Optimal Interval Partitioning for Temporal Databases

Thomas Zurek

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

Abstract

We analyse the problem of partitioning a collection of intervals into two or more subcollections, each of them holding intervals that intersect with a certain range of the intervals' domain.

Intervals that intersect with more than one range impose a considerable overhead on processing and indexing interval data. The goal of interval partitioning (IP) is to find partitions that minimise the number of overlapping intervals. We describe an algorithm for finding optimal partitions and prove its correctness. An example scenario that is used throughout the paper gives an idea of the benefits of optimal interval partitioning.
Keywords: *interval partitioning, temporal databases, parallel temporal joins, temporal index structures*

1 Introduction

Intervals are an increasingly important data type for database management systems. Intervals are used in spatial and geographical, as well as in temporal and historical applications. Several extensions to the relational model have been proposed to incorporate interval functionality. Prominent examples are IXSQL [6] and TSQL2 [8]. Under the name *period*, they form a basic data type in the oncoming SQL3 standard [1].

Interval data, however, impose several problems on conventional (relational) query processing techniques. One feature that is affected is partitioning data over interval values. **Data partitioning** plays an important role in techniques such as physical data partitioning, parallel query processing, hashing and index tree structures. These techniques are used to either (a) concentrate processing on a small and isolated portion data – as in searching – or (b) to break up a big operation into several smaller and independent ones – as in hash joins, parallel joins or parallel I/O.

Partitioning atomic data usually results in a situation in which data objects hold the *same* value on a certain attribute are to be found in the same data fragment. *Equality* of (atomic) values is the determining characteristic in this case because it is the most frequently used predicate in searching, joining and many other relational operations.

In contrast, partitioning of interval data wants to create fragments in which those data objects are concentrated that *intersect* with a certain range of the intervals' domain. *Intersection* has the same significant role for interval data as equality for atomic data. The reason behind this is that intersection of intervals establishes the notion of *simultaneity* (overlapping times) of two or more temporal data objects in the database. Whenever we look for facts that are true at the same time we will have to look for intersecting intervals.

The main problem of interval partitioning (IP), based on intersection, lies in the fact that it creates data fragments that are not disjoint. This can cause a significant overhead in query processing. The choice of an interval partition is therefore much more delicate than for atomic data. And in fact, many authors have already met the IP problem when designing algorithms or index structures for processing interval data. Examples can be found in [5], [2], [3], [4] or [9]. They avoid finding optimal solutions to IP by employing heuristics. In this paper, we want to overcome this deficit by analysing IP in some more detail and providing an algorithm that gives an optimal solution for IP.

The paper is structured as follows. Section 2 gives a motivating example for IP. Section 3 introduces the notation that is being used throughout the paper. Section 4 defines IP formally. Section 5 presents an algorithm for finding an optimal solution for IP. Section 6 presents an example for the computation of an optimal partition. Finally, section 7 summarises and concludes the paper.

2 A Motivating Example

There are several query processing techniques for which partitioning of interval data is relevant. Amongst these are temporal joins. In this case, two tuples are joined if their timestamps – usually represented as an interval – *intersect*. In contrast to this stands the 'traditional' equi-join in which tuples are joined whenever join column attribute values are *equal*.

Hash and parallel join algorithms partition the two participating relations by using the join condition for separating the data objects that cannot possibly join. The algebraic notion behind these approaches is to compute a join between two relations R and S as

$$R \bowtie S \;=\; R_1 \bowtie S_1 \cup \cdots \cup R_n \bowtie S_n \qquad (1)$$

with the R_i and S_i $(i = 1, \ldots n)$ being fragments of R and S respectively such that

$$R = \bigcup_{i=1}^{n} R_i \quad , \quad S = \bigcup_{i=1}^{n} S_i$$

The fragments are not created arbitrarily but on top of the join condition; in an equi-join condition this might be the equality of two strings. In this case, a fragment R_i of R, for example, could hold those tuples whose join attribute value begins with a certain letter or a certain range of letters. The same is done for S_i. This concentrates those tuples of R and S in one location in which they 'meet' their join partners of the other relation. Thus data partitioning has two major advantages

1. it reduces the number of necessary tuple comparisons – a fact that is exploited by hash join algorithms;

2. it breaks one big computation $R \bowtie S$ into several small and *independent* partial computations $R_1 \bowtie S_1, \ldots, R_n \bowtie S_n$ – something that is exploited by parallel join algorithms.

The reader might refer to [7] for a thorough description of the role of partitioning in join processing.

In the case of a temporal join in which the join condition is based on intersecting intervals we can still exploit property (1). But creating the fragments is much more delicate [14]: the fragments cannot be disjoint[1] as in the case of the equi-join. Tuples whose timestamp interval intersects with a certain segment of the time line can still be concentrated in a certain fragment. On the other side, an interval might intersect with more than one time segment and thus has to be present in various fragments. This does not affect the independency advantage in 2. but decreases the benefit of 1.

Figure 1 shows an example of twenty intervals (represented as bold bars) and a uniform partition of the time line that goes from 0 to 20. The breakpoints – 5, 10 and 15 in this case – are shown by vertical lines. The breakpoints themselves belong to the respective left fragment[2]. The figure then gives the resulting 'loads' of the fragments in circles which add up to a total of 39. This means that the partitioning process caused 19 overlaps, i.e. 19 times intervals cross breakpoint lines. This causes an overhead because (a) tuples or intervals are replicated when building the fragments and (b) more tuples have to be compared in total than before for processing the join.

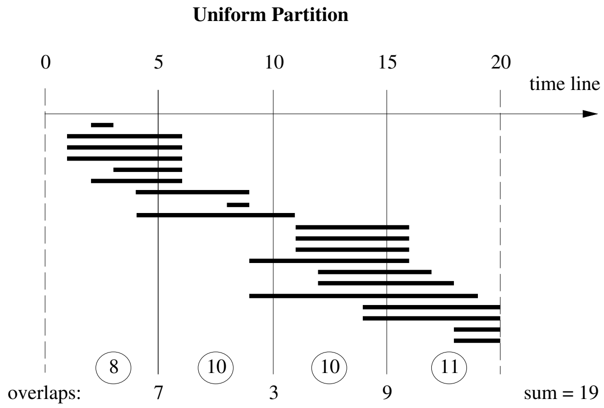

Figure 1: A uniform interval partition for an example scenario.

Imagine the scenario that a relation with 20 tuples, each with the respective interval timestamps as shown in figure 1, is joined with itself (over intersecting timestamps). Computing one big join – this corresponds to the left hand side of (1) – requires every tuple in the left-hand relation to be compared with each tuple of the right hand relation, i.e. $20^2 = 400$ comparisons[3] Computing four partial joins with the fragments shown as in figure 1 causes $8^2 + 10^2 + 10^2 + 11^2 = 385$ comparisons which is not a huge advantage over the one, 'big' operation. However, we can imagine from figure 1

[1] with exception of very rare situations

[2] Therefore, an interval falls into both, the left and the right fragment if it starts at the breakpoint.

[3] We are arguing on a pure conceptual level. Naturally, a sort-merge, hash or index based join algorithm reduces this number as it exploits different ways of data partitioning such as sorting, hashing or precomputed partitions [7] which is actually the point we want to make.

that there is a better way to partition the collection of intervals.

The problem of overlapping intervals not only makes it difficult to choose appropriate breakpoints but also makes it delicate to determine the number of breakpoints: in order to reduce the number of overlaps one has to reduce the number of breakpoints but in order to increase the number n of partial joins in (1) – e.g. to increase the degree of parallelism – one has to increase the number of breakpoints. Thus there are two contrary effects associated with interval partitioning; note that the first one does not exist in the case of an equi-join. This is a manifestation of the *min-max dilemma* [11]. We therefore need an additional input parameter. A frequent constraint is to form fragments with less than a certain number of intervals. Such a number, for example, could be implied by the available amount of memory or disk space.

One heuristical strategy is to create the fragments sequentially from left to right by filling each fragment as much as possible without passing the limiting number of intervals permitted per fragment. [3] called this the *underflow strategy* and used it for balancing a temporal index tree. Figure 2 shows the resulting partition for our example with a maximum of 10 intervals per fragment. It shows a significant improvement in comparison to the uniform partition of figure 1.

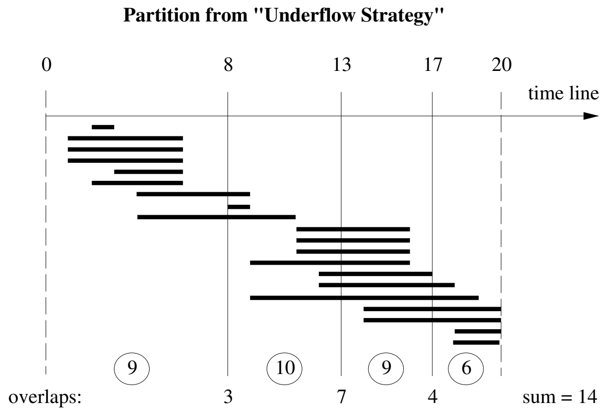

Figure 2: The interval partition using the 'underflow strategy' with a maximum of 10 intervals per fragment.

The remainder of this paper will look at the problem of how to find partition for such a collection of intervals that (a) minimises the number of overlapping intervals and (b) obeys a certain maximum load X for any fragment. Figure 3 shows an optimal partition for the example scenario with $X = 10$. It constitutes a significant improvement over the partitions in figures 1 and 2. After formally defining the interval partitioning problem in section 4 we will describe an algorithm `IP-opt` in section 5 that allows us to compute such optimal partitions. Section 6 shows how the optimal partition in figure 3 is computed by `IP-opt`.

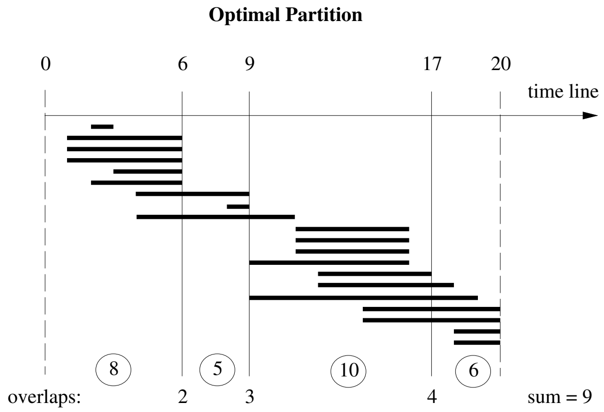

Figure 3: An optimal interval partition for the example scenario with a maximum of 10 intervals per fragment.

3 Notational Conventions

In this section, we define the notation and terminology that is used in the remainder of the paper. we adopt a general view of the interval partitioning problem. Therefore we speak of *collections of intervals* rather than *temporal relations* or any other type of relation containing interval data. A *collection* of intervals is denoted by

$$\langle u_1, \ldots, u_N \rangle$$

The difference between a set and a collection is that an element can appear multiply in a collection but only once in a set. The cardinality is defined accordingly.

Intervals are defined over a certain domain. For our purposes we can assume the set of integers to be the domain with the symbolic infimum and supremum values $-\infty$ and $+\infty$. We assume that there is a total ordering defined on the domain.

Intervals have the following notations:

$$\begin{aligned} [t_s, t_e] &= \{x : t_s \leq x \leq t_e\} \quad \text{and} \\ (t_s, t_e] &= \{x : t_s < x \leq t_e\} \end{aligned}$$

The left delimiter of an interval u is denoted by t_s or $u.t_s$ ('startpoint'), the right one as t_e or $u.t_e$ ('endpoint'). We use the first type in the collection(s) of intervals that are to be partitioned and the second type for the partitioning segments (see below).

The *range* T_U of a collection U of intervals is the part of the domain covered by the intervals in U:

$$T_U = \bigcup_{i=1}^{N} u_i$$

The *span* L_U of a collection U of intervals is defined by

$$L_U = [min\, T_U, max\, T_U]$$

The span may contain parts of the domain that are not covered by any interval and are therefore not included in the range T_U.

The sets of interval startpoints, S_U, and endpoints, E_U, are defined by

$$\begin{aligned} S_U &= \{t_s : \exists t_e \in T_U \text{ such that } [t_s, t_e] \in U\} \\ E_U &= \{t_e : \exists t_s \in T_U \text{ such that } [t_s, t_e] \in U\} \end{aligned}$$

A *partition* P is an ordered set

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

of *breakpoints* that divide the span and range into m segments $(p_{j-1}, p_j]$ for $j = 1, \ldots, m$ with p_0 and p_m being defined appropriately: suitable values are *infimum* T_U or $-\infty$ for p_0 and $max\, T_U$ or $+\infty$ for p_m.

4 Interval Partitioning Problem

We can now define the interval partitioning (IP) problem formally. An instance of IP consists of a collection U of intervals and a limit X for the maximum number of intervals allowed in a partition fragment. The reason why we have to allow collections rather than sets is the following: In a temporal relation, for example, two tuples may be distinct yet but can have the same timestamp interval. In the partitioning process it might be necessary to count every tuple but actually neglect all attribute values apart from the timestamp. Intervals that originate from one or more temporal relations can therefore appear more than once and it is important to know how many times.

The goal is to find a partition $P = \{p_1, \ldots, p_{m-1}\}$ of the span L_U; m can hold any suitable value in this case. The intervals of U are partitioned such that $u \in U$ is put into a fragment U_i iff u intersects with the range $(p_{i-1}, p_i]$ that corresponds to U_i $(i = 1, \ldots, m)$. There are two constraints for P:

1. The total numbers of intervals that overlap the partition points $p_1, \ldots, p_{m-1}$ is minimal.
2. No U_i can hold more than X intervals $(i = 1, \ldots, m)$.

The definition in figure 4 summarises the problem.

<u>Definition</u>: Interval Partitioning – IP

Instance:

- A collection U of intervals $\langle u_1, \ldots, u_N \rangle$,
- a positive number X.

Question:
Is there a partition $P = \{p_1, \ldots, p_{m-1}\}$ of U that minimises

$$\sum_{p \in P} overlaps(p) \qquad (2)$$

such that

$$load(p_{j-1}, p_j) \le X \qquad (3)$$

for $j = 1, \ldots, m$ and where

$$overlaps(p) = |\langle u \in U : u.t_s \le p < u.t_e \rangle|$$

for all $p \in L_U$ and

$$load(p_{j-1}, p_j) = |\langle u \in U : u \cap (p_{j-1}, p_j] \ne \emptyset \rangle|$$

for $j = 1, \ldots, m$.

Figure 4: Defintion of IP.

5 Optimal Partitioning

In this section, we describe how an optimal partition for IP can be found. Firstly, we analyse the space in which to look for the optimal partition (section 5.1). Secondly, we describe the algorithm `IP-opt` that gives the optimal solution if there is any (section 5.2). Finally, the correctness of `IP-opt` is proved in section 5.3.

5.1 Search Space

Looking at the scenario in figure 1, we can see that one major problem of the uniform partition is that its breakpoints go through the interior part of a large number of intervals. The breakpoints can be slightly moved to the left or right to the next point on which one or more 'broken' intervals begin or end. This possibly reduces the number of overlaps and suggests that an optimal partition should have its breakpoints within $S_U \cup E_U$. The remainder of this section shows informally that this is in fact true.

Assume that there is an optimal partition $P_U = \{p_1, \ldots, p_{m-1}\}$ for some instance of IP. Furthermore assume that P_U has a breakpoint p_j that is not a start- or an endpoint of some interval in U, i.e. $p_j \notin S_U \cup E_U$. Let *left*(p_j) and *right*(p_j) be the elements in $S_U \cup E_U$ that are closest to p_j to the left and right respectively, i.e.

$$\begin{aligned} \mathit{left}(p_j) &= \max\{q \in S_U \cup E_U : q < p_j\} \\ \mathit{right}(p_j) &= \min\{q \in S_U \cup E_U : q > p_j\} \end{aligned}$$

Intuitively we can now move p_j to a new breakpoint q which is either $\mathit{left}(p_j)$ or $\mathit{right}(p_j)$ without violating any of the two constraints (2) or (3):

- If $\mathit{left}(p_j)$ and $\mathit{right}(p_j)$ are both startpoints of an interval in U then $q = \mathit{left}(p_j)$. See figure 5.
- Otherwise, either $\mathit{left}(p_j)$ or $\mathit{right}(p_j)$ must be a pure endpoint on which no interval in U starts. Set q to this value. See figure 6.

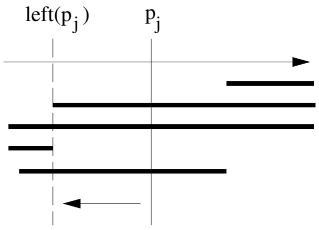

Figure 5: Moving to the nearest startpoint to the left if $\mathit{left}(p_j)$ and $\mathit{right}(p_j)$ are both startpoints.

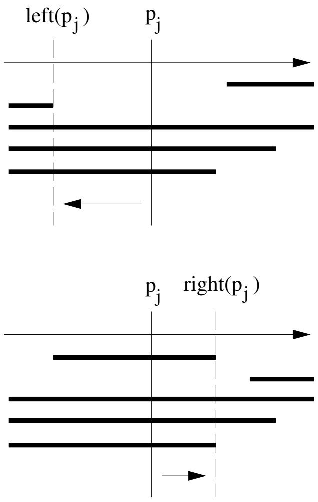

Figure 6: Moving to the nearest pure endpoints if there is such.

Using q instead of p_j will still leave us with an optimal partition because

$$\mathit{overlaps}(q) \leq \mathit{overlaps}(p_j)$$

nor will it violate any of the constraints (3) because

$$\mathit{load}(p_{j-1}, q) \leq \mathit{load}(p_{j-1}, p_j) \quad \text{and}$$

$$\mathit{load}(q, p_{j+1}) \leq \mathit{load}(p_j, p_{j+1})$$

whilst the other segment loads remain unchanged. The reader might refer to figures 5 and 6 as an informal proof as we have to omit the formal one due to space restrictions.

The significance of this observation is that we can now restrict the search for an optimal partition to the set of interval start- and endpoints which is finite:

$$|S_U \cup E_U| \leq 2 \cdot |U| = 2 \cdot N$$

This is especially valuable if the interval domain or the span are infinite. Furthermore it relates the complexity of the problem to the number of intervals and not to the length of the span.

5.2 Algorithm

A dynamic programming approach can be used for computing an optimal solution for IP if there is any. An optimal partition can be found amongst the set of start- and endpoints as we have seen in the previous section. We will refer to the elements in this set as $q_1, \ldots, q_n$ with $q_i < q_{i+1}$ for all $i = 1, \ldots, n-1$.

We first describe `IP-opt` informally. The algorithm starts with q_1 and goes through to q_n. For each q_i it will hold the necessary information for an optimal partition for the span ending at q_i, i.e. for the segment $[\min T_U, q_i]$, with all intervals in U intersecting with the partial span being considered. The algorithm computes two values for each q_i:

- $c(q_i)$ = the minimum number of overlaps for the optimal partition up to q_i,
- $\mathit{pred}(q_i) = q_j$ if q_j is the previous breakpoint that led to this minimum.

A dummy point q_0 is used in order to provide a value for $\mathit{pred}(q_1)$. This is not actually necessary but improves the readability of the algorithm. Finally, the optimal partition – if there is any – is given by the sequence $\mathit{pred}(q_n)$, $\mathit{pred}(\mathit{pred}(q_n))$, ... which ends before q_0 is produced as a breakpoint.

There is no optimal partition if – for any q_i – there is no q_j with $j < i$ such that the number of intervals intersecting with the segment $(q_j, q_i]$, i.e. the number $\mathit{load}(q_j, q_i)$, is less than the maximum load of X intervals. The loads involving the dummy point are defined as $\mathit{load}(q_0, q_i) = \mathit{load}(q_1, q_i)$ for $i = 1, \ldots, n$ and $\mathit{overlaps}(q_0) = 0$. The algorithm then looks like in figure 7.

The run time complexity of the algorithm is $O(n^2)$ as the computation of J and the min-function are $O(n)$ respectively. Computing the values for the load-function is $O(n^2)$ and can be done beforehand. Similarly, computing the values for *overlaps* is $O(n)$.

Algorithm `IP-opt`

- $c(q_0) = 0$
- **for** $i = 1$ **to** n
  - ∘ $J = \{0 \leq j < i : load(q_j, q_i) \leq X\}$
  - ∘ **if** $J = \emptyset$ **then**
    output "No optimal partition."
    stop.
  - ∘ $c(q_i) = \min_{j \in J}\{overlaps(q_j) + c(q_j)\}$
    Let $pred(q_i) = q_j$ for the minimising q_j. If there is more than one qualifying q_j then choose the smallest.

/* output of breakpoints in descending order */

- $p = q_n$
- **while** $p \geq q_1$ **do**
  $p = pred(p)$
  output p.

Figure 7: The algorithm `IP-opt`.

5.3 Correctness

We now demonstrate that `IP-opt` is correct by proving the following

Theorem

The algorithm `IP-opt` *delivers an optimal partition for an instance of IP if there is any partition that satisfies (3).*

Proof

We will prove by induction that

$$c(q_i) \;=\; \sum_{k=1}^{|P_i|} overlaps(pred^k(q_i)) \qquad (4)$$

is the invariant of the for-loop in `IP-opt` where

$$pred^k(q_i) \;=\; \underbrace{pred(pred(\ldots pred}_{k}(q_i)\ldots))$$

and

$$P_i \;=\; \{\, pred^k(q_i) :\; k \geq 1 \;\wedge\; pred^k(q_i) \geq q_1\}$$

Using the facts

- that $c(q_i)$ is always forced to be minimal through the *min* function
- and that P_i is the optimal partition for the segment $[\min T_U, q_i]$

we can conclude that $c(q_n)$ is the number of overlaps in an optimal partition as specified by IP and that an optimal partition is given by P_n.

If there is no partition that satisfies constraint (3) then there must be some q_i such that $load(q_j, q_i) > X$ for all $j < i$. This causes the set J to be empty and thus the algorithm to stop and to mention this fact. In the remainder of the proof we assume that there is an optimal partition and therefore J to be non-empty for all $i = 1, \ldots, n$.

Base case: $i = 1$

The algorithm provides $J = \{0\} \;\Rightarrow\; c(q_1) = \min\{0\} = 0$ $(*)$ with $pred(q_1) = q_0$. Thus $P_1 = \emptyset$ which causes the sum in (4) to evaluate to 0 which matches with $(*)$. Thus the assumption holds.

Hypothesis: Assume that (4) holds for all $i \leq x$ $(**)$.

Inductive Step: $i = x + 1$

Let $q_j = pred(q_i) = pred(q_{x+1})$ be the minimising point with $j < i = x + 1$ as guaranteed by the algorithm. Then

$$\begin{aligned}
c(q_{x+1}) &= overlaps(q_j) + c(q_j) \\
&= overlaps(q_j) + \\
&\quad \sum_{k=1}^{|P_j|} overlaps(pred^k(q_j)) \\
&= overlaps(pred(q_{x+1})) + \\
&\quad \sum_{k=2}^{|P_j|+1} overlaps(pred^k(q_{x+1})) \\
&= \sum_{k=1}^{|P_{x+1}|} overlaps(pred^k(q_{x+1}))
\end{aligned}$$

Therefore (4) holds for $i = x + 1$. □

6 An Example

We now want to give an example for the way in which `IP-opt` computes the optimal partition for the instance of IP as given by the scenario of intervals in figures 1 – 3 and a maximum load of $X = 10$ intervals per fragment. The example scenario comprises $n = 15$ interval start- and endpoints:

$$\{1, 2, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 18, 19, 20\}$$

which we refer to by $q_1, \ldots, q_{15}$. Additionally, $q_0 = 0$ is used as the dummy point. For this set of timepoints, the matrix $load(q_i, q_j)$ with $i, j = 0, \ldots, 15$ is computed according to the formula given in section 4. For the example scenario this matrix is given in figure 8.

Figure 9 describes the steps in which `IP-opt` computes the optimal partition: the first line gives the initial values that are assigned to the dummy point q_0. Each line after that holds the given values i, q_i and $overlaps(q_i)$ and the values for J, $pred(q_i)$ and $c(q_i)$ computed within the **for**-loop body.

Finally, in the output stage, `IP-opt` evaluates the table in figure 9 starting with the last line. The optimal partition is given by the sequence

$$\begin{aligned} pred(20) &= 17 \\ pred^2(20) &= 9 \\ pred^3(20) &= 6 \end{aligned}$$

which is the result that has already been exposed in figure 3.

7 Summary and Conclusions

In this paper, we motivated the importance of interval partitioning in the context of efficient temporal data processing. Simple approaches can lead to a significant overhead in various processing techniques involving parallelism, hashing and balanced tree index structures (sections 1 and 2). The interval partitioning problem was defined formally (section 4). We described an algorithm that provides an optimal partition for an instance of IP (section 5).

Since writing the first version of this paper we have incorporated `IP-opt` into the optimisation of parallel and partitioned sequential joins [13]. The algorithm's run time complexity of $O(n^2)$, however, proved to be a major obstacle for large collections of intervals as encountered in a database context. Nevertheless, we believe that `IP-opt` forms a powerful theoretical base for creating many near-optimal partition strategies such as

- applying `IP-opt` on a smaller set of data, e.g. on a data sample, or
- incrementally adapting an existing optimal partition whenever new data is inserted in the database; this requires to save parts of the algorithm's internal information, such as $c(q_i)$ or $pred(q_i)$, as metadata in the database's catalog.

This latter strategy is particularly important in the context of a data warehouse storing historical data.

References

[1] H. Darwen. Developments in SQL3. personal conversation, Feb. 1997.

[2] R. Elmasri, G. Wuu, and V. Kouramajian. The Time Index and the Monotonic B^+-tree. In Tansel and *et al.* [10], chapter 18, pages 433–456.

[3] H. Gunadhi and A. Segev. Efficient Indexing Methods for Temporal Relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):496–509, June 1993.

[4] C. Kolovson. Indexing Techniques for Historical Databases. In Tansel and *et al.* [10], chapter 17, pages 418–432.

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

[6] N. Lorentzos and Y. Mitsopoulos. SQL Extension for Interval Data. Technical Rep. 105, Informatics Lab., Agricultural University of Athens, 1994.

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

[8] R. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, Sept. 1995.

[9] M. Soo, R. Snodgrass, and C. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proc. of the 10th Int. Conf. on Data Engineering, Houston, Texas, USA*, pages 282–292, Feb. 1994.

[10] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases*. Benjamin/Cummings, 1993.

[11] H. Zhou. Two-stage m-way graph partitioning. *Parallel Computing*, 19(12):1359–1373, Dec. 1993.

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

[13] T. Zurek. Optimisation of Partitioned Temporal Joins. In *Proc. of the 15th BNCOD Conference, London, UK*. Springer, July 1997.

[14] 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).

$$
\begin{pmatrix}
0 & 3 & 5 & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & 3 & 5 & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & & 5 & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & & & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & & & & 7 & 7 & 8 & 10 & 13 & 15 & 17 & 17 & 17 & 19 & 19 & 19 \\
 & & & & & 7 & 8 & 10 & 13 & 15 & 17 & 17 & 17 & 19 & 19 & 19 \\
 & & & & & & 3 & 5 & 8 & 10 & 12 & 12 & 12 & 14 & 14 & 14 \\
 & & & & & & & 5 & 8 & 10 & 12 & 12 & 12 & 14 & 14 & 14 \\
 & & & & & & & & 6 & 8 & 10 & 10 & 10 & 12 & 12 & 12 \\
 & & & & & & & & & 7 & 9 & 9 & 9 & 11 & 11 & 11 \\
 & & & & & & & & & & 9 & 9 & 9 & 11 & 11 & 11 \\
 & & & & & & & & & & & 9 & 9 & 11 & 11 & 11 \\
 & & & & & & & & & & & & 5 & 7 & 7 & 7 \\
 & & & & & & & & & & & & & 6 & 6 & 6 \\
 & & & & & & & & & & & & & & 5 & 5 \\
 & & & & & & & & & & & & & & & 4
\end{pmatrix}
$$

Rows: $i = 0, \ldots, 15$
Columns: $j = 0, \ldots, 15$

Figure 8: Values for $load(q_i, q_j)$ for the scenario of figures 1 – 3.

i	q_i	$overlaps(q_i)$	J	$pred(q_i)$	$c(q_i)$
0	0	0	—	—	0
1	1	3	$\{0\}$	$q_0 = 0$	0
2	2	5	$\{0,1\}$	$q_0 = 0$	0
3	3	5	$\{0,1,2\}$	$q_0 = 0$	0
4	4	7	$\{0,1,2,3\}$	$q_0 = 0$	0
5	6	2	$\{0,1,2,3,4\}$	$q_0 = 0$	0
6	8	3	$\{0,1,2,3,4,5\}$	$q_0 = 0$	0
7	9	3	$\{3,4,5,6\}$	$q_5 = 6$	2
8	11	5	$\{5,6,7\}$	$q_5 = 6$	2
9	12	7	$\{5,6,7,8\}$	$q_5 = 6$	2
10	14	9	$\{7,8,9\}$	$q_7 = 9$	5
11	16	5	$\{7,8,9,10\}$	$q_7 = 9$	5
12	17	4	$\{7,8,9,10,11\}$	$q_7 = 9$	5
13	18	5	$\{11,12\}$	$q_{12} = 17$	9
14	19	4	$\{11,12,13\}$	$q_{12} = 17$	9
15	20	0	$\{11,12,13,14\}$	$q_{12} = 17$	9

Optimal partition $P = \{q_5, q_7, q_{12}\} = \{6, 9, 17\}$

Figure 9: Values computed for the scenario of figures 1 – 3 by `IP-opt` with $X = 10$.