Optimisation of Partitioned Temporal Joins

Thomas Zurek

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

Abstract. Partitioning data for temporal join processing is not trivial because tuples have to be replicated between data fragments. This causes three types of overheads: (a) an overhead caused by the replication process itself, (b) a processing overhead caused by the additional joining that has to be done and (c) an overhead for removing duplicates in the result. Previous work has mainly concentrated on avoiding (a) but still suffers from the consequences of (b) and (c).
In this paper, we show how partitioned temporal joins can be optimised for sequential and parallel processing by reducing tuple replication, thereby reducing the total overhead. For that purpose, a new data structure, namely the IP-table, is introduced. The idea is to have IP-tables of individual temporal relations stored in the database system's catalog from which they can be retrieved for the optimisation process. IP-tables of two or more temporal relations might be required for optimisations, too. These can be created by merging IP-tables of individual relations at optimisation time – a fast and straightforward process.
IP-tables can be used for creating *and* analysing partitions over interval timestamps. Three strategies for partitioning interval data are presented, each of which can be easily and efficiently implemented using IP-tables. The performance determining parameters of a partition can also be derived from IP-tables.

Keywords: temporal join, parallel join, optimisation, interval partitioning.

1 Introduction

Recent years have seen a significantly increasing interest by the industrial and academic community in modeling and storing data that changes over time. One important example is trend analysis, e.g. in the context of data warehouses [7]. Temporal databases have also been the focus of a large number of research papers [9]. With this development comes a whole bunch of new problems: temporal data models, temporal query languages, temporal operators etc. are required. In this paper, we look at one particular performance-critical temporal operator, namely the temporal join, and ways in which temporal join processing can be optimised. We thereby concentrate purely on temporal-specific optimisation and regard our proposal as an enhancement of existing relational optimisation techniques.

In the past, very efficient algorithms have been developed for equi-joins, i.e. joins with an equality join condition. These are based on several forms of **partitioning** [5]. Explicit partitioning – as one form that is, for example, used in hash or parallel joins – breaks up the one 'big' join operation into several *smaller* and *independent* joins. This can be summarised in the following equation:

$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q_1 \cup \cdots \cup R_m \bowtie_C Q_m \tag{1}$$

where the R_k and Q_k are referred to as *fragments* of the relations R and Q, respectively. C is a boolean expression and said to be the *join condition*.

(1) works very well for equi-joins because it is easy to create *disjoint* fragments $R_1, \ldots, R_m$ ($Q_1, \ldots, Q_m$ respectively). In practice, most join conditions in real queries are built on an equality predicate such as "key = foreign key" conditions, as in the case of a customers table being joined with an address table via equal name attribute values.

In the case of **temporal joins**, however, the join conditions are usually based on a non-equality predicate. We adopt the most frequently used temporal data model in which tuples of a temporal relation have an interval timestamp. Intervals have proved to be the most versatile representation of time: intervals and relationships between intervals can adequately express almost any time reference in natural language [1]. *Intersection* of two intervals is the most prominent and the most general join predicate in this case. It is a supertype of most other possible relationships between two intervals. Furthermore, it plays a similarly important role for interval data as equality for atomic data: it establishes the notion of *simultaneity* of two temporal objects.

A join condition based on intersection of two timestamp intervals, however, imposes a problem on partitioning temporal joins as in (1): it is usually impossible to create fragments $R_1, \ldots, R_m$ ($Q_1, \ldots, Q_m$ respectively) that are disjoint. Consider the example of figure 1. It visually describes the computation of a temporal join $R \bowtie Q$ that is partitioned into three partial joins. Tuples of R are put along a horizontal axis, tuples of Q along a vertical axis. R and Q are partitioned into three fragments respectively. Squares symbolise possible combinations of tuples of R and Q. The figure shows nested-loop computations for the partial joins $R_k \bowtie Q_k$, i.e. every tuple in R_k is compared with every tuple in Q_k. Tuple comparisons that are successful are shown in dark grey, comparisons that are performed in a preceding partial join are in black, unsuccessful comparisons in light grey. Figure 1 makes the following problems evident:

(a) Tuples have to be replicated between fragments. This causes an overhead (*replication overhead*). Figure 1 has a total of 13 tuple replications (5 for R, 7 for Q).
(b) As a consequence, the fragments grow. This implies more tuple comparisons (*processing overhead*). Figure 1 has a total of 132 comparisons. Using different breakpoints of the time domain, e.g. $\{5, 9\}$ instead of $\{4, 7\}$ as in figure 1, would lead to 156 comparisons, i.e. 18% more.
(c) Finally, the tuple replications cause duplicates in the result (see black squares). This contributes to the processing overhead but also causes an additional

problem: duplicates might have to be removed from the result. This can be quite expensive (*duplicates overhead*).

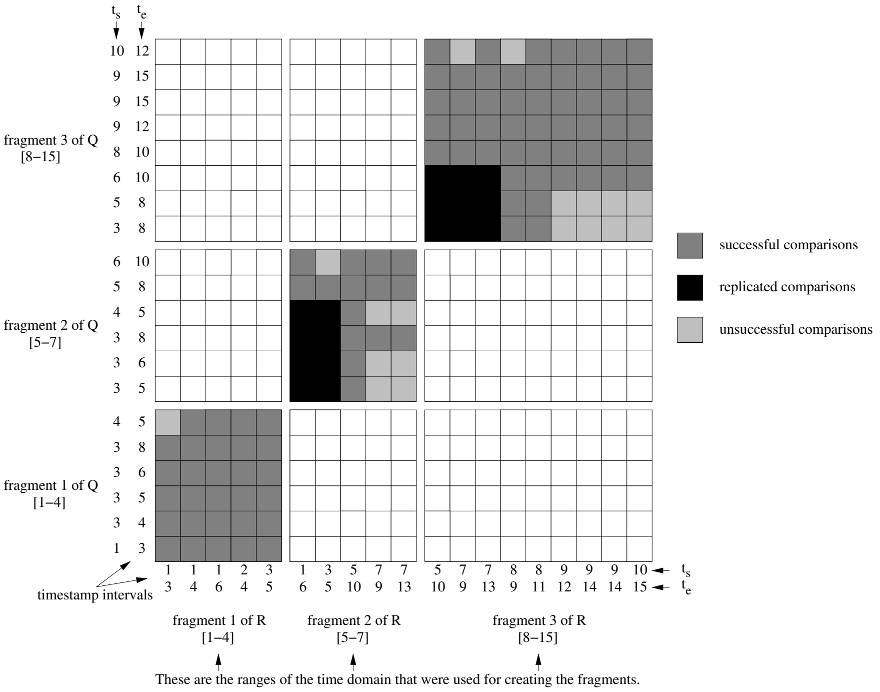


Fig. 1. Partitioned temporal join computation.

From the scenario of figure 1 we can conclude that badly chosen partitions can impose a significant overhead on the join processing costs. The major contribution of this paper is to show how the processing of partitioned temporal joins can be optimised by choosing an appropriate way of partitioning the temporal data. We will show which information about the temporal data has to be acquired in order to create *and* to analyse partitions (i.e. to predict their performance effects) and how this process can be done efficiently such that an optimisation decision can be taken before the join is processed.

First, we look at some previous work on partitioned temporal joins in section 2 and introduce the notations that are used throughout the paper in section 3. The structure of a temporal join optimiser is presented in section 4. Section 5 introduces the essential data structure, the *IP-table*[1], on which our proposal is based. Section 6 describes three partitioning strategies which can be easily and efficiently implemented using IP-tables. Finally, section 7 summarises and concludes the paper.

[1] IP = Interal Partitioning

2 Previous Work

Several authors have discussed temporal join processing based on explicit partitioning:

Leung and Muntz discuss partition-based temporal joins for parallel processing [3]. They identify the special cases of temporal intersection for which tuple replication can be (partially) avoided. We, however, concentrate on the cases in which replication *cannot* be avoided. Therefore, our work can be regarded as an enhancement of Leung and Muntz's approach.

Soo *et al.* propose a valid-time join algorithm which is based on a partitioning approach as in (1) [8]. However, the partial joins[2] $R_m \bowtie Q_m, \ldots, R_1 \bowtie Q_1$ are processed *sequentially* (in that order) whereby tuples that have to be present in the following partial join are kept in a cache. This technique avoids the overhead (a) caused by the actual replication but does not reduce the processing overhead (b) caused by the fact the number of tuples in the fragments is significantly higher for badly chosen partitions (see section 1). The latter, however, has proved to be the critical one [10]. Our optimisation techniques can therefore help to overcome this deficit.

Lu *et al.* map interval data to points in a two dimensional grid which is divided into disjoint fragments afterwards [4]. What appears to be an advantage at first sight is unfortunately built on the expense of a major drawback: a fragment R_k is not only to be joined with one corresponding fragment Q_k but with several others. As in the previous case, they can avoid the actual replication of tuples (a) but remain with the processing overhead (b).

Finally, two optimisations of partitioned temporal join processing are proposed in [12]. One reduces additional I/O caused by tuple replication, the other shows how duplicates can be avoided, thus reducing the processing overhead (b) and avoiding the duplicates overhead (c).

3 Notational Conventions

We adopt the following notations:

A *temporal relation* R is a relation in which each tuple carries one timestamp, the latter being represented as an interval. The restriction to one timestamp per tuple only simplifies the following notations. In general, two or more timestamps per tuple are possible. In this case, we concentrate on the one that is used for partitioning purposes.

Intervals are defined over a certain domain. For our purposes, we can assume the set of integers to be the domain. We assume that the left and right neighbours of a time point t can be referred to by $t-1$ and $t+1$ respectively given by the discreteness[3] of a total ordering defined on the domain. *Intervals* have the

[2] In the following we omit the reference to a join condition C and use $R \bowtie Q$ meaning a temporal intersection join between R and Q.

[3] Discreteness is merely used for simplifying the notation and does not represent a necessary precondition for the work presented in this paper.

following notations:

$$[t_s, t_e] = \{x : t_s \leq x \leq t_e\}, \qquad (t_s, t_e] = \{x : t_s < x \leq t_e\} = [t_s + 1, t_e]$$

The left delimiter, t_s, of an interval is called the interval's *startpoint*; the right one, t_e, is the *endpoint*. We use the first type in the timestamps of the temporal relations that are to be partitioned and the second type for the partitioning segments (see below).

In order to simplify expressions we use the shortcut notation $[t_s, t_e] \in R$ to express that $[t_s, t_e]$ is a timestamp of one or more tuples in R.

The *range* $T(R)$ of the intervals of a temporal relation R is the part of the domain covered by the intervals in R. The *lifespan* $L(R)$ of R is interval that contains every interval of R. $T(R)$ and $L(R)$ are defined as

$$\begin{aligned} T(R) &= \{t : t \text{ is contained in some interval } [t_s, t_e] \in R\} \\ L(R) &= [\min T(R), \max T(R)] \end{aligned}$$

The lifespan may contain parts of the domain that are not covered by any interval and are therefore not included in the range $T(R)$. The sets of interval startpoints, $S(R)$, and endpoints, $E(R)$, are defined by

$$\begin{aligned} S(R) &= \{t_s : \exists t_e \in T(R) \text{ such that } [t_s, t_e] \in R\} \\ E(R) &= \{t_e : \exists t_s \in T(R) \text{ such that } [t_s, t_e] \in R\} \end{aligned}$$

The following functions are defined on the time domain:

$$\begin{aligned} s_R(t) &= \text{number of intervals in } R \text{ that start at time point } t \\ e_R(t) &= \text{number of intervals in } R \text{ that end at time point } t \\ i_R(t) &= \text{number of intervals in } R \text{ that intersect with / contain time point } t \\ o_R(t) &= \text{number of intervals in } R \text{ that overlap time point } t \\ &\quad\ \text{(i.e. intersect with } t \textit{ but do not end at } t) \end{aligned}$$

A *partition* P is an ordered set $\{p_1, \ldots, p_{m-1}\}$ of *breakpoints* that divide the lifespan and range into m segments $(p_{k-1}, p_k]$ for $k = 1, \ldots, m$ with p_0 and p_m being the left and right border of the plot, for example, $p_0 = \min T(R) - 1$ and $p_m = \max T(R)$.

4 Structure of the Optimisation Process

Figure 2 shows the dataflow in the optimisation process for a partitioned temporal join between two temporal relations R and Q. Data is represented as rectangles, computation as ovals. The entire process consists of four stages corresponding to the four grey boxes in figure 2:

1. Firstly, the temporal relations have to be analysed to acquire some information about the structure and characteristics of the temporal data. In its simplest form, this information can be represented by a data sample that has been drawn from the two relations. This approach has been taken in [8]. A second form is to get some meta-information on the data which might be stored in the database catalog. We follow the latter approach and define *IP-tables* for this purpose in section 5.

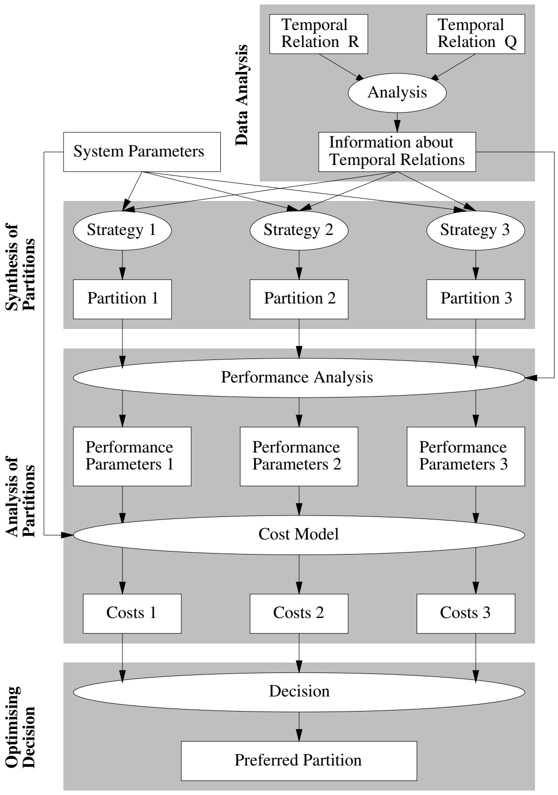


Fig. 2. Structure of the optimisation process.

2. Based on the information acquired in stage 1 and the systems parameters (number of processing nodes, amount of memory etc. that are currently available) several strategies can be applied to find suitable partitions for the temporal data. Section 6 presents three such strategies.
3. Using the partitions and information on the temporal data, performance-determining parameters, such as the loads of the fragments R_k and Q_k, can be approximated (e.g. when using data samples) or exactly calculated (e.g. by our approach). These parameters are then fed into a cost model which derives the processing costs of the partitioned temporal join based on the respective partition and the current system parameters.
4. Finally, the cheapest partition is chosen.

5 IP-Tables

5.1 Motivation

Before defining IP-tables in the following section we want to make two observations which serve as a motivation:

① The most important parameters that determine the performance of a partitioned join as in (1) are the cardinalities $|R_k|$ and $|Q_k|$ of the fragments R_k and

Q_k. Imagine now a temporal relation R and a partition $\{p_1, \ldots, p_{m-1}\}$. Then the number $|R_k|$ of tuples in a fragment R_k can be determined by (i) the number of tuples that overlap from preceding fragments R_j $(j < k)$ plus (ii) the tuples that start in the time segment $(p_{k-1}, p_k] = [p_{k-1}+1, p_k]$ that corresponds to R_k. Obviously (i) corresponds to the number of intervals which contain the first point in the time segment but start before, i.e. the number $o_R(p_{k-1})$, whereas (ii) can be determined by summing up the values $s(p_{k-1}+1), \ldots, s(p_k)$. Thus

$$|R_k| \quad = \quad o_R(p_{k-1}) \; + \sum_{t=p_{k-1}+1}^{p_k} s_R(t) \tag{2}$$

Several other performance influencing parameters can be calculated in a similar way such as the following two:

$$\begin{aligned} \textit{number of overlapping intervals} &= \sum_{k=1}^{m-1} o_R(p_k) \\ \textit{average interval length} &= \frac{1}{|R|} \cdot \sum_{t=\min T(R)}^{\max T(R)} s_R(t) + o_R(t-1) \end{aligned}$$

② For our purposes we require only the values $s_R(t)$ and $o_R(t)$ for those t at which at least one interval starts or ends: for all other time points t it is $s_R(t) = 0$ and $o_R(t) = o_R(t')$ with t' being the next start- or endpoint (of some interval) before t. It is only there where the sum (2) changes. This allows us to concentrate on the start- and endpoints of the intervals rather than the entire time span [11].

5.2 Definition

An IP-table for one or more temporal relations stores information about the temporal structure of the time intervals appearing in these relations; the IP-table is specific to those temporal relations. We now define an IP-table for a temporal relation R; the definition for two or more relations works accordingly. The IP-table for R consists of three columns, each with N entries. Initially, N is the number of distinct start- and endpoints used in intervals of R; in section 5.4 we show how N can be reduced if the IP-table gets too big:

- The first column contains the values

  $V(R) \;=\; S(R) \cup E(R) \;=\; \{t_1, \ldots, t_N\}$ such that $t_j < t_{j+1}$ for $j = 1, \ldots, N-1$.

- The second column holds the values $s_R(t_j)$ for $j = 1, \ldots, N$.
- The third column holds the values $o_R(t_j)$ for $j = 1, \ldots, N$.

For each t_j we can derive the values $e_R(t_j)$ and $i_R(t_j)$ by using the following equations:

$$
\begin{aligned}
i_R(t_1) &= s_R(t_1) && \\
i_R(t_j) &= s_R(t_j) + o_R(t_j - 1) && \text{for } j = 2, \ldots, N \\
e_R(t_j) &= i_R(t_j) - o_R(t_j) && \text{for } j = 1, \ldots, N
\end{aligned}
$$

Using these constraints it becomes obvious that we can alternatively store any pair of the values $s_R(t_j), e_R(t_j), i_R(t_j), o_R(t_j)$.

Figure 7(a) shows an example for timestamp intervals of a temporal relation R. Intervals are represented as bold bars connecting their start- and endpoint respectively. Figure 8 shows the IP-table for R (in bold typeface) plus the derivable values $e_R(t_j)$ and $i_R(t_j)$ for demonstration purposes.

5.3 Merging IP-Tables

Two (or more) IP-tables of two (or more) temporal relations can be merged into one IP-table that describes the characteristics of the union of the participating relations. This is very useful as we can precompute the IP-tables for individual relations and merge them when optimising a temporal join between those relations. This is only relevant for join algorithms that require two or more input relations to be partitioned along certain constraints. Partitioning then needs information on all these relations, thus we need the IP-table of the union of the relations.

The merging process is fairly straightforward: Imagine two IP-tables of two relations R and Q participating in a temporal join $R \bowtie Q$. They have time point sets $V(R)$ and $V(Q)$ and functions s_R, o_R and s_Q, o_Q respectively. These two tables can be merged into one IP-table with time point set

$$V(R \cup Q) \;=\; V(R) \cup V(Q)$$

and functions $s_{R\cup Q}, o_{R\cup Q}$ defined as

$$
\begin{aligned}
s_{R\cup Q}(t) &= s_R(t) + s_Q(t) \\
o_{R\cup Q}(t) &= o_R(t) + o_Q(t)
\end{aligned}
$$

Please note that the IP-table of R might not hold values for all $t \in V(Q)$ and, similarly, the IP-table of Q not for all $t \in V(R)$. This is not actually a problem as these values can be derived by using the second observation made in section 5.1.

The correctness of the merge process as described should be obvious. We omit a formal proof due to the space restrictions for this paper. Figure 3 shows how the IP-table approach fits into the optimisation process as outlined in figure 2.

5.4 Condensing IP-Tables

In section 5.2, we assumed that the number N of entries in the IP-table equals the number of distinct interval start- and endpoints, i.e. $N \;=\; |S(R) \cup E(R)|$.

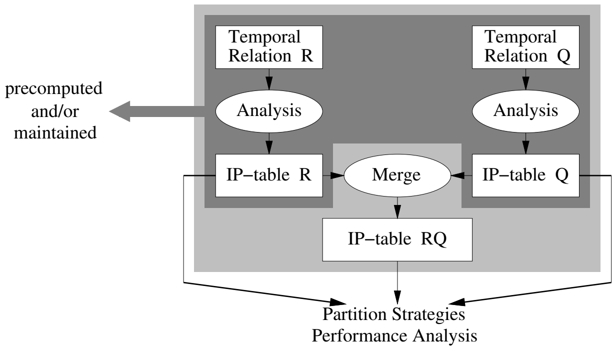


Fig. 3. Acquiring information about (temporal) characteristics of temporal relations by using IP-graphs

Critics might argue that – in the worst case – a temporal relation with one million tuples has an IP-table with two million entries and that this might cause the IP-table approach to be unusable. In the following, we want to address these concerns.

First, IP-table entries are very short: a time point or chronon, t_j, might be represented as 6 bytes, and the two integers, $s_R(t_j)$ and $o_R(t_j)$, say 4 bytes each. In total these are 14 bytes. Considering a typical tuple size of 500 bytes or more then an IP-table will still have only $\frac{2 \cdot 14}{500} = 5.6\%$ of the size of the originating relation.

Second, we analysed several real-world temporal relations[4] R and found ratios $\frac{|S(R) \cup E(R)|}{|R|}$ between 0.09 and 0.12, i.e. far away from the 2.0 for the worst case. If we assume a tuple size of 500 bytes then the IP-table would be less than 0.4% of the size of the originating relation. This is far below the 3% for data samples claimed in [8].

Third, IP-tables can be condensed (compressed). One possibility is to use only every a-th element of $S(R) \cup E(R)$. This results in values $t'_j, s'_R(t'_j), o'_R(t'_j)$ which can be derived according to the following equations for $j = 1, \ldots, N'$:

$$N' = \left\lfloor \frac{N}{a} \right\rfloor, \quad t'_j = t_{j \cdot a}, \quad s'_R(t'_j) = \sum_{l=(j-1)a+1}^{j \cdot a} s_R(t_l), \quad o'_R(t'_j) = o_R(t_{j \cdot a})$$

This process divides the size of the IP-table by a, thus the figure 0.4% obtained above is reduced to 0.04% for $a = 10$. Naturally, the IP-table becomes coarser and less precise. This decreases the quality of the resulting partitions but makes the process of deriving the partition more efficient. We conducted various experiments on this issue and found almost no drawbacks for $a \leq 10$.

[4] Login information to several computing clusters and airline timetables.

6 Partitioning Strategies

There are various ways to partition interval data. In this section, we present three strategies, all of which can be based on the information stored in an IP-table. They create partitions $\{p_1, \ldots, p_{m-1}\}$ for computing a partitioned temporal join (1) of two relations R and Q. Ideally, the fragments $R_1, \ldots, R_m$ and $Q_1, \ldots, Q_m$ are well balanced with respect to each other, i.e. the R_j and Q_j contain a similar number of tuples, and the total number of tuples that have to be replicated is as small as possible.

6.1 Uniform Strategy

This is the simplest strategy. It divides either the joint range $T(R \cup Q)$ – as in [8] – or the joint lifespan $L(R \cup Q)$ into m equally sized segments. As a third option we propose to divide the startpoints' span

$$SP(R \cup Q) = [\min S(R \cup Q), \max S(R \cup Q)]$$

for the following reason: after $\max S(R \cup Q)$ no more intervals start, i.e. no more intervals are added to the plot. If a breakpoint p_k was chosen after that point the fragments R_{k+1} and Q_{k+1} would hold only intervals that are already in R_k and Q_k and thus would already be joined in $R_k \bowtie Q_k$. Thus a join $R_{k+1} \bowtie Q_{k+1}$ would be without significance. It is therefore feasible to divide the startpoints' span rather than the lifespan in order to avoid such a situation.

The three different spans can be easily calculated by using the IP-table for $R \cup Q$. Let $V(R \cup Q) = \{t_1, \ldots, t_N\}$ with $t_j < t_{j+1}$.

- The lifespan $L(R \cup Q)$ is simply the span between the first and the last time point in the table, i.e. $[t_1, t_N]$.
- The range $T(R \cup Q)$ is the lifespan minus certain gaps. The latter can be simply identified because they are exactly those areas between a t_{j-1} and a t_j that are not overlapped by any intervals, i.e. $o_{R \cup Q}(t_j) = 0$. Thus all entries in the table with a 0 entry in the third column identify a gap.
- Finally, the startpoints' span is simply the interval between t_1 and the highest t_j entry such that $s_{R \cup Q}(t_j) > 0$.

If either the lifespan or the startpoints' span is used, a uniform partition $P = \{p_1, \ldots, p_{m-1}\}$ can be computed as $p_k = t_1 - 1 + \lceil k \cdot \frac{\text{length of the span}}{m} \rceil$ for $k = 1, \ldots, m-1$. In the case of partitioning $T(R \cup Q)$ the gaps have to be considered as in the algorithm of figure 4. Figure 7(b) shows a uniform (lifespan) partition of the example of figure 7(a). Circled numbers show the number of intervals per fragment; numbers below the breakpoint lines show the number of overlapping intervals $o_R(p_k)$ at the respective breakpoint p_k.

It is obvious that a uniform strategy is liable to perform badly in the case of skewed, i.e. not uniformly distributed, data. There is hardly any control over the fragments' loads and thus over the load balance. There are, however, many examples for temporal relations that have periodically repeated patterns of tuple timestamps: imagine a relation logging the starting and ending times of calls made by customers of a telephone company. The distribution of phone calls over a daytime period will vary significantly with many calls during business hours and few in the early morning and late evening. Considering a long time period, however, a daily pattern is probably repeated periodically. Thus we can expect a poor partition result when uniformly partitioning a one-day-span but a much better one for a period comprising several days, in particular when m matches the number of days.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

target = ⌈ length of the range / m ⌉   /* target length of the segments */
k      = 1                             /* number of next p_k to be computed */
length = 0                             /* length of current segment */

for j = 2 to N do               /* scans the IP-table for R ∪ Q */
   if length > target then      /* current segment exceeds target length */
      p_k    = t_{j-1}
      k      = k + 1
      length = 0
   fi
   if o_{R∪Q}(t_j) > 0 then     /* consider length only if there was no gap */
      length = length + (t_j - t_{j-1})
   fi
od
```

Fig. 4. Algorithm for partitioning $T(R \cup Q)$ uniformly.

6.2 Underflow Strategy

The goal of this strategy is to sequentially fill the fragments R_1, R_2, ... (Q_1, Q_2, ... , respectively) such that a given number X of tuples per fragment is 'underflowed', i.e. not exceeded. The IP-tables for the individual relations R and Q together with equation (2) can be used for this purpose. Figure 5 presents an algorithm that implements this strategy using the IP-tables of R and Q. Figure 7(c) shows the partition resulting from partitioning the example of figure 7(a) using the underflow strategy.

The advantage of this strategy is that the fragments' loads are well controlled and can be expected to be well balanced as long as the values $s_R(t_j)$ are relatively small in order to approach the limit X as close as possible (see if-condition in figure 5). There is, however, no control over the number of intervals overlapping the breakpoints. Incorporating a mechanism for this is the goal of the strategy discussed in section 6.3.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

k      = 1          /* number of next breakpoint p_k to be computed */
load_R = s_R(t_1)   /* load of current fragment R_k */
load_Q = s_Q(t_1)   /* load of current fragment Q_k */

for j = 2 to N do
  if load_R + s_R(t_j) > X or load_Q + s_Q(t_j) > X then  /* |R_k| or |Q_k|   */
     p_k    = t_{j-1}                                     /* would exceed     */
     k      = k + 1                                       /* the max. load X  */
     load_R = o_R(t_{j-1})
     load_Q = o_Q(t_{j-1})
  fi
  load_R = load_R + s_R(t_j)
  load_Q = load_Q + s_Q(t_j)
od
```

Fig. 5. Algorithm for the underflow strategy using the IP-tables of R and Q.

6.3 Minimum-Overlaps Strategy

Similar to the underflow strategy the goal of this one is to create fragments R_1, R_2, ... (Q_1, Q_2, ... , respectively) such that a given number X of tuples is not exceeded *and* that the sum of intervals overlapping the breakpoints is minimal. This problem has a polynomial solution which is based on a dynamic programming approach [11]. Describing the algorithm in detail would go beyond the space limits of this paper. We therefore restrict ourselves to the following informal description:

The time points $t_1, \ldots, t_N$ of the IP-table of $R \cup Q$ are processed from 1 to N. For each t_j a partition of the span $[t_1, t_j]$ is computed that has minimal sum of overlaps and has fragment loads less than X. More precisely, for each t_j a predecessor $pred(t_j)$ is determined which represents the preceding breakpoint that leads to this 'optimal' partition. The final minimising partition is then given by the sequence $pred(t_N), pred(pred(t_N)), \ldots$ (until a delimiting dummy point t_0 appears). Figure 6 summarises the algorithm. In figure 7(d) the partition is shown that results from partitioning the example of figure 7(a) using the minimum-overlaps strategy.

This strategy can be expected to perform at least as well as the underflow strategy. Its disadvantage, however, is its run time complexity of $O(N^2)$ compared to $O(N)$ of the underflow strategy which restricts its suitability to small IP-tables.

7 Summary and Conclusions

In this paper, we have shown how partitioned temporal joins can be optimised for sequential and parallel processing. For that purpose, a new data structure,

/* $V(R \cup Q) = \{t_1, \ldots, t_N\}$ with $t_j < t_{j+1}$ */

$t_0 = -\infty$ /* a dummy point */

for $j = 1$ **to** N **do**

among the points $t_0, \ldots, t_{j-1}$ choose a point, denoted as $pred(t_j)$, such that the numbers of intervals of R and Q that are respectively intersecting with the segment $(pred(t_j), t_j]$ does not exceed X *and* such that the total sum of overlaps $\sum o_R(pred^z(t_j)) + o_Q(pred^z(t_j))$ (with $z \geq 1$ and $pred^z(t_j) > t_0$) is minimal*.

od

The minimising partition is given by the sequence $pred(t_N), pred(pred(t_N)), \ldots$ (it stops before t_0 appears).

* $pred^z(t_j) = \underbrace{pred(\ldots pred}_{z}(t_j)\ldots)$

Fig. 6. Algorithm of the minimal overlaps strategy for relations R and Q.

namely the IP-table, has been introduced. The idea is to have IP-tables of individual temporal relations stored in the database system's catalog from which they can be retrieved for the optimisation process. IP-tables of two or more temporal relations might be required for optimisations. These can be created by merging IP-tables of the individual relations – a fast and straightforward process – at optimisation time.

IP-tables can be used for creating *and* analysing partitions over interval timestamps. Creating such a partition is not trivial because it determines significant overhead costs caused by the necessary replication of tuples. Three strategies for partitioning interval data have been presented, each of which can be easily and efficiently implemented using IP-tables. The performance determining parameters of a partition can also be derived from the IP-tables of the relations that participate in the join, such as the fragments loads (2) or the total number of tuples to be replicated.

The IP-table approach has major advantages over alternative techniques such as optimisations based on data samples. Firstly, IP-tables can be precomputed and stored in the system's catalog or – in case of an IP-table for more than one relation – can be easily computed by using precomputed tables. Secondly, IP-tables are much smaller data structures than e.g. data samples. Thus they can be scanned and analysed much more efficiently. This accelerates the creation and analysis stages of the optimisation process. Thirdly, the information stored in IP-tables is precise. Information drawn from data samples, however, is only an approximation. In summary: the IP-table approach is faster and more precise.

In the future, we want to analyse the IP-table approach is some more detail. IP-tables are similar to histograms as used in join selectivity estimation [2], [6] and, in fact, can be used for this purpose as well [13]. We will investigate this relationship in more detail.

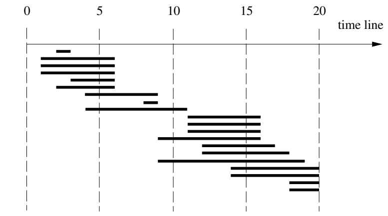
Acknowledgement

I would like to thank my department for their financial support for the publication of this paper.

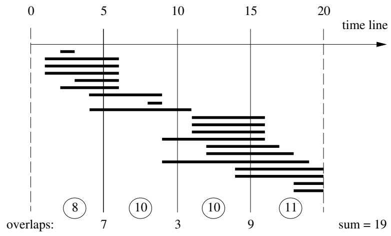
References

1. J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, Nov. 1983.
2. R. Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, Case Western Reserver University, Sept. 1980.
3. T. Leung and R. Muntz. 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, Aug. 1992.
4. H. Lu, B.-C. Ooi, and K.-L. Tan. On Spatially Partitioned Temporal Join. In *Proc. of the 20th Internat. Conf. on Very Large Data Bases (VLDB), Santiago de Chile*, pages 546–557, Sept. 1994.
5. P. Mishra and M. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, pages 63–113, Mar. 1992.
6. V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings ACM SIGMOD Conference on Management of Data, Montreal, Canada*, pages 294–305, June 1996.
7. Red Brick Systems. The Data Warehouse. White paper, Red Brick Systems, Aug. 1995.
8. M. Soo, R. Snodgrass, and C. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proc. of the 10th International Conference on Data Engineering, Houston, Texas, USA*, pages 282–292, Feb. 1994.
9. V. Tsotras and A. Kumar. Temporal Database Bibliography Update. *SIGMOD Record*, 25(1), Mar. 1996.
10. T. Zurek. Parallel Temporal Nested-Loop Joins. Technical Report ECS-CSG-20-96, Dept. of Computer Science, Edinburgh University, Jan. 1996.
11. T. Zurek. Optimal Interval Partitioning for Temporal Databases. In *Proc. of the 3rd BIWIT Workshop, Biarritz, France.* IEEE Computer Society Press, July 1997.
12. T. Zurek. Parallel Temporal Joins. In *"Datenbanksysteme in Büro, Technik und Wissenschaft" (BTW), German Database Conference, Ulm, Germany*, pages 269–278. Springer, Mar. 1997. (in English).
13. T. Zurek. Size and Selectivity Calculation for Temporal Joins. submitted for publication, July 1997.

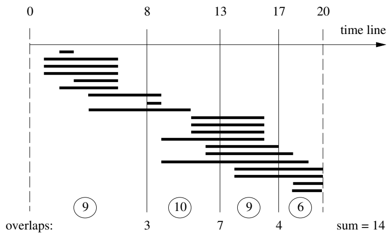
Appendix: An Example Scenario



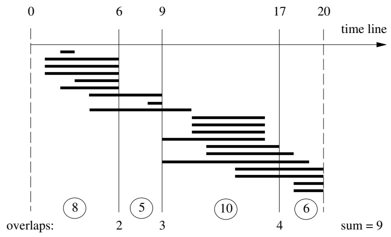
(a) Example Scenario



(b) Uniform



(c) Underflow, $X = 10$



(d) Min. Overlaps, $X = 10$

Fig. 7. Scenario for interval timestamps and the results of 3 partitioning strategies.

j	t_j	$s_R(t_j)$	$o_R(t_j)$	$e_R(t_j)$	$i_R(t_j)$	j	t_j	$s_R(t_j)$	$o_R(t_j)$	$e_R(t_j)$	$i_R(t_j)$
1	**1**	**3**	**3**	0	3	9	**12**	**2**	**7**	0	7
2	**2**	**2**	**5**	0	5	10	**14**	**2**	**9**	0	9
3	**3**	**1**	**5**	1	6	11	**16**	**0**	**5**	4	9
4	**4**	**2**	**7**	0	7	12	**17**	**0**	**4**	1	5
5	**6**	**0**	**2**	5	7	13	**18**	**2**	**5**	1	6
6	**8**	**1**	**3**	0	3	14	**19**	**0**	**4**	1	5
7	**9**	**2**	**3**	2	5	15	**20**	**0**	**0**	4	4
8	**11**	**3**	**5**	1	6						

Fig. 8. The IP-table (in bold) for the intervals in fig. 7 plus $e_R(t_j)$ and $i_R(t_j)$.