Disjoint interval partitioning

Cafagna, Francesco ; Böhlen, Michael Hanspeter

DOI: https://doi.org/10.1007/s00778-017-0456-7

Noname manuscript No.
(will be inserted by the editor)

Disjoint Interval Partitioning

Francesco Cafagna · Michael H. Böhlen

Received: date / Accepted: date

Abstract In databases with time interval attributes, query processing techniques that are based on sort-merge or sort-aggregate deteriorate. This happens because for intervals no total order exists and either the start or end point is used for the sorting. Doing so leads to inefficient solutions with lots of *unproductive comparisons* that do not produce an output tuple. Even if just one tuple with a long interval is present in the data, the number of unproductive comparisons of sort-merge and sort-aggregate gets quadratic.

In this paper we propose $\mathcal{DIP}$ (Disjoint Interval Partitioning), a technique to efficiently perform sort-based operators on interval data. $\mathcal{DIP}$ divides an input relation into the minimum number of partitions, such that all tuples in a partition are non-overlapping. The absence of overlapping tuples guarantees efficient sort-merge computations without backtracking. With $\mathcal{DIP}$ the number of unproductive comparisons is linear in the number of partitions. In contrast to current solutions with inefficient random accesses to the active tuples, $\mathcal{DIP}$ fetches the tuples in a partition sequentially. We illustrate the generality and efficiency of $\mathcal{DIP}$ by describing and evaluating three basic database operators over interval data: join, anti-join, and aggregation.

Keywords Temporal data · Interval data · Query Processing · Join · Anti-Join · Aggregation

1 Introduction

Many databases model real-world states that change. To model state changes the most common approach is to associate each tuple with a time interval $T = [T_s, T_e)$ that represents the time period during which the tuple is valid [8]. In this paper we propose an efficient technique to perform sort-based computations over temporal relations, i.e., relations with an interval attribute. For example, the temporal relations in Figure 1 record the bookings of luxury suites at hotels $\boldsymbol{R}$ and $\boldsymbol{S}$, where T is the booking period of room # at price $.

$\boldsymbol{R}$	T	#	$
r_1	$[1,5)$	1	80
r_2	$[6,8)$	1	60
r_3	$[7,8)$	2	80
r_4	$[7,10)$	3	75
r_5	$[10,11)$	2	70
r_6	$[10,13)$	5	80

$\boldsymbol{S}$	T	#	$
s_1	$[0,8)$	6	60
s_2	$[1,2)$	2	70
s_3	$[3,4)$	2	80
s_4	$[5,11)$	3	60
s_5	$[9,12)$	2	90
s_6	$[11,12)$	1	90

Fig. 1: Temporal relations $\boldsymbol{R}$ and $\boldsymbol{S}$

Techniques based on sorting have a long tradition in DBMSs and are used extensively by the query evaluation engine. Specifically, sort-merge is used for joins, anti-joins and nearest neighbour joins [6], whereas sort-aggregate is used for aggregations and duplicate elimination [12]. Consider a temporal join where tuples $r_i \in \boldsymbol{R}$ and $s_k \in \boldsymbol{S}$ shall be joined iff their intervals overlap. To ensure that all join matches for an outer tuple r_{i+1} are found, sort-merge must *backtrack* in the inner relation to the first tuple $s_j \in \boldsymbol{S}$ that overlaps with tuple r_i. This is equivalent to the handling of non-key attributes in sort-merge joins [21], but the crucial

F. Cafagna[1,2] · M. H. Böhlen[1]
[1]Department of Computer Science, University of Zürich, Switzerland
[2]AdNovum Informatik, Switzerland
E-mail: {cafagna, boehlen}@ifi.uzh.ch

difference when dealing with T is that the join matches in $\boldsymbol{S}$ for tuple r_{i+1} are *non-consecutive*, and many non-matching tuples might have to be rescanned. Backtracking makes sort-merge inefficient for interval data.

Example 1 To compute a temporal join using sort-merge, $\boldsymbol{R}$ and $\boldsymbol{S}$ are sorted by start point T_s and then processed as illustrated in Figure 2. The middle part illustrates the pairs of tuples that are compared. The numbering illustrates the order in which the comparisons are performed. Thus, we first compare r_1 with s_1. The tuples are joined since $[1, 5)$ overlaps with $[0, 8)$. We proceed with the tuples from $\boldsymbol{S}$ until we fetch a tuple that starts after r_1 ends. Thus, no tuples after s_4 must be looked at. Next, tuple r_2 is fetched and we must backtrack in $\boldsymbol{S}$. To ensure that all join matches for r_2 are found, we must go back to the first join match of r_1 (i.e., s_1), and compare tuple r_2 with s_1, s_2, s_3, s_4, and s_5. Similarly all other tuples in $\boldsymbol{R}$ are processed. Observe that r_2 joins with non-consecutive tuples in $\boldsymbol{S}$: it joins with s_1, does not join with s_2 and s_3, joins with s_4, and does not join with s_5. In total 29 tuple comparisons are done.

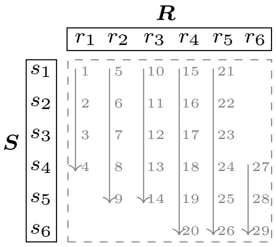


Fig. 2: Temporal join using sort-merge: for $r_{i+1} \in \boldsymbol{R}$, backtracking must go back in $\boldsymbol{S}$ to the first join match of r_i.

A comparison that does not produce a result tuple is an *unproductive* comparison. A sort-merge join may perform many unproductive comparisons due to backtracking. To limit the amount of unproductive comparisons in sort-merge computations, we propose $\mathcal{DIP}$ (*Disjoint Interval Partitioning*). $\mathcal{DIP}$ partitions an input relation into the smallest possible number of partitions, each storing tuples with non-overlapping time intervals. Figure 3 shows the result of $\mathcal{DIP}$ applied to our example relations. The partitioning yields three outer and two inner $\mathcal{DIP}$ partitions. Note that tuples of different partitions may overlap, but inside a single partition tuples do not overlap. Thus, a subsequent merge that does a coordinated scan of partitions to determine the overlapping tuples, does not have to backtrack. Moreover, since $\mathcal{DIP}$ produces partitions with tuples that are sorted, no additional sorting is required prior to computing a merge.

$\boldsymbol{R}_1$	T	#	$
r_1	$[1, 5)$	1	80
r_2	$[6, 8)$	1	60
r_6	$[10, 13)$	5	80

$\boldsymbol{S}_1$	T	#	$
s_1	$[0, 8)$	6	60
s_5	$[9, 12)$	2	90

$\boldsymbol{R}_2$	T	#	$
r_3	$[7, 8)$	2	80
r_5	$[10, 11)$	2	70

$\boldsymbol{S}_2$	T	#	$
s_2	$[1, 2)$	2	70
s_3	$[3, 4)$	2	80
s_4	$[5, 11)$	3	60
s_6	$[11, 12)$	1	90

$\boldsymbol{R}_3$	T	#	$
r_4	$[7, 10)$	3	75

Fig. 3: *Create*$\mathcal{DIP}(\boldsymbol{R})$ and *Create*$\mathcal{DIP}(\boldsymbol{S})$

Example 2 Figure 4 illustrates the computation of the temporal join over $\mathcal{DIP}$ partitions. Two merge steps are computed. First, all partitions of $\boldsymbol{R}$ are joined with $\boldsymbol{S}_1$ (comparison 1 to 9), and then all partitions of $\boldsymbol{R}$ are joined with with $\boldsymbol{S}_2$ (comparison 10 to 22). During a merge step each input partition is scanned just once. For example, for joining the $\boldsymbol{R}$ partitions with partition $\boldsymbol{S}_1$, tuple r_1 is compared with s_1, and, since the tuples overlap, a join match is produced. Since r_1 ends before s_1, we advance in $\boldsymbol{R}_1$ and fetch r_2 producing a second join match. Tuple r_6 is fetched next and compared to s_1 without producing a join match. Since r_6 ends after s_1 we are sure that in $\boldsymbol{R}_1$ we have found all tuples overlapping s_1. We therefore switch to partition $\boldsymbol{R}_2$ (and later to $\boldsymbol{R}_3$), which is processed similarly. After the tuples overlapping s_1 have been found in all outer partitions, we fetch s_5 from $\boldsymbol{S}_1$ and resume the scan of $\boldsymbol{R}_1$ from where it stopped (i.e., r_6): no backtracking is necessary. The middle part of Figure 4 illustrates that with $\mathcal{DIP}$ partitions the number of comparisons is much less than the number of comparisons in Figure 2 where no $\mathcal{DIP}$ partitions are used.

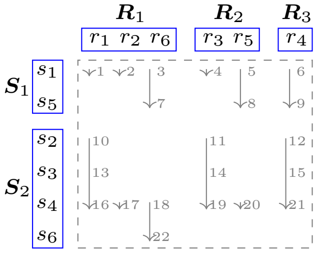


Fig. 4: A temporal join between $\mathcal{DIP}$ partitions is performed without backtracking.

$\mathcal{DIP}$ guarantees that the number of unproductive comparisons is upper-bounded by $c \times n$ where c is the number of partitions and n is the number of tuples. The number of partitions is the maximum number of tuples in a relation that overlap *at a common time.* While backtracking makes sort-merge quadratic as soon as *one* long-lived tuple exists in a relation, $\mathcal{DIP}$ gets quadratic only if there exists a time point that is included in the intervals of *all* tuples in a relation.

Existing partitioning techniques segment the time domain and place the tuples into segments they overlap [7]. Various research questions have been tackled in this context. Among others, disjoint segments [27], overlapping segments [10], variable-size segments [10], and the replications of tuples in all segments they overlap [20] have been investigated. In all cases the (implicit) goal has been to place tuples with similar intervals into the same partitions. $\mathcal{DIP}$ does exactly the opposite: it puts tuples that do not overlap into the same partition. This yields more joins between partitions, but the joins no longer require a nested-loop and are performed much more efficiently: in $O(n)$ rather than $O(n^2)$ time.

Our approach is general, simple and systematic: to compute a temporal join, anti-join, or aggregation, we first compute $\mathcal{DIP}$ on the input relations, and then apply a sequence of merges on the partitions. In our experiments we show that $\mathcal{DIP}$, despite its generality, manages data histories much more efficiently than the more specialized state-of-the-art solutions. The number of partitions is independent of the length of the history, and there is only a linear dependency between the runtime and the size of partitions. Furthermore, we show that current solutions such as the Timeline Index [16] or the Sweepline [3] algorithm, although incur less unproductive comparisons, are slower than $\mathcal{DIP}$ since they suffer from random (disk or memory) accesses: the Timeline Index since it does one index look-up for each matching tuple; Sweepline since after a series of insertions into and deletions from the list of active tuples, the active tuples are scattered in memory [24].

Our technical contributions are as follows:

1. We propose the *Create*$\mathcal{DIP}(\boldsymbol{R})$ algorithm to efficiently partition a relation $\boldsymbol{R}$ into the minimum number of $\mathcal{DIP}$ partitions with non-overlapping tuples.
2. We introduce reduction rules to compute joins, anti-joins, and aggregations over temporal relations using $\mathcal{DIP}$ partitions. We prove that the number of unproductive comparisons per tuple is upper-bounded by the number of $\mathcal{DIP}$ partitions for any of those operators.
3. We introduce an efficient algorithm, $\mathcal{DIP}$*Merge*, to efficiently compute a temporal join, anti-join, and aggregation over multiple $\mathcal{DIP}$ partitions with one sequential scan of the input partitions and no backtracking.
4. We experimentally show that $\mathcal{DIP}$ is the only technique that, either with disk- or memory-resident data, computes temporal joins, anti-joins, and aggregations without deteriorating if the data history grows.

The paper is organized as follows. Section 2 discusses related work. After the background in Section 3, we present Disjoint Interval Partitioning ($\mathcal{DIP}$) in Section 4 and its implementation in Section 5. Section 6 quantifies the costs for, respectively, a temporal join, anti-join, and aggregation using $\mathcal{DIP}$. Section 7 describes the implementation of $\mathcal{DIP}$*Merge*. Section 8 reports the results of our empirical evaluation. Section 9 draws conclusions and points to future work.

2 Related Work

We discuss related works based on the class of problems they solve: first we describe general approaches that cover temporal joins [27, 25] as well as temporal aggregations [4, 30]; next we describe solutions for temporal joins; finally we conclude with solutions for temporal aggregations. Temporal anti-joins have received very little attention: only temporal alignment [9] offers a solution for computing them.

General solutions: Dignös et al. [9] proposed an approach that computes temporal operators by first producing all adjusted time intervals that appear in the result (through a normalization or alignment operation [2]), and then applies the corresponding non-temporal operator to the relations with adjusted time intervals. The interval adjustment is computed with a left outer join on T with inequality conditions on the start and end points of intervals. This is a difficult to optimize primitive and is computed through a nested-loop with a quadratic number of comparisons.

The Timeline Index [16] has been introduced to compute temporal joins and temporal aggregations with the main memory system SAP HANA. The Timeline Index consists of a *Version Map* that stores an Event ID for each T_s and T_e, and of an *Event List* that stores, for each Event ID, the ID of the tuples starting (indicated by 1) and ending (indicated by 0) their validity. Since the index tracks all tuples that are valid at each time point, temporal queries can be implemented by scanning Event List and Version Map concurrently. Temporal aggregates are computed cumulatively while scanning the index. For COUNT, the index is scanned and, for each interval delimited by two timestamps, the count is incremented or decremented according to the number of 0s and 1s. For SUM and AVG each timestamp requires a look-up to fetch the value of the tuple(s) originating or ending to incrementally update the aggregate value. For MIN or MAX, while scanning the index, a list of the Top-K Min/Max values is kept (to use in case the current Min/Max value ceases its validity). For each newly fetched tuple, the validity of each of the K tuples must be checked. No solution is given for determining K. Temporal Joins are computed using sort-merge on the indexes. After a joined pair is built, a look-up for each tuple ID is done (implying that, if a tuple is a join match for k tuples in $\boldsymbol{R}$, k look-ups for the same tuple are done). We experimentally show that this method inherits the disadvantages of traditional index joins, i.e., it is only efficient when few index look-ups are done, otherwise it does not scale.

Solutions for joins: Dignös et al. [10] introduced Overlap Interval Partitioning (*OIP*). The approach divides the time domain into k granules, creates partitions with increasing length that span the entire time domain, and puts each tuple into the shortest partition into which the tuple fits. The join is computed by identifying for each outer partition the overlapping inner partitions. Finding the overlapping partitions is very efficient, but a nested-loop is necessary to join partitions with overlapping tuples. This is a performance bottleneck. When joining partitions with short intervals many unproductive comparisons happen since short tuples overlap with only few other tuples. If the length of the data history increases, the number of short partitions increases too, causing a high number of unproductive comparisons.

Enderle et al. [11] proposed the Relational Interval Tree [19] to compute temporal joins. This approach is index-based, similar to the TimeLine Index, but can be applied to joins only. As mentioned above, index-based techniques are good for few look-ups but, even if a single look-up is fast, cannot compete with more advanced techniques for computing joins if the number of index look-ups is high.

A Sweepline algorithm has been proposed by Arge et al. [3]. It sorts the relations by T_s, and, while scanning the relations, keeps a list of the active $\boldsymbol{R}$ (and $\boldsymbol{S}$) tuples. When a new $\boldsymbol{R}$ (or $\boldsymbol{S}$) tuple is fetched, it is compared with all active $\boldsymbol{S}$ (or $\boldsymbol{R}$) tuples. If an active tuple ceases its validity, it is removed from the list. The allocation and deallocation yields a poor memory locality since, after a series of insertions and deletions into the list of active tuples, the elements of the list become scattered in memory [24]. This causes random accesses when traversing the list, which are considerably slower than sequential accesses [28]. Piatov et al. [24] address this drawback by pre-allocating the space for the active tuples and, when an active tuple is removed from the list, the last inserted active tuple is moved to the free place. This requires that all tuples of the relation have the same size, which is not a realistic assumption in the general case.

MapReduce [7] has been used to compute interval joins. The proposed approach partitions the time domain into q segments, and assigns to each reducer $\boldsymbol{R}_i$ all tuples overlapping the i-th segment. Similar to other approaches it uses a nested-loop to join the tuples of two partitions, outputs the joined tuples, and broadcasts the tuples that span multiple segments to the other reducers. A similar approach that is not MapReduce-based has been proposed by Soo et al. [27]. Both approaches do not give an efficient solution for the nested-loop join between partitions.

Solutions for aggregations: In order to incrementally compute temporal aggregates, the Aggregation Tree has been proposed [18]. The approach has two limitations. First, the entire tree must be kept in memory. For a relation $\boldsymbol{R}$, the size of the tree is up to $2n$ (i.e., the number of different values for T_s and T_e). Second, if the input is sorted by T_s (as is often the case for temporal data), the aggregation tree will be unbalanced, and the time to create it is $O(n^2)$. The Balanced Aggregate Tree [23] addresses the unbalancedness of the Aggregation Tree with a red-black tree. Since the tree stores time instants rather than time intervals it cannot be used to compute Min/Max aggregations. Moreover, to determine an aggregate value at a specific point in time, the tree must be scanned from the beginning to the look-up time point. The SB-Tree [31] reduces the number of tree nodes since multiple intervals are stored in each node (like a B-Tree), each with its corresponding aggregate value. All approaches can only be applied to distributive aggregation functions [13] and must duplicate the index for each aggregation function. Our partitioning is run once and also works for non-distributive functions (e.g., standard deviation).

Moon et al. [23] present a scalable algorithm based on buckets. They partition the time domain into q uniform buckets and assign to each bucket every tuple that overlaps. Tuples spanning multiple buckets are split and assigned to each overlapping bucket. Aggregation is applied inside each bucket by using one of the above mentioned algorithms. To reconstruct the tuples that have been split, adjacent result tuples are merged if they have the same aggregation value. This violates change preservation (lineage) [5] because if two adjacent result tuples have the same aggregation value, but originate from different tuples, the result will only include one tuple instead of two.

Sort-aggregate [12] is a common technique to compute non-temporal aggregates based on sorting and is implemented in many commercial DBMSs. It sorts the data by the grouping attributes, and then computes the aggregate over the tuples within the same group (which, after the sorting, are placed next to each other). This approach can also be applied to temporal data (e.g., sorting the relation by T_s), but backtracking is needed to fetch tuples that have been scanned before and are still valid. As for sort-merge, we experimentally show that this approach becomes quadratic as soon as one tuple with a long time interval exists.

3 Preliminaries

3.1 Notation

We assume a relational schema $(T, A_1, \ldots, A_p)$ where $A_1, \ldots, A_p$ are the non-temporal attributes, and $T = [T_s, T_e)$ is an interval attribute with T_s and T_e being, respectively, its inclusive starting and exclusive ending points. $\boldsymbol{R}$ is a relation over schema $(T, A_1, \ldots, A_p)$ with cardinality n. For a tuple $r \in \boldsymbol{R}$ and an attribute A_i, $r.A_i$ denotes the value of A_i. Given tuples r and s, r is *disjoint* from s iff $r.T_e \leq s.T_s \vee r.T_s \geq s.T_e$, otherwise the tuples are *overlapping*. For example, the tuples $([1,3), a)$ and $([2,6), b)$ are overlapping, whereas the tuples $([1,3), a)$ and $([8,9), c)$ are disjoint.

Table 1 summarizes the symbols and notation that we use in this paper. For the number of partitions, we use relation names in subscripts to refer to specific relations. For example, $c_{\boldsymbol{R}}$ denotes the number of $\mathcal{DIP}$ partitions of relation $\boldsymbol{R}$, while $c_{\boldsymbol{S}}$ denotes the number of $\mathcal{DIP}$ partitions of relation $\boldsymbol{S}$.

Symbol	Meaning	Example
$\boldsymbol{R}$	relation	$\boldsymbol{R}, \boldsymbol{S}$
r	tuple	r, s, r_1, s_j
$r.A_i$	attribute value of a tuple	$r.A_i, r.T$
$\boldsymbol{R}_i$	i-th $\mathcal{DIP}$ partition of $\boldsymbol{R}$	$\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{S}_i$
$\mathcal{H}$	partition heap	$\mathcal{H}$
m	# outer part. processed in parallel	m
n	# of tuples of a relation	n
c	# of $\mathcal{DIP}$ partitions of a relation	$c_{\boldsymbol{R}}, c_{\boldsymbol{S}}, c$
b	size of partition in # of blocks	b
B	size of relation in # of blocks	B
$r.X$	lead of tuple r (cf. Definition 2)	$r.X$

Table 1: Notation.

3.2 Temporal Operators

Table 2 lists and defines a temporal join, a temporal anti-join, and a temporal aggregation. As usual, the semantics of a temporal operator are defined in terms of snapshot reducibility [26] and change preservation [9, 5]. Briefly, *snapshot reducibility* ensures that the result of a temporal operator at any time point p is equal to the result of the corresponding non-temporal operator applied to the input tuples that are valid at p. Thus, the time points to be associated with output tuples depend on the semantics of the non-temporal operator. For a join these are the times during which the outer and inner tuples are both valid; for an anti-join these are the times when an outer tuple is valid and no inner tuple is; for an aggregation these are the times when a set of tuples is valid. *Change preservation* ensures that the result of a temporal operator respects lineage. Thus, any change in the input tuples is reflected in the intervals of the output tuples (cf. Section 6.3).

T. Op.	Definition
$\boldsymbol{R} \bowtie_T \boldsymbol{S}$	$\{ <\tau, r.A_1, \ldots, r.A_p, s.B_1, \ldots, s.B_q> \mid$ $r \in \boldsymbol{R} \wedge s \in \boldsymbol{S} \wedge overlap(r,s) \wedge \tau = (r.T \cap s.T) \}$
$\boldsymbol{R} \rhd_T \boldsymbol{S}$	$\{ <\tau, r.A_1, \ldots, r.A_p> \mid$ $r \in \boldsymbol{R} \wedge \tau.T_s \geq r.T_s \wedge \tau.T_e \leq r.T_e \wedge$ $\nexists s \in \boldsymbol{S}(overlap(s,\tau)) \wedge$ $\exists u \in \boldsymbol{S}(\tau.T_s \in \{r.T_s, u.T_e\} \wedge \tau.T_e \in \{r.T_e, u.T_s\}) \}$
$\vartheta^T_{f(A_i)} \boldsymbol{R}$	$\{ <\tau, f(\boldsymbol{R}'.A_i)> \mid$ $r, s \in \boldsymbol{R} \wedge len(\tau) > 0 \wedge$ $\tau.T_s \in \{r.T_s, r.T_e\} \wedge \tau.T_e \in \{s.T_s, s.T_e\} \wedge$ $\forall u \in \boldsymbol{R}(overlap(u,\tau) \leftrightarrow (\tau - u.T = \emptyset \wedge u \in \boldsymbol{R}'))\}$

Table 2: Semantics of Temporal Operators (T. Op.).

The result of the operators applied to our running example are shown in Figure 10 for a join, Figure 12 for an anti-join, and Figure 16 for an aggregation, and will be explained in Section 6.

3.3 Backtracking over Interval Data

This section shows that the number of unproductive comparisons of sort-merge and sort-aggregate algorithms over relations with overlapping tuples gets quadratic with just one long-lived tuple.

Let L_x be the *longest* interval valid in a relation for time point t_i, and $\boldsymbol{L}$ be the set of the longest intervals in a relation for all possible time points. In Figure 5, for example, the longest interval at time t_p is L_b. $\boldsymbol{L} = \{L_a, L_b, L_c\}$ since between time t_1 and t_j interval L_a is the longest, between t_j and t_k interval L_b is the longest, and between t_k and t_z the longest interval is L_c. We assume that the intervals in $\boldsymbol{L}$ do not overlap and at least one tuple is valid at each point. This yields a lower bound for the number of comparisons in sort-merge computations, which is sufficient for our analyses. If two longest intervals overlap more comparisons are needed since backtracking must go back further.

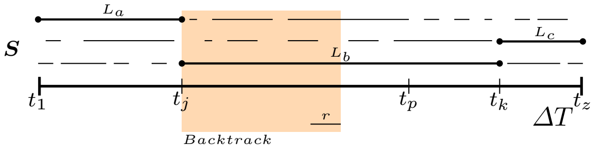


Fig. 5: For $r \in \boldsymbol{R}$, on average, half of the tuples within L_i are compared because of backtracking.

Figure 5 illustrates that for an outer tuple r that overlaps a longest interval L_k, backtracking re-fetches all tuples between $L_k.T_s$ and $r.T_e$. The cost of sort-merge in terms of tuple comparisons is $Comp(\text{Merge}) + n \times Comp(\text{Backtrack})$. Let $\bar{\boldsymbol{L}}$ be the average length of the longest intervals and $|\Delta T|$ the length of the time domain. $Comp(\text{Merge}) = n + n - 1$ is the number of comparisons of the merge procedure since in each step the algorithm advances either the outer or the inner relation. $Comp(\text{Backtrack}) = \frac{\bar{\boldsymbol{L}}}{|\Delta T|} \times n \times \frac{1}{2}$ quantifies the number of inner tuples rescanned on average for an outer tuple r: $\frac{\bar{\boldsymbol{L}}}{|\Delta T|} \times n$ is the number of tuples within L_k, and, since on average r ends in the middle of L_k, half of them are refetched and compared against r.

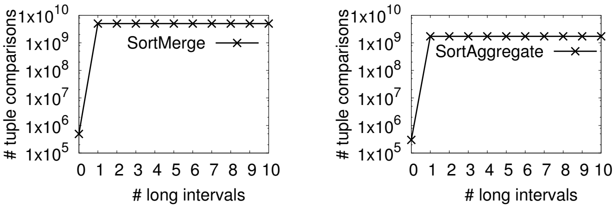
Lemma 1 *(Cost of backtracking) The number of unproductive comparisons for a temporal join $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ using backtracking becomes quadratic as soon as just one long-lived tuple exists.*

Proof If $\boldsymbol{S}$ includes one long-lived tuple with an interval L_x that spans the entire time domain, then $|L_x| = |\Delta T|$, $\boldsymbol{L} = \{L_x\}$, $\frac{\bar{\boldsymbol{L}}}{|\Delta T|} = \frac{|L_x|}{|\Delta T|} = \frac{|\Delta T|}{|\Delta T|} = 1$. The number of unproductive comparisons are obtained by subtracting the cardinality of the result from the cost of the join. In the worst case, each $r \in \boldsymbol{R}$ only overlaps with L_x and no other interval in $\boldsymbol{S}$, and we get:

$$
\begin{aligned}
&Comp(\text{Merge}) + n \times Comp(\text{Backtrack}) - |\boldsymbol{R} \bowtie_T \boldsymbol{S}| \\
&= (n + n - 1) + n(1 \times n \times \tfrac{1}{2}) - n \\
&= O(n^2) \text{ unproductive comparisons.} \qquad \blacksquare
\end{aligned}
$$

This result is experimentally confirmed in Figure 6(a), where a temporal join on the fact table of the Swiss Feed Data Warehouse [29] is computed. We show that as soon as measurements that are time-invariant, and therefore valid over the entire time domain ΔT, are taken into account (e.g., the *Protein Digestibility* value), sort-merge becomes inefficient.

Figure 6(b) illustrates that also sort-aggregate [12], i.e., temporal aggregation computed using sorting, suffers from a quadratic number of unproductive comparisons. For non-temporal data sort-aggregate makes only one scan to compute the aggregate because, after the sorting, all tuples of the same group are consecutive. When dealing with time intervals, instead, sort-aggregate must backtrack to fetch tuples that have been scanned before but are still valid.



(a) Sort-merge (b) Sort-aggregate

Fig. 6: Sort-merge and sort-aggregate deteriorate to a nested-loop as soon as a single long-lived tuple exists.

4 Disjoint Interval Partitioning

Definition 1 *($\mathcal{DIP}$ partition).* Consider a relation $\boldsymbol{R}$ with schema $(T, A_1, \ldots, A_p)$. A $\mathcal{DIP}$ partition $\boldsymbol{R}_i \subseteq \boldsymbol{R}$ is a subset of $\boldsymbol{R}$ such that:

$$\forall (r, s) \in \boldsymbol{R}_i \big(r \neq s \Rightarrow disjoint(r, s)\big)$$

Thus, a $\mathcal{DIP}$ partition $\boldsymbol{R}_i$ is a set of non-overlapping tuples from $\boldsymbol{R}$. In Figure 3 we have three outer $\mathcal{DIP}$ partitions ($\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3$) and two inner $\mathcal{DIP}$ partitions ($\boldsymbol{S}_1, \boldsymbol{S}_2$). Tuples of different partitions may overlap, but within a single partition all tuples are disjoint.

4.1 Efficient Merging of $\mathcal{DIP}$ Partitions

We use $\mathcal{DIP}$ to speed up the *merge* in sort-merge computations. The advantage of $\mathcal{DIP}$ is that a temporal operator can be computed between two $\mathcal{DIP}$ partitions using a merge procedure that does not have to backtrack, i.e., does one scan of the partitions with sequential IOs only.

Lemma 2 *(No Backtracking) Consider two $\mathcal{DIP}$ partitions $\boldsymbol{R}_i$ and $\boldsymbol{S}_j$. During a sort-merge computation no backtracking must be done in $\boldsymbol{S}_j$ to find the tuples that overlap $r \in \boldsymbol{R}_i$.*

Proof Let $r_1, r_2 \in \boldsymbol{R}_i$. Since the partitions are sorted (e.g., by T_s), we assume without loss of generality that r_1 precedes r_2. Tuples r_1 and r_2 are disjoint since they are in the same $\mathcal{DIP}$ partition. Since all tuples in $\boldsymbol{S}_j$ are disjoint, at most one tuple s_k may exist that overlaps r_1 such that $s_k.T_e > r_1.T_e$ (cf. Figure 7). Thus, all tuples that precede s_k must end before r_2 starts and therefore cannot overlap r_2. Tuple s_k is the only tuple that can overlap both with r_1 and with r_2. $\blacksquare$

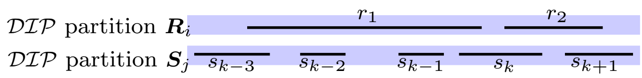

$\mathcal{DIP}$ partition $\boldsymbol{R}_i$
r_1
r_2
$\mathcal{DIP}$ partition $\boldsymbol{S}_j$
s_{k-3}
s_{k-2}
s_{k-1}
s_k
s_{k+1}

Fig. 7: Illustration of Lemma 2.

Lemma 2 guarantees that, if s_k is the last tuple that overlaps a tuple in $\boldsymbol{R}_i$, there is no need to rescan any tuple before s_k to find the matches for the next tuple in $\boldsymbol{R}_i$. This means that a merge procedure between $\mathcal{DIP}$ partitions can be computed without backtracking, i.e., with just one scan of the input partitions.

Figure 8 illustrates that Lemma 2 also holds when multiple outer partitions are merged simultaneously with $\boldsymbol{S}_j$. The scan of an outer partition (e.g., $\boldsymbol{R}_1$) proceeds until all tuples overlapping s_k have been found. Since in $\boldsymbol{R}_1$ only the last scanned tuple may overlap with s_{k+1}, we mark its position in $\boldsymbol{R}_1$, and process $\boldsymbol{R}_2$ (and later $\boldsymbol{R}_3$) to find the other join matches of s_k. After all join matches have been found, s_{k+1} is fetched and the scan of the outer partitions resumes from the previously marked positions. Of the tuples previously accessed in $\boldsymbol{R}_1, \boldsymbol{R}_2$ and $\boldsymbol{R}_3$, only the marked ones (i.e., the bold-faced ones in Figure 8) may overlap with s_{k+1}.

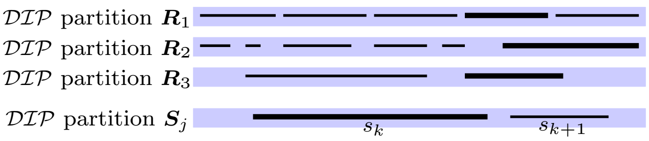

$\mathcal{DIP}$ partition $\boldsymbol{R}_1$
$\mathcal{DIP}$ partition $\boldsymbol{R}_2$
$\mathcal{DIP}$ partition $\boldsymbol{R}_3$
$\mathcal{DIP}$ partition $\boldsymbol{S}_j$
s_k
s_{k+1}

Fig. 8: Efficient merging without backtracking: in each outer partition, no tuple before the one (shown in bold) scanned last for s_k can overlap s_{k+1}.

5 Efficient Data Partitioning

We use $\mathcal{DIP}$ as the essential first step to efficiently compute temporal operators. Since for such computations the number of unproductive comparisons is limited by the number of $\mathcal{DIP}$ partitions, we first provide the *Create*$\mathcal{DIP}$ algorithm to partition the input relation into the *minimum* number of $\mathcal{DIP}$ partitions.

5.1 The Partitioning Algorithm *Create*$\mathcal{DIP}$

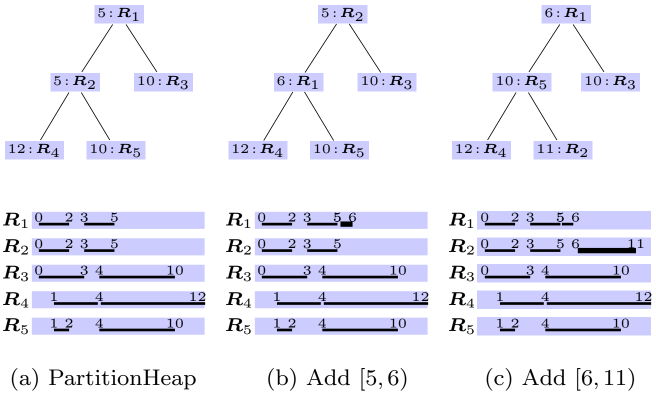
Algorithm *Create*$\mathcal{DIP}(\boldsymbol{R})$ sorts the input relation $\boldsymbol{R}$ by T_s. It then scans the data and places tuples into partitions where they do not overlap another tuple. In order to determine if tuple t overlaps with a tuple in $\mathcal{DIP}$ partition $\boldsymbol{R}_i$, the following Lemma asserts that it is enough to compare t with only one tuple, namely the last tuple inserted into $\boldsymbol{R}_i$.

Lemma 3 *(Transitivity) Consider tuples $r, s \in \boldsymbol{R}_i$ and a new tuple t such that $r.T_s \leq s.T_s \leq t.T_s$. Then $disjoint(r, s) \wedge disjoint(s, t) \iff disjoint(r, t)$.*

Proof The end point of an interval is always larger than the start point: $s.T_e > s.T_s$. Since the tuples in a $\mathcal{DIP}$ partition do not overlap, we have $r.T_e < s.T_s$. Since $s.T_s \leq t.T_s$ (recall that we process tuples ordered by T_s), $r.T_e \leq t.T_s$ follows. Thus, r does not overlap t. ■

We use Lemma 3 to efficiently determine if a tuple can be placed in a partition. This is the case if a tuple does not overlap with the last element of a $\mathcal{DIP}$ partition. We store the partitions in a min-heap $\mathcal{H}$ [17, p. 58]. Each partition is represented by a node whose key is the T_e value of the last tuple inserted into the partition and whose value is a pointer to the partition. The root points to the partition whose last inserted tuple ends the earliest among all partitions. Thus, a new tuple r can either be placed in the root partition if it does not overlap with its last element or we know for sure that r overlaps with all partitions and a new one must be created.

Example 3 Consider a relation sorted by T_s whose first ten tuples have been partitioned as illustrated in Figure 9(a). The next tuple r has $r.T = [5, 6)$. We compare $[5, 6)$ with the key of the root: since the starting point of $[5, 6)$ is larger or equal than 5 (i.e., $[5, 6)$ does not overlap with $[3, 5)$) we add $[5, 6)$ to the root partition and reorganize the heap by swapping the root with its left child. Figure 9(b) illustrates the result. The next tuple r has $r.T = [6, 11)$: since its starting point is greater or equal than 6, r is inserted into the root partition $\boldsymbol{R}_2$.


5 : $\boldsymbol{R}_1$
5 : $\boldsymbol{R}_2$
10 : $\boldsymbol{R}_3$
12 : $\boldsymbol{R}_4$
10 : $\boldsymbol{R}_5$
$\boldsymbol{R}_1$ 0 2 3 5
$\boldsymbol{R}_2$ 0 2 3 5
$\boldsymbol{R}_3$ 0 3 4 10
$\boldsymbol{R}_4$ 1 4 12
$\boldsymbol{R}_5$ 1 2 4 10
5 : $\boldsymbol{R}_2$
6 : $\boldsymbol{R}_1$
10 : $\boldsymbol{R}_3$
12 : $\boldsymbol{R}_4$
10 : $\boldsymbol{R}_5$
$\boldsymbol{R}_1$ 0 2 3 5 6
$\boldsymbol{R}_2$ 0 2 3 5
$\boldsymbol{R}_3$ 0 3 4 10
$\boldsymbol{R}_4$ 1 4 12
$\boldsymbol{R}_5$ 1 2 4 10
6 : $\boldsymbol{R}_1$
10 : $\boldsymbol{R}_5$
10 : $\boldsymbol{R}_3$
12 : $\boldsymbol{R}_4$
11 : $\boldsymbol{R}_2$
$\boldsymbol{R}_1$ 0 2 3 5 6
$\boldsymbol{R}_2$ 0 2 3 5 6 11
$\boldsymbol{R}_3$ 0 3 4 10
$\boldsymbol{R}_4$ 1 4 12
$\boldsymbol{R}_5$ 1 2 4 10

(a) PartitionHeap (b) Add $[5, 6)$ (c) Add $[6, 11)$

Fig. 9: Partition Heaps after inserting [5,6) and [6,11).

Figure 9(c) illustrates the result after reorganizing the heap.

Algorithm 1 describes the details of our implementation. Tuple r is the next tuple to process. If r overlaps with the last tuple in the root partition (line 4), we create a new partition at the end of the heap (line 5), put r in it, and call *HeapifyUp* to reorganize the heap. If r does not overlap with the last tuple of the root partition, we add r to it (line 9) and call *HeapifyDown* to reorganize the heap. When all tuples have been processed, the algorithm returns the partition heap $\mathcal{H}$.

Algorithm 1: $\mathit{Create}\mathcal{DIP}(\boldsymbol{R})$

```
1  Sort($\boldsymbol{R}$) by $T_s$ ;
2  $\mathcal{H}.root \leftarrow \varnothing$ ;
3  while $(r = fetchtuple(\boldsymbol{R})) \neq null$ do
4      if $overlap(\mathcal{H}.root.last, r)$ then
5          $\mathcal{H}.new \leftarrow growHeap(\mathcal{H})$ ;
6          Add $r$ to $\mathcal{H}.new$ ;
7          $HeapifyUp(\mathcal{H}.new)$ ;
8      else
9          Add $r$ to $\mathcal{H}.root$ ;
10         $HeapifyDown(\mathcal{H}.root)$ ;
11 return $\mathcal{H}$;
```

Create$\mathcal{DIP}$ implements the solution to the interval partitioning problem [17, p.116] (aka interval-graph coloring problem). The algorithm is correct since all intervals are assigned to a partition: if a tuple cannot be placed into an existing partition without overlapping another tuple a new partition is created. The set of partitions is minimal since the number of partitions c is equivalent to the *depth* of the set of intervals, i.e., the maximum number of intervals that overlap a common time point.

For data with a long history, i.e., data collected over many years, the number of partitions c is small compared to the size of the relation. For example, in a database storing the bookings of a hotel, c is equal to the number of rooms (e.g., in the worst case all rooms are occupied on a given day), which is smaller than all bookings recorded since the beginning. In data collected in a network of sensors, c is the number of sensors (e.g., all sensors record a value at the same time), which is smaller than the number of observations collected through the sensors over the years.

5.2 Properties of $\mathcal{DIP}$ Partitioning

Lemma 4 *The runtime complexity for computing Create$\mathcal{DIP}$ on a relation with n tuples is upper-bounded by $O(n \log n)$.*

Proof The cost of our partitioning is given by the sum of: *i)* the cost of sorting, i.e., $O(n \log n)$, and *ii)* the cost of the algorithm itself, i.e., $O(n \log c)$ since, for each tuple, in the worst case each call of *HeapifyUp* or *HeapifyDown* propagates a node from the root to a leaf or vice versa (with cost $\log c$). Since $c \leq n$ we get $O(n \log n + n \log c) = O(n \log n)$. ■

During the second phase of sort-merge computations the sorted $\mathcal{DIP}$ partitions are merged. A property of $\mathcal{DIP}$ is that the number of comparisons of the merge step is guaranteed to be independent of the size of the $\mathcal{DIP}$ partitions. Consider a relation with cardinality n that is partitioned into c $\mathcal{DIP}$ partitions: our approach always makes the same number of comparisons, independent of how many tuples are placed in the partitions.

Example 4 Let $\boldsymbol{R} = \{\boldsymbol{R}_1, \boldsymbol{R}_2\}$ and $\boldsymbol{S} = \{\boldsymbol{S}_1, \boldsymbol{S}_2\}$ be two relations with $10k$ tuples each. $\mathcal{DIP}$ first joins $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ with $\boldsymbol{S}_1$ and then with $\boldsymbol{S}_2$. Since partitions are totally ordered, a join between the partitions can be done with a scan. Thus, we are guaranteed to have at most $(|\boldsymbol{R}_1|+|\boldsymbol{R}_2|+2|\boldsymbol{S}_1|)+(|\boldsymbol{R}_1|+|\boldsymbol{R}_2|+2|\boldsymbol{S}_2|) = 2|\boldsymbol{R}|+2|\boldsymbol{S}| = 40k$ comparisons in total, which is independent of the size of the partitions.

With current partitioning approaches, instead, the cost of a join is not known a priori since it does not only depend on the number of partitions, but also on the number of tuples stored in each partition. Since partitions are not totally ordered each join must be implemented with a nested loop. Let $\boldsymbol{R} = \{\boldsymbol{R}'_1, \boldsymbol{R}'_2\}$ and $\boldsymbol{S} = \{\boldsymbol{S}'_1, \boldsymbol{S}'_2\}$ be two relations each split into two partitions, such that $\boldsymbol{R}'_1$ must be joined with $\boldsymbol{S}'_1$ only, and $\boldsymbol{R}'_2$ must be joined with $\boldsymbol{S}'_2$ only. Thus, if each relation has $10k$ tuples and the partitions have size $5k$ each, then the two nested-loops perform $|\boldsymbol{R}'_1| \times |\boldsymbol{S}'_1| + |\boldsymbol{R}'_2| \times |\boldsymbol{S}'_2| = 5k \times 5k + 5k \times 5k = 50M$ comparisons. If the partitions have sizes $|\boldsymbol{R}'_1| = |\boldsymbol{S}'_1| = 9k$ and $|\boldsymbol{R}'_2| = |\boldsymbol{S}'_2| = 1k$, then the nested loop joins perform $9k \times 9k + 1k \times 1k = 82M$ comparisons.

6 Temporal Operators

Our approach reduces a temporal operator O_T over an entire relation to a sequence of $\mathcal{DIP}$ operators $O_T^{\mathcal{DIP}}$, i.e., temporal operators over $\mathcal{DIP}$ partitions. We show how to compute temporal joins ($\bowtie_T$), anti-joins ($\rhd_T$), and aggregations (ϑ_F^T), by reducing these operators to, respectively, $\mathcal{DIP}$ joins ($\bowtie_T^{\mathcal{DIP}}$), $\mathcal{DIP}$ anti-joins ($\rhd_T^{\mathcal{DIP}}$) and $\mathcal{DIP}$ full outer joins ($⟗_T^{\mathcal{DIP}}$).

6.1 Temporal Join

A temporal join $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ returns the pairs (r, s), with $r \in \boldsymbol{R}$ and $s \in \boldsymbol{S}$, whose time interval T overlaps. Figure 10 illustrates the join result of our example relations. It computes the price difference between the luxury suites of hotel $\boldsymbol{R}$ and those of hotel $\boldsymbol{S}$. For example, the second output row says that suite #1 of hotel $\boldsymbol{R}$ and suite #2 of hotel $\boldsymbol{S}$ have a price difference of 10$ during time $[1, 2)$, while the third row says that, at time $[3, 4)$, they cost the same.

$\pi_{T,\boldsymbol{R}.\#,\boldsymbol{S}.\#,\boldsymbol{R}.\$-\boldsymbol{S}.\$}(\boldsymbol{R} \bowtie_T \boldsymbol{S})$

T	$\boldsymbol{R}.\#$	$\boldsymbol{S}.\#$	$\boldsymbol{R}.\$ - \boldsymbol{S}.\$$
$[1, 5)$	1	6	20
$[1, 2)$	1	2	10
$[3, 4)$	1	2	0
..	..	..	..
$[10, 11)$	5	3	20
$[10, 12)$	5	2	-10
$[10, 12)$	5	1	-10

Fig. 10: Temporal join applied to the running example.

To compute a temporal join using $\mathcal{DIP}$, sets of m outer partitions (e.g., $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$) are joined with each inner partition until all outer partitions have been processed:

$$\boldsymbol{R} \bowtie_T \boldsymbol{S} \iff \bigcup_{i=1}^{c_{\boldsymbol{R}}/m} \bigcup_{j=1}^{c_{\boldsymbol{S}}} \left(\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j\right) \quad (1)$$

Thus, to compute a temporal join, we compute $\frac{c_{\boldsymbol{R}} \times c_{\boldsymbol{S}}}{m}$ $\mathcal{DIP}$ joins. Each $\mathcal{DIP}$ join joins m outer partitions with each inner partition: first $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ are joined with $\boldsymbol{S}_1$, then with $\boldsymbol{S}_2$, etc.

Figure 11 illustrates the differences between $\mathcal{DIP}$ and other approaches on our running example. The thickness of the arrows is proportional to the cost of each join in terms of number of comparisons. With $\mathcal{DIP}$ many outer partitions can be processed simultaneously. Furthermore, even if the total number of merges between partitions might be higher for $\mathcal{DIP}$, the cost of each $\mathcal{DIP}$ join is small compared to the cost of the other approaches since it requires only one scan of the input partitions (it is computed in linear rather than quadratic time).

Example 5 We use Equation (1) to compute a temporal join between the relations of our running example, i.e., $\boldsymbol{R}$ with $c_{\boldsymbol{R}} = 3$ and $\boldsymbol{S}$ with $c_{\boldsymbol{S}} = 2$. Figure 4 illustrates the process for computing $\big(\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_1\big) \bigcup \big(\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_2\big)$. Clearly, the join of all $\mathcal{DIP}$ partitions is done with much less unproductive comparisons than the sort-merge join in Figure 2.

Equation (1) shows that the higher m, the fewer $\mathcal{DIP}$ joins are computed. The value of m is given by the number of partitions that can be processed simultaneously. In typical commercial operating systems this is about 10^4 (the number of files a process is allowed to keep open at a time). We will show that when *all* tuples overlap and n partitions are created, m is the factor by which we reduce the quadratic worst case I/O cost for computing a temporal join, which is significant.

6.1.1 CPU Cost

We quantify the CPU overhead in terms of *unproductive comparisons*, i.e., the number of tuple comparisons that do not produce an output tuple. We determine an upper-bound for the number of unproductive comparisons. For simplicity, we use c to indicate both the number of partitions of $\boldsymbol{R}$ and those of $\boldsymbol{S}$, and $\frac{n}{c}$ to indicate the number of tuples of a partition.

Lemma 5 *Consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$ that have been partitioned into c $\mathcal{DIP}$ partitions each. The number of unproductive comparisons for computing $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ using $\mathcal{DIP}$ is upper-bounded by $c \times n$.*

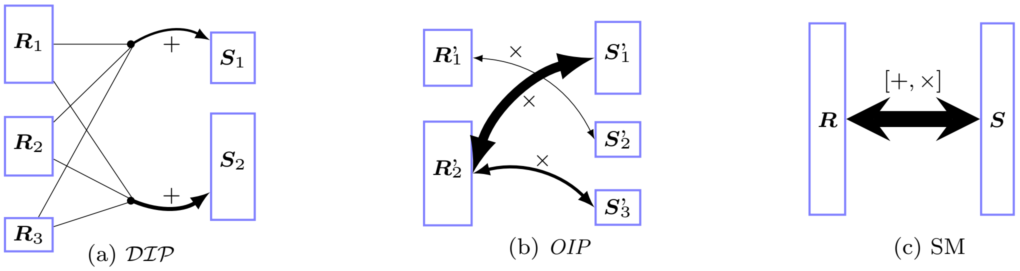
Proof From Equation (1) we get

$$\text{CPU}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) = \text{CPU}\Big(\bigcup_{i=1}^{c/m} \bigcup_{j=1}^{c} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j)\Big)$$

A $\mathcal{DIP}$ join $\bowtie_T^{\mathcal{DIP}}$ is implemented as a merge of $\mathcal{DIP}$ partitions without backtracking, i.e., a procedure that in each iteration either advances one (outer or inner) tuple or switches the current outer partition. Thus, the number of iterations is given by the total size of the m outer partitions, plus $m-1$ partition switches per inner tuple, plus the size of the inner partition. We get:

$$\begin{aligned}\text{CPU}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) &= \sum_{i=1}^{c/m} \sum_{j=1}^{c} (\tfrac{n}{c} * m + (m-1)\tfrac{n}{c} + \tfrac{n}{c}) \\ &= \sum_{i=1}^{c/m} (n * m + n * m) \\ &\approx c * n\end{aligned}$$

The number of unproductive comparisons is $CPU(\boldsymbol{R} \bowtie_T \boldsymbol{S}) - |\boldsymbol{R} \bowtie_T \boldsymbol{S}|$, i.e., the number of comparisons minus the number of result tuples. In the worst case we have 0 result tuples, and we get $c*n$ unproductive comparisons. ■



(a) $\mathcal{DIP}$ (b) *OIP* (c) SM

Fig. 11 $\mathcal{DIP}$ joins with linear cost (indicated by a + sign) many outer partitions with each inner partition. *OIP* joins with quadratic cost (indicated by a × sign) each outer partition with few inner partitions. Sort-merge backtracks without any guarantee about the complexity (i.e., ranges from linear to quadratic).

6.1.2 I/O Cost

This section quantifies the number of block I/Os for computing a temporal join using $\mathcal{DIP}$. We assume that all $\mathcal{DIP}$ partitions are equally sized, and each of them has $b = \frac{B}{c}$ blocks.

Lemma 6 *Consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$ partitioned into c $\mathcal{DIP}$ partitions each. The number of I/Os for computing $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ using $\mathcal{DIP}$ is $min(c, \frac{n}{m}) \times B$.*

Proof From Equation (1) we get:

$$\mathrm{IO}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) =$$
$$\mathrm{IO}\Big(\bigcup_{i=1}^{c/m} \bigcup_{j=1}^{c} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j)\Big)$$

With equally sized partitions we obtain:

$$= \tfrac{c}{m} \times c \times \mathrm{IO}(\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j)$$

Equation (1) shows that for $c_{\boldsymbol{S}}$ subsequent calls of $\mathcal{DIP}$ join only the inner partition changes. Since the outer partitions $\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\}$ are reused, we cache the first M blocks of each $\boldsymbol{R}_i$, and obtain:

$$\begin{aligned} &= \tfrac{c}{m} \times c \times \big((b - M) * m + b\big) \\ &= c^2 \times (b - M) + \tfrac{c^2}{m} \times b \qquad (2) \end{aligned}$$

When dealing with data histories, tuples are valid at different points in time and partitions get large since old tuples do not overlap with recent ones. This means $b \gg M$, and from (2) we get:

$$\begin{aligned} \mathrm{IO}_{General}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) &= c^2 \times b + \tfrac{c^2}{m} \times b \\ &\approx c \times B \qquad (2a) \end{aligned}$$

where $B = c \times b$ are the blocks of an input relation. In other words, in the general case, our approach is linear in the number of partitions: independent of m it fetches each block c times.

The worst case for our approach is when $c \approx n$, i.e., many partitions exist (e.g., most tuples overlap). In such a case the partitions are small since only few non-overlapping tuples can be stored in a partition. With small partitions we have $c \approx n \iff b \leq M$, and from (2) we get:

$$\begin{aligned} \mathrm{IO}_{Worst} &= c^2 \times 0 + \tfrac{c^2}{m} \times b = \tfrac{c}{m} \times B \\ &\approx \tfrac{n}{m} \times B \qquad (2b) \end{aligned}$$

Summarizing:

$$\begin{aligned} \mathrm{IO}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) &= min(\mathrm{IO}_{General}, \mathrm{IO}_{Worst}) \\ &= min(c, \tfrac{n}{m}) \times B \end{aligned}$$ ■

Thus, while in the general case $\mathcal{DIP}$ fetches each block c times, m helps to speed up our worst-case scenario: if m outer partitions are processed simultaneously, we reduce the number of I/Os to perform by a factor of m. This is effective already for small values of m: for example if $m = 10$ we make an order of magnitude less I/Os. In our experiments we will show that, if m is just 0.1% the number of partitions, i.e., 0.1% of the partitions are processed simultaneously, our approach reaches the same performances as state of the art solutions that put overlapping tuples in the same partition [10].

6.2 Temporal Anti-Join

A temporal anti-join $\boldsymbol{R} \rhd_T \boldsymbol{S}$ returns, for each $r \in \boldsymbol{R}$, its maximal sub-intervals (if any) during which no tuple in $\boldsymbol{S}$ exists. Figure 12 illustrates the result of a temporal anti-join $\boldsymbol{R} \rhd_T \boldsymbol{S}$ on our example relations. The anti-join returns the price of the luxury suites of hotel $\boldsymbol{R}$ when no suite has been booked in hotel $\boldsymbol{S}$. The result includes one tuple since $[12, 13)$ is the only time interval during which a tuple in $\boldsymbol{R}$ is valid and no tuple in $\boldsymbol{S}$ exists.

In order to take advantage of Lemma 2, and compute the anti-join without backtracking, we transform the anti-join problem into a problem of finding overlapping intervals. To do so, we do not compare $r \in \boldsymbol{R}_i$ with the time interval of $s \in \boldsymbol{S}$, but with its *lead*.

$\boldsymbol{R} \rhd_T \boldsymbol{S}$

T	#	\$
[12, 13)	5	80

Fig. 12: Temporal anti-join applied to the main example

Definition 2 *(Lead).* Let s be a tuple of a relation $\boldsymbol{S}$; the lead of s, indicated by $s.X = [X_s, X_e)$, is the largest interval (if any) not overlapping any $\boldsymbol{S}$ tuple and such that $s.X_e = s.T_s$.

Example 6 In relation $\boldsymbol{Z}$ of Figure 13, we have $z_1.X = [-\infty, 0)$, $z_2.X = [1, 3)$, and $z_4.X = [10, 11)$. Tuple z_3 does not have a lead.

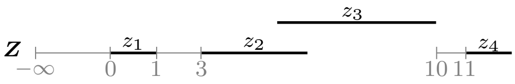


Fig. 13: Leads of example tuples.

The lead of $s \in \boldsymbol{S}$ is the maximal interval preceding $s.T$ during which no tuple in $\boldsymbol{S}$ exists. If a tuple $r \in \boldsymbol{R}$ overlaps with $s.X$, then $r.T \cap s.X$ is the time during which r must be returned as a result tuple for $\boldsymbol{R} \rhd_T \boldsymbol{S}$. A lead has always length larger than 0. If there does not exist such an interval for s, then s does not have a lead. In a relation, e.g., $\boldsymbol{S}$, there cannot exist two leads that overlap with each other: this guarantees that no backtracking is needed for computing $\boldsymbol{R}_i \rhd_T \boldsymbol{S}$ (cf. Example 7).

The lead of a tuple $s_j \in \boldsymbol{S}$ can be computed on the fly. Since $\boldsymbol{S}$ is sorted by T_s, the lead is computed as $s_j.X = [s_{j-a}.T_e, s_j.T_s)$, with $a > 0$, where s_{j-a} is the tuple preceding s_j with the largest T_e value. If $s_j.X$ has a duration larger than 0, then s_j has a lead; otherwise it doesn't. For example, in Figure 15, $s_1.X$ and $s_7.X$ are the only leads.

To compute a temporal anti-join, the first m $\mathcal{DIP}$ partitions $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ are anti-joined with the entire relation $\boldsymbol{S}$; the same is done for $\{\boldsymbol{R}_{m+1}, \ldots, \boldsymbol{R}_{m*2}\}$, and so on:

$$\boldsymbol{R} \rhd_T \boldsymbol{S} \iff \bigcup_{i=1}^{c_{\boldsymbol{R}}/m} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \rhd_T^{\mathcal{DIP}} \boldsymbol{S}) \tag{3}$$

Figure 14 illustrates that the cost for a $\mathcal{DIP}$ anti-join is linear in the size of $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ and $\boldsymbol{S}$.

Example 7 We use Equation 3 to compute a temporal anti-join on relations $\boldsymbol{R}$ and $\boldsymbol{S}$ of our running example (cf. Figure 15). Only $\boldsymbol{R}$ is partitioned, and a $\mathcal{DIP}$ anti-join $\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\} \rhd_T^{\mathcal{DIP}} \boldsymbol{S}$ is computed. Tuples r_1 and s_1 are the first to be fetched, and $s_1.X = [-\infty, s_1.T_s) = [-\infty, 0)$. Tuple r_1 does not overlap with $s_1.X$. Since $s_1.X$ ends before r_1, we switch to $\boldsymbol{R}_2$ and r_3 is fetched. Tuple r_3 does not overlap with $s_1.X$, and, r_4 is fetched from $\boldsymbol{R}_3$. We can conclude that $s_1.X$ does not overlap with any outer tuple, therefore a new tuple is fetched from $\boldsymbol{S}$, i.e., s_2. Since the length of $s_2.X = [8, 1)$ is not larger than 0 (i.e., s_2 does not have a lead), no output is produced for r_1, nor for r_3, nor for r_4. Eventually s_7 is fetched, whose lead is larger than 0. Since r_6 overlaps with $s_7.X$, a result tuple for r_6 with time $r_6.T \cap s_7.X = [12, 13)$ is produced.

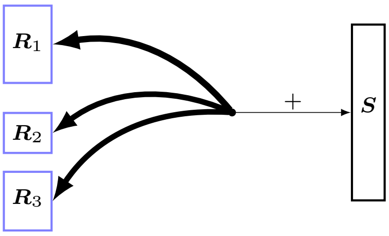
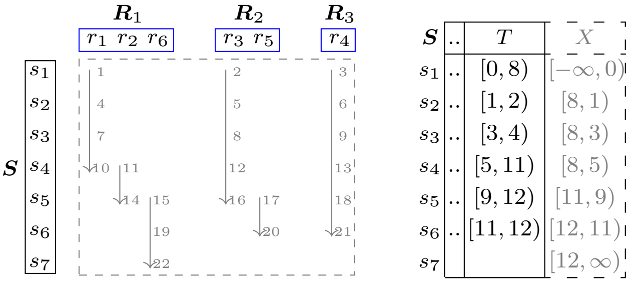


Fig. 14: A temporal anti-join between $\boldsymbol{R}$ and $\boldsymbol{S}$ is computed by joining m outer partitions with $\boldsymbol{S}$. No backtracking is done.



$\boldsymbol{S}$	..	T	X
s_1	..	[0, 8)	$[-\infty, 0)$
s_2	..	[1, 2)	[8, 1)
s_3	..	[3, 4)	[8, 3)
s_4	..	[5, 11)	[8, 5)
s_5	..	[9, 12)	[11, 9)
s_6	..	[11, 12)	[12, 11)
s_7			$[12, \infty)$

Fig. 15: Anti-join computed using $\mathcal{DIP}$: for each $\boldsymbol{R}_i$ tuple, its timestamp is compared with the lead $s.X$ during which no tuple exists in $\boldsymbol{S}$; no backtracking is needed.

6.2.1 CPU Cost

We determine the CPU cost as the upper-bound for the number of unproductive comparisons for a temporal anti-join. Again, we use c to indicate the number of partitions.

Lemma 7 *Consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$, and let c be the number of $\mathcal{DIP}$ partitions of $\boldsymbol{R}$. The number of unproductive comparisons for computing $\boldsymbol{R} \rhd_T \boldsymbol{S}$ using $\mathcal{DIP}$ is upper-bounded by $c \times n$.*

Proof From Equation (3) we get

$$\mathrm{CPU}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) = \mathrm{CPU}\Big(\bigcup_{i=1}^{c/m} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \triangleright_T^{\mathcal{DIP}} \boldsymbol{S})\Big).$$

Remember that for $\triangleright_T^{\mathcal{DIP}}$ no backtracking is needed. Since $\boldsymbol{S}$ is not partitioned, the number of iterations is at most $m*\frac{n}{c}+(m-1)*n+n$, i.e., the cost for scanning $\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\}$, plus $m-1$ partition switches for each inner tuple, plus the cost for scanning $\boldsymbol{S}$. Thus, the number of tuple comparisons in the worst case is:

$$\begin{aligned}\mathrm{CPU}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) &= \sum_{1}^{c/m}(m * \tfrac{n}{c} + (m-1) * n + n) \\ &= n + c * n \\ &\approx c * n\end{aligned}$$

In terms of unproductive comparisons we have 0 result tuples in the worst case and get: $\mathrm{CPU}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) - |\boldsymbol{R} \triangleright_T \boldsymbol{S}| = c * n$ unproductive comparisons. ■

Lemma 7 asserts that for computing a temporal anti-join $\mathcal{DIP}$ limits the number of comparisons per tuple to the number of partitions. State-of-the-art techniques [9], instead, make a nested-loop for computing the intervals of the output tuples. This yields a quadratic number of comparisons.

6.2.2 I/O Cost

This section quantifies the number of block I/Os for computing a temporal anti-join using $\mathcal{DIP}$. Again, we assume that the $\mathcal{DIP}$ partitions are equally sized, i.e., $b = \frac{B}{c}$.

Lemma 8 *Consider relation* $\boldsymbol{R}$ *partitioned into c* $\mathcal{DIP}$ *partitions, and relation* $\boldsymbol{S}$*. The number of I/Os for computing* $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ *using* $\mathcal{DIP}$ *is* $\frac{c}{m} \times B$*.*

Proof From Equation 3, we get

$$\mathrm{IO}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) = \mathrm{IO}\Big(\bigcup_{i=1}^{c/m} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \triangleright_T^{\mathcal{DIP}} \boldsymbol{S})\Big)$$

With equally sized partitions, we get:

$$\begin{aligned}&= \tfrac{c}{m} \times \mathrm{IO}(\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \triangleright_T^{\mathcal{DIP}} \boldsymbol{S}) \\ &= \tfrac{c}{m} \times \big(b * m + B\big) \\ &= B + \tfrac{c}{m} \times B \\ &\approx \tfrac{c}{m} \times B \end{aligned} \qquad (4)$$

■

When computing $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ with $\mathcal{DIP}$, independent of the number of partitions, the tuples of $\boldsymbol{R}$ are scanned only once, while those of $\boldsymbol{S}$ are scanned $\frac{c}{m}$ times. Overall, the cost of our approach is linear with the number of partitions c. In addition, processing m outer partitions simultaneously further reduces the number of I/Os by a factor of m.

6.3 Temporal Aggregation

A temporal aggregation $\vartheta_F^T(\boldsymbol{R})$ returns, for each maximal interval during which a set of $\boldsymbol{R}$ tuples is valid, the result of an aggregation function F. For example, in Figure 16 the average price of the luxury suites booked in hotel $\boldsymbol{R}$ is computed. The first output row says that, between time 1 and 5, the average price is 80\$. Note that, due to change preservation [9], two different tuples are returned for $[8, 10)$, and $[10, 11)$ because, even if their aggregation value is the same, their lineage is different.

$\vartheta^T_{avg(\$)}(\boldsymbol{R})$

T	$avg(\$)$
$[1, 5)$	80
$[6, 7)$	60
$[7, 8)$	71.6
$[8, 10)$	75
$[10, 11)$	75
$[11, 13)$	80

Fig. 16: Temporal aggregation *avg* applied to relation $\boldsymbol{R}$

A temporal aggregation $\vartheta_F^T(\boldsymbol{R})$ on a table can be decomposed into a sequence of full outer joins between its $\mathcal{DIP}$ partitions:

$$\vartheta_F^T(\boldsymbol{R}) \iff \pi_{T,F'}(\boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2 ⟗_T^{\mathcal{DIP}} \cdots ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_c) \qquad (5)$$

The proof of this equivalence is given in Appendix A.

As shown in Figure 17, the first partition is full outer joined with the second partition. Afterwards, the intermediate result is full outer joined with the third partition, etc. In other words, $c-1$ $\mathcal{DIP}$ full outer joins are computed. Finally, for each result tuple, the projection uses function F' to aggregate the c values in a tuple using the same aggregation as the one in F (e.g., AVG).

Example 8 We use Equation (5) to transform the temporal aggregation $\vartheta^T_{avg(\$)}(\boldsymbol{R})$ of our running example to $\pi_{T,\mathrm{AVG}(\boldsymbol{R}_1.\$,\boldsymbol{R}_2.\$,\boldsymbol{R}_3.\$)}(\boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_3)$.

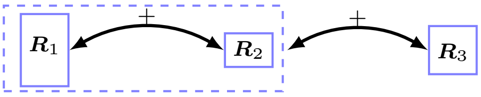


Fig. 17: A temporal aggregation is computed by (full-outer) joining at linear cost the $\mathcal{DIP}$ partitions.

Without loss of generality, we consider only the attributes needed to compute the aggregation, i.e., T and \$. The first full outer join yields $\boldsymbol{R}_1$ $⟗_T^{\mathcal{DIP}}$ $\boldsymbol{R}_2 = \{([1,5), 80, \mathit{null}),\ ([6,7), 60, \mathit{null}),\ ([7,8), 60, 80),\ ([10,11), 80, 70),\ ([11,13), 80, \mathit{null})\}$. Those tuples are further joined to $\boldsymbol{R}_3$ producing the result shown in Figure 18. The projection $\pi_{T,\mathrm{AVG}(\boldsymbol{R}_1.\$,\boldsymbol{R}_2.\$,\boldsymbol{R}_3.\$)}$ outputs, for each time interval in the result, the average of the three prices, which corresponds to the result in Figure 16.

$\boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_3$

T	$\boldsymbol{R}_1.\$$	$\boldsymbol{R}_2.\$$	$\boldsymbol{R}_3.\$$
$[1,5)$	80	null	null
$[6,7)$	60	null	null
$[7,8)$	60	80	75
$[8,10)$	null	null	75
$[10,11)$	80	70	null
$[11,13)$	80	null	null

Fig. 18: Full outer join between the $\mathcal{DIP}$ partitions of $\boldsymbol{R}$.

A temporal full outer join between $\boldsymbol{R}_i$ and $\boldsymbol{R}_{i+1}$ returns all join matches ($\boldsymbol{R}_i \bowtie_T^{\mathcal{DIP}} \boldsymbol{R}_{i+1}$) plus all anti-join matches of $\boldsymbol{R}_i \triangleright_T^{\mathcal{DIP}} \boldsymbol{R}_{i+1}$ and of $\boldsymbol{R}_{i+1} \triangleright_T^{\mathcal{DIP}} \boldsymbol{R}_i$. Each full outer join of the sequence, and not just the first, can be computed without backtracking. This is so because the result of a full outer join between two $\mathcal{DIP}$ partitions is also a $\mathcal{DIP}$ partition: it does not generate tuples with overlapping timestamps.

6.3.1 CPU Cost

Lemma 9 *The number of unproductive comparisons for a temporal aggregation on relation* $\boldsymbol{R}$ *is upper bounded by* $c \times n$.

Proof Consider Equation (5). Since the projection π can be computed on the fly while writing the result tuples (without doing additional comparisons) we get:

$$\mathrm{CPU}(\vartheta_F^T(\boldsymbol{R})) = \\ \mathrm{CPU}(\boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2 ⟗_T^{\mathcal{DIP}} \ldots ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_c)$$

When computing a temporal aggregation using full outer joins, a comparison between r and s is unproductive if the tuples do not overlap, since such a comparison only adds NULL values to the result (which do not change the aggregate result). Remember that $c-1$ full outer joins are computed. Since the highest cardinality of a temporal aggregation is $2n-1$ [18] (i.e., the number of different T_s and T_e values - 1), in the worst case most of those $2n-1$ intervals are produced by the first full outer join, and the remaining $c-2$ joins perform about $2n-1$ unproductive comparisons each. Thus, we get $(c-2) \times (2n-1) \approx c \times n$ unproductive comparisons. ■

Figure 19 illustrates such a worst case scenario for computing a temporal aggregation using $\mathcal{DIP}$, with $n = 8$ tuples and $c = 3$ partitions (note that the last partition stores only one tuple).

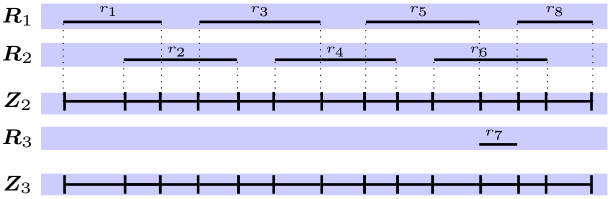


Fig. 19: Highest cardinality for a full outer join $\boldsymbol{Z}_2 = \boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2$. For the full outer join $\boldsymbol{Z}_3 = \boldsymbol{Z}_2 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_3$ all comparisons except one are unproductive since they do not change any aggregate value.

The first full outer join produces $13 \approx 2n-1$ intervals, and the second does $13 \approx 2n-1$ unproductive comparisons (including the one with the last lead) since only one overlapping tuple exists in $\boldsymbol{R}_3$.

6.3.2 I/O Cost

Lemma 10 *The number of I/Os for computing a temporal aggregation* $\vartheta_F^T(\boldsymbol{R})$ *using* $\mathcal{DIP}$ *is upper-bounded by* $c \times B$.

Proof Consider Equation (5). Since the projection π can be computed on the fly while writing the result tuples (without additional I/Os) we get:

$$\mathrm{IO}(\vartheta_F^T(\boldsymbol{R})) = \\ \mathrm{IO}(\boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2 ⟗_T^{\mathcal{DIP}} \ldots ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_c).$$

For the first full outer join, b blocks are read for the outer input, and b blocks for the inner one. In the worst case, $2(|\boldsymbol{R}_1|+|\boldsymbol{R}_2|) = 2(\frac{n}{c}+\frac{n}{c}) = 4\frac{n}{c}$ tuples are returned and $4 \times b$ blocks are needed for storing this intermediate result. For the second join, $4 \times b$ blocks are read for the outer input and b for the inner. In the worst case, $6 \times b$ blocks are needed for storing the intermediate result. Generalizing, for the $(c-1)$-th full outer join, i.e., the last one to compute, we read in the worst case

Temporal Join $\boldsymbol{R} \bowtie_T \boldsymbol{S}$

```
$\boldsymbol{R}_1, \ldots, \boldsymbol{R}_{c_{\boldsymbol{R}}} \leftarrow Create\mathcal{DIP}(\boldsymbol{R})$
$\boldsymbol{S}_1, \ldots, \boldsymbol{S}_{c_{\boldsymbol{S}}} \leftarrow Create\mathcal{DIP}(\boldsymbol{S})$
$\boldsymbol{Z} = \emptyset$
for $i = 1$ to $c_{\boldsymbol{R}}$ do
    $k = min(i + m - 1, c_{\boldsymbol{R}})$
    for $j = 1$ to $c_{\boldsymbol{S}}$ do
        $\boldsymbol{T} = \mathcal{DIP}Merge(\{\boldsymbol{R}_i, \ldots, \boldsymbol{R}_k\}, \boldsymbol{S}_j, \bowtie)$
        $\boldsymbol{Z} = \boldsymbol{Z} \cup \boldsymbol{T}$
    $i = k$
return $\boldsymbol{Z}$
```

Temporal Anti-join $\boldsymbol{R} \rhd_T \boldsymbol{S}$

```
$\boldsymbol{R}_1, \ldots, \boldsymbol{R}_c \leftarrow Create\mathcal{DIP}(\boldsymbol{R})$
$\boldsymbol{Z} = \emptyset$
for $i = 1$ to $c$ do
    $k = min(i + m - 1, c)$
    $\boldsymbol{T} = \mathcal{DIP}Merge(\{\boldsymbol{R}_i, \ldots, \boldsymbol{R}_k\}, \boldsymbol{S}, \rhd)$
    $\boldsymbol{Z} = \boldsymbol{Z} \cup \boldsymbol{T}$
    $i = k$
return $\boldsymbol{Z}$
```

Temporal Aggregation $\vartheta_F^T(\boldsymbol{R})$

```
$\boldsymbol{R}_1, \ldots, \boldsymbol{R}_c \leftarrow Create\mathcal{DIP}(\boldsymbol{R})$
$\boldsymbol{Z} \leftarrow \boldsymbol{R}_1$
for $i = 2$ to $c$ do
    $\boldsymbol{Z} = \mathcal{DIP}Merge(\{\boldsymbol{Z}\}, \boldsymbol{R}_i, ⟗)$
$\boldsymbol{Z} = \pi_{T,F'}(\boldsymbol{Z})$
return $\boldsymbol{Z}$
```

Fig. 20 Each temporal operator is computed calling multiple times $\mathcal{DIP}Merge$.

$2 \times (c-1) \times b$ blocks for the outer input and b for the inner, and we write $2 \times c \times b$ blocks for the result. Summing up the I/Os of all $c-1$ full outer joins we get:

$$\begin{aligned} \mathrm{IO}(\vartheta_F^T(\boldsymbol{R})) =& \sum_1^{c-1}(2 \times i \times b + b + 2 \times (i+1) \times b) \\ =& 2 \times b \sum_1^{c-1} i + \sum_1^{c-1} b + 2 \times b \sum_1^{c-1}(i+1) \end{aligned}$$

Since $\sum_1^{c-1} i = \frac{(c-1)c}{2}$ and $\sum_1^{c-1}(i+1) = \frac{c(c+1)-2}{2}$, we obtain

$$\begin{aligned} &2\frac{(c-1)c}{2} \times b + (c-1)b + 2\frac{c(c+1)-2}{2} \times b = \\ &2 \times c^2 b + (c-3)b = 2c \times B + B - 3b \simeq c \times B \end{aligned}$$ ■

The I/O cost for computing a temporal aggregation is linear in the number of the partitions. Furthermore, opposite to the state of the art approaches, such as the Aggregation Tree [18], the Balanced Tree [23], and the SB-Tree [31], our approach is not limited to distributive aggregates: standard deviation, for example, is also computable using $\mathcal{DIP}$.

7 Implementation

In this section we discuss our implementation. First we describe how to implement each temporal operator in the executor of the DBMS using a sequence of merges. This is done with a general $\mathcal{DIP}Merge$ function that merges $\mathcal{DIP}$ partitions for either a temporal join, a temporal anti-join or a temporal aggregation. Next, we propose an efficient implementation of $\mathcal{DIP}Merge$, i.e., the algorithm that computes temporal joins, anti-joins, and full outer joins without backtracking.

7.1 Implementing the Temporal Operators

Equations (1), (3), and (5) directly lead to the algorithms in Figure 20. In the executor of the DBMS, each temporal operator is computed by first creating the partitions (i.e., calling $Create\mathcal{DIP}$), and then calling iteratively $\mathcal{DIP}Merge$ as follows:

For $\boldsymbol{R} \bowtie_T \boldsymbol{S}$, first $\boldsymbol{R}$ and $\boldsymbol{S}$ are partitioned by $Create\mathcal{DIP}$. Then m outer $\mathcal{DIP}$ partitions are $\mathcal{DIP}Merge$d with each inner partition, and the result tuples are collected in $\boldsymbol{Z}$.

For $\boldsymbol{R} \rhd_T \boldsymbol{S}$, only $\boldsymbol{R}$ is partitioned. Then m outer partitions are $\mathcal{DIP}Merge$d with the entire $\boldsymbol{S}$ relation, and the result tuples are collected in $\boldsymbol{Z}$.

For $\vartheta_{f(A)}^T(\boldsymbol{R})$, the first $\mathcal{DIP}$ partition is $\mathcal{DIP}Merge$d with the second, and the result tuples are collected in $\boldsymbol{Z}$; $\boldsymbol{Z}$ is iteratively $\mathcal{DIP}Merge$d with the subsequent $\mathcal{DIP}$ partitions[1]. Finally, a projection on $\boldsymbol{Z}$ computes the aggregation function F' on the values $\boldsymbol{R}_1.A, \ldots, \boldsymbol{R}_c.A$.

7.2 Implementation of $\mathcal{DIP}Merge$

Algorithm 2 shows the implementation of $\mathcal{DIP}Merge$. The first argument is a set of $\mathcal{DIP}$ partitions $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ each with schema $(T, A_1, \ldots, A_p)$. The second argument $\boldsymbol{S}$ is an inner $\mathcal{DIP}$ partition (or the entire relation) with schema $(T, B_1, \ldots, B_q)$. Finally, Op is the operator to compute, i.e., a temporal join, anti-join, or full outer join. The algorithm computes Op with a single scan of $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ and $\boldsymbol{S}$, without backtracking.

At the beginning, the lead X of the current tuple in the i-th outer partition $r[i]$ is initialized as the interval between $-\infty$ and the starting point of the first tuple. We do the same for the current tuple s in $\boldsymbol{S}$. Initially $i = 1$. During each iteration, the algorithm fetches a new tuple from $\boldsymbol{R}_i$ (line 19). When all tuples in $\boldsymbol{R}_i$ that overlap s have been found, the algorithm switches to partition $\boldsymbol{R}_{i+1}$ (line 27). Once all outer partitions have been checked against s, the algorithm fetches a new $\boldsymbol{S}$ tuple (line 31) and restarts processing $\boldsymbol{R}_i$ from its last

[1] Our implementation applies standard DBMS optimization techniques, such as projecting on the attributes required for the aggregation (i.e., T and A), so that the full outer join result includes only the needed attributes.

Algorithm 2: $\mathcal{DIP}Merge(\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}, \boldsymbol{S}, Op)$

```
   Input  : $\boldsymbol{R}_i(T, A_1, ..., A_p), \boldsymbol{S}(T, B_1, ..., B_q), Op \in \{\bowtie, \triangleleft, ⟗\}$
   Output: $\boldsymbol{Z}(T, A_1, \ldots, A_p, B_1, \ldots, B_q)$
1  for $i = 1$ to $m$ do
2      $r[i] \leftarrow fetchRow(\boldsymbol{R}_i)$
3      $r[i].X = [-\infty, r[i].T_s)$                          // lead of $r[i]$
4  $i = 1$
5  $s \leftarrow fetchRow(\boldsymbol{S})$
6  $s.X = [-\infty, s.T_s)$                                    // lead of $s$
7  while $!null(r[i].T) \vee !null(s.T)$ do
8      if $Operator = ⟗$ then
9          if $len(r[i].X) > 0 \wedge overlap(r[i].X, s.T)$ then
10             $\boldsymbol{Z} = \boldsymbol{Z} \cup <(r[i].X \cap s.T), null^p, s.B_1, \ldots, s.B_q>$
11     if $Operator \in \{\triangleright, ⟗\}$ then
12         if $len(s.X) > 0 \wedge overlap(r[i].T, s.X)$ then
13             $\boldsymbol{Z} = \boldsymbol{Z} \cup <(r[i].T \cap s.X), r.A_1, \ldots, r.A_p, null^q>$
14     if $Operator \in \{\bowtie, ⟗\}$ then
15         if $overlap(r[i].T, s.T)$ then
16             $\boldsymbol{Z} = \boldsymbol{Z} \cup <$
                 $(r[i].T \cap s.T), r[i].A_1, ..., r[i].A_p, s.B_1, ..., s.B_q>$
17     if $!null(r[i].T) \wedge (null(s.T) \vee r[i].T_e \leq s.T_e)$ then
18         if $r[i].T_e > r[i].X_s$ then $longestR[i] = r[i].T_e$
19         $r[i] \leftarrow FetchRow(\boldsymbol{R}_i)$
20         if $!null(r[i])$ then
21             $r[i].X = [longestR[i], r[i].T_s)$
22         else
23             $r[i].T = null$
24             $r[i].X = [longestR[i], \infty)$
25     else
26         if $i < m$ then
27             $i = i + 1$
28         else
29             $i = 1$
30             if $s.T_e > s.X_s$ then $longestS = s.T_e$
31             $s \leftarrow FetchRow(\boldsymbol{S})$
32             if $!null(s)$ then
33                 $s.X = [longestS, s.T_s)$
34             else
35                 $s.T = null$
36                 $s.X = [longestS, \infty)$
37 return $\boldsymbol{Z}$
```

scanned tuple. The result tuples change depending on the *Op* to be computed (lines 8-16):

- **Join:** For the join matches, we directly use Lemma 2 since $r[i]$ and s only join iff they overlap: if tuple $r[i]$ is the last join match of s, then no tuple before $r[i]$ can match with the successor of s. Line 16 outputs the join matches by concatenating the attributes of $r[i]$ and s.
- **Anti-Join:** For the anti-join matches, the lead $s.X$ must be considered. Lemma 2 holds between $r[i].T$ and $s.X$. Line 13 outputs the anti-join matches. Since no $\boldsymbol{S}$ tuple exists for an anti-join result tuple, i.e., during $(r[i].T \cap s.X)$, for each attribute $B_1, \ldots, B_q$ of the inner input a NULL value is returned.
- **Full Outer Join:** To make sure that the full outer join returns a $\mathcal{DIP}$ partition with sorted elements (so that the next full outer join of the sequence does not require any additional sorting), the anti-join matches *must* be written before the join matches. Since the lead $s.X$ (or $r[i].X$) is the interval between s (or $r[i]$) and its predecessor, $s.X$ comes always before $s.T$ (as well as $r[i].X$ comes before $r[i].T$), and an interval overlapping with $s.X$ is written before an interval overlapping with $s.T$.

The algorithm ends when all input tuples have been processed (i.e., when $r[i].T$ and $s.T$ are both null). Note that in case only one input (e.g., $\boldsymbol{S}$) has been scanned entirely, the algorithm goes on to return the anti-join matches of all remaining outer tuples.

8 Experiments

For the experiments on disk, we used an Intel Core i7-3820QM Processor @ 2.7 GHz machine with 4GB main memory and a Samsung 840 EVO 500 GB Solid State Drive (Sequential Read Speed 540 MB/s, and Sequential Write Speed 520 MB/s), running OS X 10.11.6. (L1 chace: 32KB, L2 cache: 256 KB, L3 cache: 8 MB). For the experiments in main memory, we used a 2 x Intel(R) Xeon(R) CPU E5-2440 (6 cores each) @ 2.40GHz with 64GB main memory, and running CentOS 6.4 (L1 cache: 192 KB, L2 cache: 1536 KB, L3 cache: 15 MB). For the main memory experiments, all indices and all data are kept in memory and no disk I/O for reading or sorting is done.

We compute the performances of Temporal Alignment (Align, [9]), the TimeLine Index (TimeLine, [16]), Overlap Interval Partitioning (OIP, [10]), Sort-Merge (SM, [14]), the Sweepline algorithm (Sweep, [3]), the Aggregation Tree (AggTree, [18]) Sort-Aggregate (SortAgg, [12]) and $\mathcal{DIP}$. All approaches have been implemented by the authors using C. Real world data as well as synthetic data is used. We use the Swiss Feed Data [29], and the Time Interval (TI) [1], INFECTIOUS [15], and GREEND [22] datasets as real world datasets. For each dataset, we also include the cost of a sequential scan. When comparing with a sequential scan we exclude the cost for sorting, indexing, partitioning, etc. In all other experiments the costs for sorting, indexing and partitioning are included.

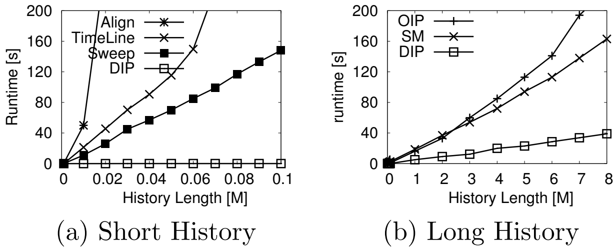
8.1 Real World Data

In this subsection we compare the runtimes of the approaches for computing temporal joins, anti-joins, and aggregations. We use the Swiss Feed Data Warehouse [29] and fix the ratio between the length of the history and the number of tuples to 1:1, e.g., a history of 100k granules stores 100k tuples, and we then increase the history length. Intervals have length varying from 1 to 10k granules: 90% of the intervals have length smaller than 10 granules (they represent lab-measurements that change over time, and must be repeated frequently); 9.5% of the remaining intervals have length up to 1000 granules; 0.5% up to 10000 granules (they represent lab-measurements of values that remain constant, and are repeated seldomly). We vary those parameters in the experiments in Subsection 8.2.

8.1.1 Temporal Joins

First, we compute a temporal join that joins the values of two different nutritive values (Protein and Fat). The runtime is measured for disk-based computations and for in-memory computations.

Execution on disk. Figure 21(a) shows that Align performs badly when the data history grows, since it checks $|\boldsymbol{R}| \times |\boldsymbol{S}|$ comparisons. The TimeLine index performs better since it avoids unproductive comparisons, however each result tuple (r, s) is produced by making one index look-up in $\boldsymbol{R}$ and one in $\boldsymbol{S}$. This is expensive for disk-resident data since each index look-up fetches a block. Finally, long-lived tuples, (e.g., 10k granules long) are fetched multiple times with one index look-up for each tuple they match. Sweepline does not perform well on disk since the active tuples have to be updated when the sweepline advances. This is expensive for disk-resident data.

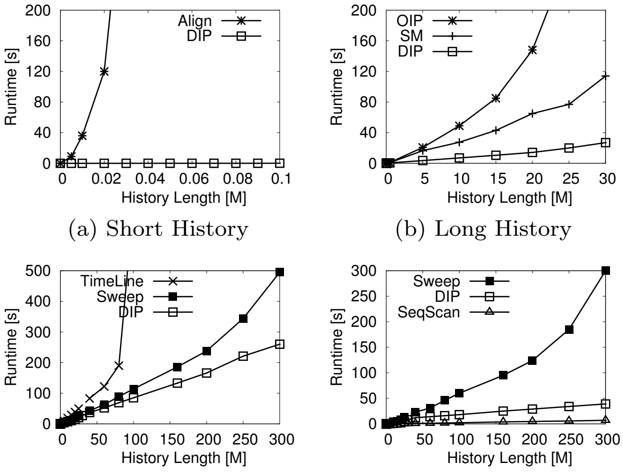


(a) Short History (b) Long History

Fig. 21: Temporal join on disk

In Figure 21(b) we show approaches that scale better on disk, and can handle more data. OIP performs worse than $\mathcal{DIP}$ and SM because of the many short intervals present in the dataset. Those tuples are a bottleneck for OIP since they make the nested-loop between the partitions very expensive in terms of unproductive comparisons: with 8M tuples, 6.5×10^{10} combinations are checked by OIP, 4×10^{9} by SM, and only 6×10^{7} by $\mathcal{DIP}$.

Execution in main memory. Figure 22 shows that all approaches benefit from an in-memory execution as expected.



(a) Short History (b) Long History

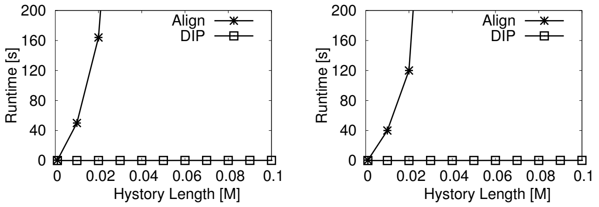
(c) Very Long History (d) Join Runtime Only

Fig. 22: Temporal join in main memory

Figure 22(b) shows that the runtime of OIP, SM, and $\mathcal{DIP}$ is proportional to the amount of unproductive comparisons: with 30M tuples, 2×10^{15} unproductive comparisons are done by OIP, 2.1×10^{11} by SM, and 2.1×10^{9} by $\mathcal{DIP}$. In Figure 22(c), we show that for a history of 300M tuples, $\mathcal{DIP}$ is more than four minutes faster than Sweepline. This is so because, although Sweepline does at most one unproductive comparison per tuple, the list of active tuples is allocated and deallocated at run time yielding a poor memory locality. Computing a random memory access per active tuple makes the join computation expensive for Sweepline. Figure 22(d) shows that, if the sorting (for Sweepline) and the partitioning (for $\mathcal{DIP}$) are computed offline, $\mathcal{DIP}$ computes the join one order of magnitude faster than Sweepline. Our results confirm the experimental evaluation by Stroustrup [28], which shows that accessing memory sequentially is one order of magnitude faster than accessing it randomly. The cost for a join on $\mathcal{DIP}$ partitions has only a slightly higher linear factor than a sequential scan.

8.1.2 Temporal Anti-Joins

In this experiment we compute a temporal anti-join to find all intervals for which a protein measurement but no fat measurement exists. To the best of our knowledge, only Dignös et al. [9] provide a solution for computing temporal anti-joins. The nested-loop with which the alignment operator is computed is however expensive, since query optimizers are not able to use interval T to optimize the query plan. Figure 23 shows that the runtime of alignment on disk is similar to the runtime in main memory because a small dataset, once it has been fetched from disk, is cached in main memory. However, checking n^2 combinations is expensive even in main memory. $\mathcal{DIP}$ provides the first non-quadratic solution for computing temporal anti-joins.



(a) Disk (b) Memory

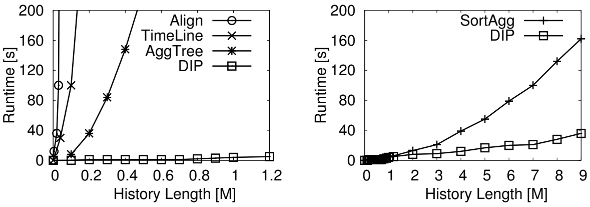
Fig. 23: Temporal anti-join.

8.1.3 Temporal Aggregation

This experiment reports the runtime for the computation of a temporal aggregation, i.e., we compute the average value for the measurements stored in the Swiss Feed Data Warehouse. The Aggregation Tree is not efficient (Figure 24.a) and does not scale even with high memory availability (Figure 25.a). The TimeLine Index performs very badly on disk, but is robust in memory until 150M tuples. Afterwards it deteriorates since the index gets large (remember that for each tuple two entries are stored) and, at the same time, the number of look-ups increases. $\mathcal{DIP}$ does not require an index and stays stable. Sort-aggregate requires backtracking and performs slower than $\mathcal{DIP}$ (Figure 24.a). In main memory, our approach grows linearly with the length of the data history (Figure 25.b).

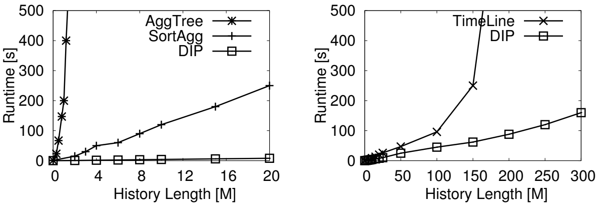
8.1.4 TI Dataset

The TI dataset [1] is public and stores the Universal Resource Identifiers (URIs) for the time intervals commonly used by the UK Government. Tuples are stored as $< T_s, T_e, \text{URI} >$ pairs. The time granularity is expressed in number of days. The intervals have length 1 (i.e., one day), {28,29,30,31} (i.e., one month), {365,366} (i.e., one year), {547,548} (i.e., one and a half years), and {730,731} (i.e., two years). The history is 60 years long.



(a) Short History (b) Long History

Fig. 24: Temporal aggregation on disk.



(a) Short History (b) Long History

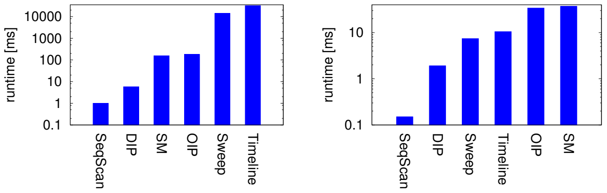
Fig. 25: Temporal aggregation in main memory.

Figure 26 shows the runtime for computing a self-join on disk and in main-memory using the TI dataset. In memory, $\mathcal{DIP}$ is four times faster than Sweepline, and over an order of magnitude faster than the other approaches. On disk, $\mathcal{DIP}$ is two order of magnitude faster than other approaches; Sweepline deteriorates since for each tuple the file storing the active tuples must be rewritten entirely (the URIs have different length). $\mathcal{DIP}$ is the only approach that is robust both if the dataset is memory- and if it is disk-resident. It accesses the tuples sequentially and, at the same time, keeps the number of unproductive comparisons low.

8.1.5 GREEND Dataset

In this experiment we use a long data history with many short intervals and a few long intervals. The GREEND dataset [22] is public and contains detailed power usage information obtained through a measurement campaign in households in Austria and Italy from January 2010 to October 2014. Tuples are stored as $< T_s, T_e, Device_1, \dots, Device_m >$ where T_s is the time when the current measurement has been taken, T_e is the time when the next measurement has been taken,

	SeqScan	$\mathcal{DIP}$	Sweep	SM	OIP	Timeline
Disk	1.0	5.8	14000	156	180	32000
Memory	0.15	1.90	7.42	34	37	10.5

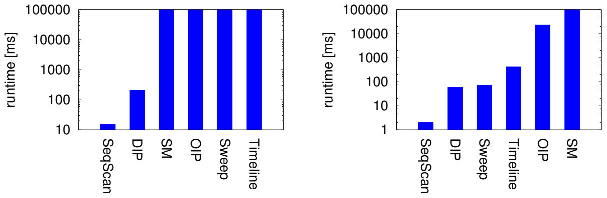


(a) Disk (b) Memory

Fig. 26: Temporal join for the TI dataset in milliseconds.

and $Device_i$ stores the amount of energy consumption of a given device. Intervals have on average 1.7 seconds length.

	SeqScan	$\mathcal{DIP}$	Sweep	SM	OIP	Timeline
Disk	15	212	> 1 day	> 1 day	1 day	> 1 day
Mem	2	58	72	> 1 day	23300	423



(a) Disk (b) Memory

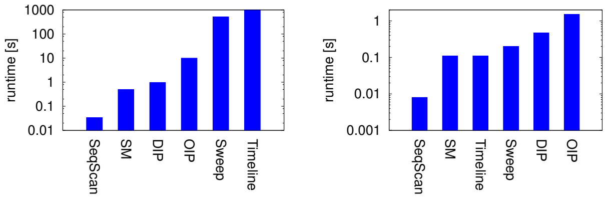
Fig. 27: Temporal join for the GREEND dataset in seconds.

Figure 27 reports the runtime for computing a self-join on the GREEND dataset. $\mathcal{DIP}$ performs best. On disk it is the only approach that does not deteriorate. Although the average interval length is small, sort-merge performs poorly since eight long-lived tuples are present in the dataset and backtracking makes it akin to a nested loop. In memory Timeline performs an order of magnitude slower than $\mathcal{DIP}$ since the 8 long-lived tuples are refetched for each join match by a new index look-up. OIP performs badly because of the nested loop with which the partitions are joined.

8.1.6 INFECTIOUS Dataset

This experiment is a best case scenario for existing approaches since there are no long-lived tuples, the history is short, and the number of overlapping intervals is small. The INFECTIOUS dataset [15] is public and stores the timpestamp at which a contact between visitors occurred during the artscience exhibition "INFECTIOUS: Stay Away!" which took place at the Science Gallery in Dublin, Ireland, from May to July 2009. The history is two months long. Tuples are stored as $< T_s, T_s + 20, Visitor_1, Visitor_2 >$. The time granularity is expressed in seconds, and the intervals have all length 20 seconds. During the art exhibition, contacts between different visitors happen at the same time (intervals are either equal or disjoint), with a peak of 51 contacts in the same 20 seconds slot: 51 $\mathcal{DIP}$ partitions are produced.

	SeqScan	SM	$\mathcal{DIP}$	Sweep	OIP	Timeline
Disk	.034	0.5	0.96	510	10	1050
Mem	.008	0.11	0.47	0.20	1.50	0.11



(a) Disk (b) Memory

Fig. 28: Temporal join for the INFECTIOUS dataset in seconds.

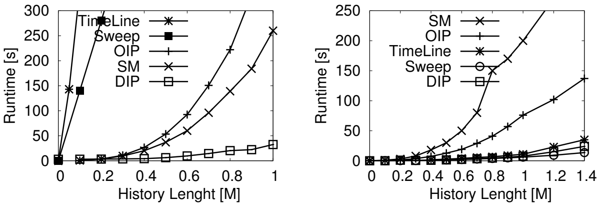
Figure 28 shows the runtime for computing a self-join on disk and in main-memory using the INFECTIOUS dataset. The results in Figure 28(a) show that $\mathcal{DIP}$ stays competitive even if few tuples overlap. Clearly sort-merge performs better since all intervals have 20 seconds length and no long-lived tuple exists: backtracking refetches only few tuples. In main memory also Timeline and Sweepline perform well: the first since, with short intervals only, few join matches have to be retrieved through the index; the second because at a given time point all active tuples are recent, and thus allocated close to each other.

8.2 Synthetic data

In this subsection we use synthetic data, and evaluate the approaches by varying the characteristics of the data history. We first increase the number of partitions by increasing the number of tuples valid as time passes by. Then, we show the effect of processing m partitions simultaneously for the average and worst case scenario.

8.2.1 Size of Dataset

This experiment shows how the approaches behave when the number of tuples valid as the time passes by increases, i.e., when recently more data are collected compared to the past. For each 100k time granules in the history, 100k more tuples exist compared to the previous 100k time granules (e.g., from the 0-th to 100k-th time granule of the history we have 100k tuples; from the 100k-th to 200k-th granule we have 200k tuples; from the 200k-th to 300k-th granule we have 300k tuples; etc.).



(a) Disk (b) Memory

Fig. 29: Increase of the number of tuples collected throughout of the data history

In Figure 29(a), we can see that $\mathcal{DIP}$ is the only approach that stays robust for disk-resident data. This is so because $\mathcal{DIP}$ is not affected by the size of the partitions: for c $\mathcal{DIP}$ partitions the amount of unproductive comparisons of $\mathcal{DIP}$ does not change if the partitions are equally sized or if they are unbalanced. For OIP, if the partitions are unbalanced, the unproductive comparisons increase. Sweepline and Timeline perform well for an in-memory execution since the history length is just 1.4M granules: for Timeline the number of lookups is small; for Sweepline few insertions and deletions are done in the list of active tuples. For a longer data history (cf. Figure 22) both approaches do not scale.

8.2.2 Varying m in the Average Case for $\mathcal{DIP}$

In this experiment, we show how $\mathcal{DIP}$ behaves in the average case for different values of m. Partitions are stored on disk. In Figure 30, we show that the performances of a join increases only by an order of two when m grows since, as shown in Equation (2a), relation $\boldsymbol{R}$, independent of the value of m, must be scanned c times. The number of scans of $\boldsymbol{S}$, instead, is reduced by a factor of m. For an anti-join, instead, $\boldsymbol{R}$ is scanned only once, therefore when the number m of outer partitions processed simultaneously increases, the number of times $\boldsymbol{S}$ is scanned decreases (Equation 4). Figure 30 shows an improvement of the performances of an order of magnitude.

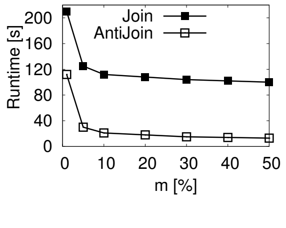


Fig. 30: Increase of m for a join and an anti-join.

8.2.3 Varying m in the Worst Case for $\mathcal{DIP}$

In this experiment, we show the worst case for computing a join using $\mathcal{DIP}$. This happens if all tuples overlap, and each tuple is placed in a different partition. Note that this means there is a time point where all data is valid, which is not usually the case for temporal databases. Since the partitions are small, they are kept in memory. In this experiment each $\boldsymbol{R}$ tuple overlaps with all $\boldsymbol{S}$ tuples, and all approaches are quadratic. The data is partitioned into 10k $\mathcal{DIP}$ partitions.

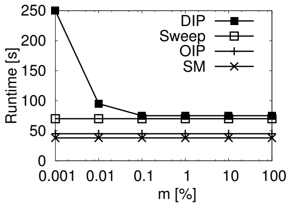


Fig. 31: High number of $\mathcal{DIP}$ partitions.

In Figure 31 we show that, already with a small amount of cache and parallel processing, our approach becomes competitive in a worst case scenario. The graph shows that as soon as 0.1% of the outer partitions are processed in parallel, $\mathcal{DIP}$ reaches the performance of the Sweepline approach. This is so for two reasons (cf. Equation 2): *i)* small outer partitions are entirely cached and can be reused for the next $\mathcal{DIP}$*Merge*; *ii)* when m grows, the number of scans of $\boldsymbol{S}$ decreases by a factor of m. OIP and SM are slightly faster in a worst case scenario since the tuples of a relation (for SM) and of a partition (for OIP) are accessed sequentially, while for $\mathcal{DIP}$ and for Sweepline tuples are accessed randomly since each tuple is in a different partition (for $\mathcal{DIP}$) and each active tuple in a different memory block (for Sweepline).

9 Conclusions and Future Work

In this paper we have proposed *Disjoint Interval Partitioning* ($\mathcal{DIP}$). $\mathcal{DIP}$ partitions a temporal relation into the minimum number c of partitions storing non-overlapping tuples. $\mathcal{DIP}$ is a new and general approach that makes sort-based operator efficient in the presence of interval data. We have proved that temporal joins, anti-joins, and aggregation are computed with at most c unproductive comparisons per tuple, independently of the size of the partitions. We have empirically shown that $\mathcal{DIP}$ outperforms the state-of-the-art solutions when computing temporal operators over historical data.

Interesting directions for future work are to: *i)* incrementally update the $\mathcal{DIP}$ partitions: if a new tuple r is stored in the database and its timestamp is in the past, then checking only the last tuple of the partitions does not ensure that r is disjoint from all other tuples; *ii)* efficiently incorporating conditions over non-temporal attributes: while for a temporal equijoin they can be trivially computed on the fly, for anti-joins it becomes complex to generate the leads since their starting point depends on the previously scanned tuple that has the same non-temporal values; *iii)* investigate the potential of $\mathcal{DIP}$ in column stores; *iv)* consider techniques that support block suballocation for cases where the partitions are much smaller than a block.

10 Acknowledgments

This work has been developed in the context of the Tameus project between the University of Zurich and Agroscope, with funding from the Swiss National Science Foundation. We thank Jerinas Gresch from Siemens for contributing to the implementation of $\mathcal{DIP}$*Merge*. We thank the anonymous reviewers for their insightful suggestions and comments, which helped us to improve the paper.

References

1. The time intervals dataset. `https://data.gov.uk/dataset/time-intervals`, 2015.
2. M. Agesen, M. H. Böhlen, L. Poulsen, and K. Torp. A split operator for now-relative bitemporal databases. In *ICDE*, pages 41–50, 2001.
3. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *VLDB*, pages 570–581, 1998.
4. M. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.
5. M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, Dec. 2000.
6. F. Cafagna, M. H. Böhlen, and A. Bracher. Nearest neighbour join with groups and predicates. In *DOLAP*, pages 39–48. ACM, 2015.
7. B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing interval joins on map-reduce. In *EDBT*, pages 463–474, 2014.
8. C. Date, H. Darwen, and N. Lorentzos. *Temporal Data and the Relational Model.* Morgan Kaufmann Publishers, pages 77-86, 2003.
9. A. Dignös, M. H. Böhlen, and J. Gamper. Temporal alignment. In *SIGMOD*, pages 433–444, 2012.
10. A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *SIGMOD*, pages 1459–1470, 2014.
11. J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *SIGMOD*, pages 683–694, 2004.
12. G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.
13. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.
14. H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *ICDE*, pages 336–344, 1991.
15. L. Isella, J. Stehlé, A. Barrat, C. Cattuto, J. Pinton, and W. Van den Broeck. What's in a crowd? analysis of face-to-face behavioral networks. *Journal of Theoretical Biology*, 271(1):166–180, 2011.
16. M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *SIGMOD*, pages 1173–1184, 2013.
17. J. Kleinberg and E. Tardos. *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
18. N. Kline and R. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231, Mar 1995.
19. H.-P. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, pages 407–418, 2000.
20. T. Y. C. Leung and R. R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *VLDB*, pages 383–394, 1992.
21. W. Li, D. Gao, and R. T. Snodgrass. Skew handling techniques in sort-merge join. In *SIGMOD*, pages 169–180, 2002.

22. A. Monacchi, D. Egarter, W. Elmenreich, S. D'Alessandro, and A. M. Tonello. GREEND: an energy consumption dataset of households in Italy and Austria. In *SmartGridComm*, pages 511–516, 2014.
23. B. Moon, I. Fernando Vega Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *TKDE*, 15(3):744–759, May 2003.
24. D. Piatov, S. Helmer, and A. Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016.
25. A. Segev and H. Gunadhi. Event-join optimization in temporal relational databases. In *VLDB*, pages 205–215, 1989.
26. R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
27. M. Soo, R. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *ICDE*, pages 282–292, 1994.
28. B. Stroustrup. Software development for infrastructure. *IEEE Computer*, 45(1):47–58, 2012.
29. A. Taliun, M. Böhlen, A. Bracher, and F. Cafagna. A gis-based data analysis platform for analyzing the time-varying quality of animal feed and its impact on the environment. In *iEMSs*, pages 1447–1454, 2012.
30. I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *TKDE*, 17(2):271–286, Feb. 2005.
31. J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, Oct. 2003.

A Proof of Equivalence Rule

Now we prove that $\vartheta_F^T(\boldsymbol{R})$ gives the same result as Equation (3).

Lemma 11 *A Temporal Aggregation on an input relation* $\boldsymbol{R}$ *can be decomposed as the full outer join between its* $\mathcal{DIP}$ *partitions:*

$$\vartheta_F^T(\boldsymbol{R}) = \pi_{T,F'}(\boldsymbol{R}_1 ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_2 ⟗_T^{\mathcal{DIP}} \ldots ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_c) \qquad (6)$$

where F' *is an aggregation function that has the same semantic as* F *but applies to columns rather than to rows.*

Proof Proof by induction. We rewrite the sequence of full outer joins in the equivalence rule as:

$$\boldsymbol{Z}_n = \begin{cases} \boldsymbol{R}_1, & \text{if } n = 1 \\ \boldsymbol{Z}_{n-1} ⟗_T^{\mathcal{DIP}} \boldsymbol{R}_n, & \text{if } 2 \leq n \leq p \end{cases}.$$

We check that each conjunction of the definition of $\vartheta_F^T(\boldsymbol{R})$ in Table 2 is satisfied by $\boldsymbol{Z}_n$, with the hypothesis that $\boldsymbol{Z}_{n-1}$ satisfies it:

1. *for each* $w \in \boldsymbol{Z}_n$, $w.T_s$ *and* $w.T_e$ *correspond to the starting or ending point of two tuples* $r, s \in \boldsymbol{R}$, *i.e.,* $w.t_s = (r.T_s \vee r.T_e) \wedge w.t_e = (s.T_s \vee s.T_e)$

$n = 1$ Since $\boldsymbol{R}_1 = \boldsymbol{R}$ then $\forall w \in \boldsymbol{Z}_1 \Rightarrow (\exists r \in \boldsymbol{R} : w.T_s = r.T_s \wedge w.T_e = r.T_e)$,which satisfies condition 1 for $r = s$.

$n > 1$ Remember that $\boldsymbol{Z}_{n-1} ⟗_T \boldsymbol{R}_n$ corresponds to the union of the Join and of the anti-joins between $\boldsymbol{Z}_{n-1}$ and $\boldsymbol{R}_n$, and viceversa. We now show that condition 1 holds for each of those three joins. For $\boldsymbol{Z}_{n-1} \bowtie_T \boldsymbol{R}_n$, given an overlapping pair (z, r), a result interval is $w.T = [max(z.T_s, r.T_s), min(z.T_e, r.T_e))$: $z.T_s$ and $z.T_e$ by hypothesis satisfy condition 1; since $\boldsymbol{R}_n$ is a partition (i.e., a selection) of $\boldsymbol{R}$, then $r.T_s$ and $r.T_e$ also satisfy condition 1 (for $r = s$). For $\boldsymbol{Z}_{n-1} \triangleright_T \boldsymbol{R}_n$, a result interval is $w.T = [z.T_s, z.T_e)$ if no overlapping tuple in $\boldsymbol{R}_n$ exists (which by hypothesis hold condition 1); if a tuple $r_j \in \boldsymbol{R}_n$ exists such that $overlap(z, r_j)$, then a result interval can be *i)* $w.T = [z.T_s, r_j.T_s)$, *ii)* $w.T = [r_j.T_e, z.T_e)$, *iii)* $w.T = [r_j.T_e, r_{j+1}.T_s)$. All these intervals satisfy condition 1. Analogous for $\boldsymbol{R}_n \triangleright_T \boldsymbol{Z}_{n-1}$.

2. *for each* $w \in \boldsymbol{Z}_n$, *there must not exist in* $\boldsymbol{R}$ *a tuple that starts or ends within* $w.T$, *i.e.,* $\forall u \in \boldsymbol{R}\big(overlap(u, w) \leftrightarrow (w.T - u.T = \emptyset)\big)$.

$n = 1$ By definition a $\mathcal{DIP}$ partition does not store overlapping tuples: given $w \in \bar{\boldsymbol{R}}_1$, a tuple $u \in \boldsymbol{R}_1$ with $w.T_s \leq u.T_s \leq w.T_e$ or $w.T_s \leq u.T_e \leq w.T_e$ cannot exist.

$n > 1$ For $\boldsymbol{Z}_{n-1} \bowtie_T \boldsymbol{R}_n$ and $\boldsymbol{Z}_{n-1} \triangleright_T \boldsymbol{R}_n$ condition 2 holds since the timestamp of each result tuple w is a subinterval of a tuple $z \in \boldsymbol{Z}_{n-1}$ (which, by hypothesis, satisfies condition 2). For $\boldsymbol{R}_n \triangleright_T \boldsymbol{Z}_{n-1}$, the timestamp of each result tuple w is the sub-interval of $r \in \boldsymbol{R}_n$ during which no tuple in $\boldsymbol{Z}_{n-1}$ exists. This means that in all previous $\mathcal{DIP}$-partitions no tuple existed during $w.T$. Since the union of all the $\mathcal{DIP}$ partitions gives $\boldsymbol{R}$, then no tuple u exists in $\boldsymbol{R}$ overlapping w other than itself.

3. *for each* $w \in \boldsymbol{Z}_n$, *the set* $\boldsymbol{R}$*' of all tuples valid over* $w.T$ *must be returned in the join result, i.e.,* $\forall u \in \boldsymbol{R}\big(overlap(u, w) \leftrightarrow u \in \boldsymbol{R}'\big)$

$n = 1$ By definition $r.T$ stores the interval of validity of r.

$n > 1$ The full outer join returns, by definition, the tuple of $\boldsymbol{R}_n$ overlapping $w.T$. If no tuple overlapping $w.T$ exists in $\boldsymbol{R}_n$, it returns a *null* value.