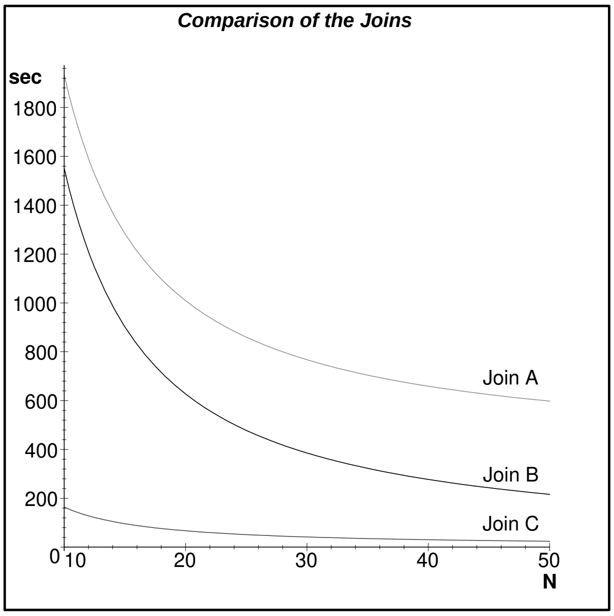

Figure 2: Performances of the three join algorithms

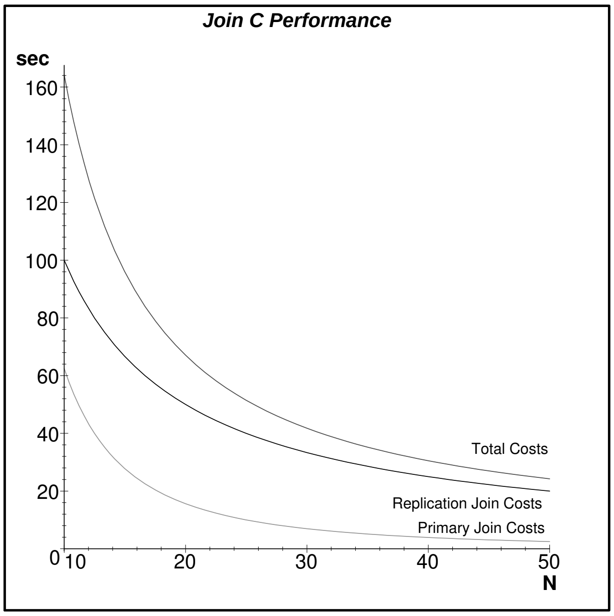

Figure 3: The components of the join C costs

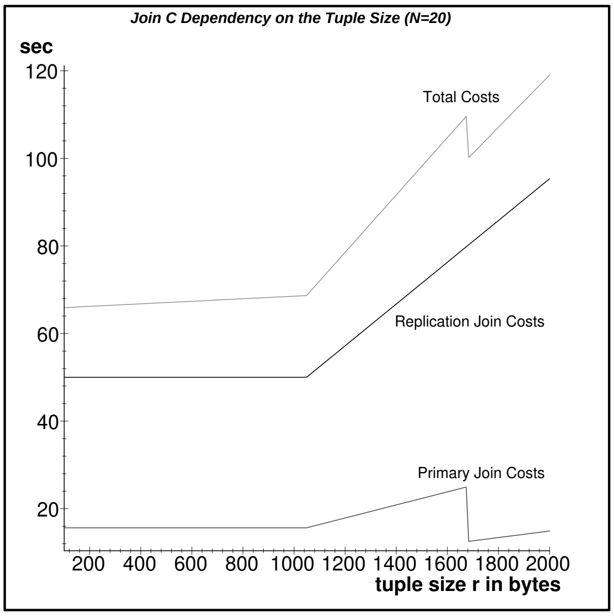

Figure 4: The costs of join C depending on the tuple size r

6 Conclusions

In this report we discussed the parallel processing of temporal joins. We focused mainly 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 consists of non-equality predicates. Data partitioning 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 considerable overhead.

We showed that this overhead can be reduced significantly if one divides a partition 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 we gave a performance model for three parallel joins. This was based on a general-purpose parallel hardware architecture. This makes results generally useful. The joins were evaluated on top of this model using a certain workload. 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.

References

Allen, J. (1983). Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843.

Baru, C., Fecteau, G., Goyal, A., Hsiao, H., Jhingran, A., Padmanabhan, S., Copeland, G., and Wilson, W. (1995). DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322.

Gunadhi, H. and Segev, A. (1990). A Framework for Query Optimization in Temporal Databases. In Michalewicz, Z., editor, *Proc. of the 5th International Conf. on Statistical and Scientific Database Management, Charlotte, NC, USA*, number 420 in Lecture Notes in Computer Science (LNCS), pages 131–147. Springer.

Gunadhi, H. and Segev, A. (1991). Query Processing Algorithms for Temporal Intersection Joins. In *Proc. of the 7th International Conference on Data Engineering, Kobe, Japan*, pages 336–344.

Hua, K., Lee, C., and Peir, J.-K. (1991). Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing. In *Proceedings of the 1st International Conference on Parallel Distributed Information Systems, Miami Beach, FL, USA*, pages 262–270.

Leung, T. and Muntz, R. (1990). Query Processing for Temporal Databases. In *Proc. of the 6th International Conference on Data Engineering, Los Angeles, CA, USA*, pages 200–208.

Leung, T. and Muntz, R. (1992). Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proc. of the 18th International Conference on Very Large Data Bases, Vancouver, Canada*, pages 383–394.

Lu, H., Ooi, B.-C., and Tan, K.-L. (1994). On Spatially Partitioned Temporal Join. In *Proc. of the 20th Internat. Conf. on Very Large Data Bases (VLDB), Santiago de Chile*, pages 546–557.

Mishra, P. and Eich, M. (1992). Join Processing in Relational Databases. *ACM Computing Surveys*, pages 63–113.

Norman, M. and Thanisch, P. (1995). *Parallel Database Technology: An Evaluation and Comparison of Scalable Systems*. The Bloor Research Group, UK. ISBN 1-874160-17-1.

Piatetsky-Shapiro, G. and Connell, C. (1984). Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings ACM SIGMOD 1984 Conference on Management of Data*, pages 256–276.

Rana, S. and Fotouhi, F. (1993). Efficient Processing of Time-Joins in Temporal Data Bases. In *Proc. of the 3rd Internat. Symposium on Database Systems for Advanced Applications*, pages 427–432.

Soo, M., Snodgrass, R., and Jensen, C. (1994). Efficient Evaluation of the Valid-Time Natural Join. In *Proc. of the 10th International Conference on Data Engineering, Houston, Texas, USA*, pages 282–292.