Department of Informatics, University of Zürich

BCs Thesis

Temporal Operators on Partitioned Relations

Jerinas Gresch

Freienbach SZ, Switzerland
Matrikelnummer: 12-716-627
Email: jerinas.gresch@uzh.ch

June 15, 2016

supervised by Prof. Dr. M. Böhlen and F. Cafagna

University of Zurich[UZH]

Department of Informatics

Acknowledgments

I would like to thank the whole research group of database technology for giving me the opportunity to write my bachelor thesis. Also, I want to thank Prof. Dr. Michael Böhlen for providing interesting topics such as my thesis. Very special thanks to my supervisor Francesco Cafagna, who supported me with helpful informations.

Abstract

Having an outer and an inner relation, many operators (i.e. joins, anti-joins and aggregation) can be computed. By applying a timestamp $T = [T_s, T_e)$ on operators, we refer to them as temporal operators. Our main goal is to compute these temporal operators efficiently. So far, a technique called *DIP* (Disjoint Interval Partitioning) has been developed, which partitions the relation into sets, in which no tuple overlaps.

The goal of this thesis is to implement a partitioning algorithm and temporal operators (such as join, anti-join and aggregation) applied to a pair of partitions.
We have furthermore optimized the merge of the partitions by passing multiple partitions at the same time.

Contents

Abbreviations

DIP	Disjoint Interval Partition
R	Relation
r	tuple from an outer partition
s	tuple from an inner partition
s.X	Lead of the tuple *s*
r.X	Lead of the tuple *r*
internal	Position in the array of outer partitions
parallel	Number of outer partitions to call in parallel
DIPMerge	Function that applies a temporal operators

1. Introduction

1.1 Overview

Applying operators such as joins, anti-joins and aggregations is a common thing in database system. In the relations that we are going to use in this thesis, 2 types are important. A tuple is composed of 2 timestamps, a start time and an end time. In addition, each tuple has its own unique number as an identifier. We talk about temporal operators when we apply an operator on such relations. Imagine a real-life database model of an airport. The time period when a plane is flying has to be stored. When we are interested in the set of planes flying at the same time, we can apply a temporal join. Each plane has a start and end time, which indicates the time it flies. Whenever these periods overlap with another planes' flying period, we get periods where the planes fly at the same time. The goal of the thesis is to implement efficient algorithms applying temporal operators.

1.2 Approach

As already mentioned, a relation can have many tuples that overlap with each other. 2 Tuples overlaps when the start time of one tuple is before the end time of another tuple and the end time of the first tuple is after the start time of the second tuple. Applying temporal operators on such intervals can be very inefficient (this is shown in section 3.1). Therefore, the relations are split into partitions. In a partition, no tuple overlaps with another tuple.
Generally, two relations are used in this thesis: an outer and an inner relation. Each relation will be split into partitions. Having many outer and inner partitions, the temporal operators are applied on them. When we talk about *DIP*, we refer to the *Disjoint Interval Partitioning*[1]. The initial goal was to implement the *DIP*, where the number of partitions does not have to be fixed, and the *DIPMerge*, which is a unique function that computes temporal join, anti-join and aggregation on the partitions. The partitioning will be discussed in section 2 and the temporal operators in section 3. Both of them are implemented for memory and disk computation. For disk, every relation and partition is stored in a file, on which we use the operators. There won't be any actual code presented in this thesis, only some pseudo code which I consider to be important. The specific code is available in the attached CD.
For memory, we only use the files once, namely when we read the two relations the first time. The *DIPMerge* function will be presented in section 3. Since I had reached these goals in just 3 months, I have done additional optimizations such as processing many partitions simultaneously. In section 4, this improvement is shown. Following that, a comparison between the results of the original algorithm and the results of the improved algorithm is presented. This new implementation can be used for speeding up the join in a worst case scenario and for speeding up the anti-join in any scenario.

2. Partitioning

2.1 Structure

The implementation is written in C. My supervisor, F. Cafagna, provided me with a framework to work with. I used Eclipse as development environment and the code is compiled with a makefile and invoked through the command line as shown at the end of the thesis. Basic requirements are already covered in the provided framework, such as the structure of a tuple. A tuple consists of 2 timestamps of the type long long and one data variable that represents the value of its tuple.
The text files are relations which include 3 different columns. In our approach, the first column is the start time, the second column is the end time, and the third column represents the value of the tuple. A relation of 5 tuples looks like this:

```
19770820    19770821    1
19770825    19770901    2
19770830    19770911    3
19770920    19770921    4
19770928    19770929    5
```

Listing 1: Input file

Each column is separated with a tab to the other column. In this example, the first tuple starts at August 20, 1977, ends at August 21 1977 and has the value 1. Notice that the second tuple overlaps with the third tuple. The second tuple ends at September first, 1977, but the third tuple already starts at August 30, 1977. Those two tuples are both valid in the period from August 30 until September 1.
A file can easily be read and iterated line by line using the function *fscanf*. Knowing that each column is separated with a tab, a line can be fragmented and assigned to the three variables. We only check if the start time of a tuple is smaller than the end time to guarantee that the relation is consistent. For an execution on disk, the tuples are stored on a binary file, for a memory execution, the tuples are stored on a list.
The next step is to take two relations and create the disjoint interval partitions. F. Cafagna had already started with a prototype of the partitioning. In his approach, he fixed the amount of partitions a relation can have (he stores the partitions in an array). However, there might be relations with more partitions than the determined upper limit. This is where my implementation starts.

2.2 Dynamic partitioning

The *DIP* algorithm consists of the following steps:

1. Fetch a new tuple from the relation *R*
2. Check if the tuple overlaps with the last tuple of the first partition *L.head*
   - If they don't overlap, append the tuple to *L.head*
   - If they overlap, sort the list of partitions by the tails' end time in ascending order
3. Check if the tuple overlaps with the last tuple for the new first partition *L.head*
   - If they don't overlap, append the tuple to *L.head*
   - If they still overlap, create a new partition, add the tuple to this partition and let this partition be the new first partition
4. Repeat this algorithm until every tuple from *R* has been fetched

The partitioning function expects a relation as a parameter. For our algorithm, we sort the list *R* after start time in ascending order. Since we do not want to store the partitions in arrays anymore, the partitions are also stored in a list. Therefore, we need to modify the list properties first. A partition consists of a block of tuples, where tuples are consecutive. In addition, we add a pointer to partition, so that each partition points to the next partition. The first partition *L.head,* where *L* is an element of the partition list R_1, needs to be initialized with the first tuple *r* from the relation *R* and points to no other partition so far. Now, we iterate through the whole relation. We first fetch a tuple *r* from *R.* This tuple *r* is compared with the last tuple in the first partition *L.head* indicated as *L.head.last* in line 5. In case they do not overlap, *r* will be appended to *L.head*. The condition of an overlap is defined as following: the start time of r_1 is smaller than the end time of r_2 and the end time of r_1 is bigger than the start time of r_2.

```
1  Sort(R) by T_s, (T_e − T_s)_desc ;
2  R_1 ← ∅ ;
3  L = <R_1> ;
4  while (r = fetchtuple(R)) ≠ null do
5      if L.head = ∅ ∨ disjoint(L.head.last, r) then
6          Add r to L.head ;
7      else
8          l = L.head ;
9          while l.next ≠ null ∧ l.last.T_e > l.next.last.T_e do
10             Swap(l, l.next) ;
11         if overlap(L.head.last, r) then
12             R_{len(L)+1} = ∅;
13             L = prepend(R_{len(L)+1}, L) ;
14         Add r to L.head ;
15 return L;
```

Figure 1: DIP pseudocode

If they do overlap, we need to check the other partitions. Therefore, we need to iterate through the whole list of partitions. A temporary partition *l*, pointing to an element of *L*, is initialized as *L.head*. Now, we are able to access every partition list *L* by assigning *l* to *l.next*, since every list R_i in *L* points to the next R_{i+1}. Since there is only one partition so far, we have to create a new partition for the current *r.* We create a new list and assign it to the first partition. In addition, the new first partition points to the old partition and *r* can be added to the first partition.
Afterwards, a new *r* is fetched from *R*. Again, *r* is compared with the last tuple *L.head*. In case they overlap, we need to consider the other partitions. Therefore, we reorder the list of partitions. The partition with the smallest end time needs to be the first partition. If the tail of any partition does not overlap with *r,* it will certainly be the tail of the partition with the smallest end time. Remember that *R* is sorted after ascending start time. At line 10, the current temporary partition *l* is advanced with *l.next*. In this case, the end time of the last tuple of *l.next* is smaller than the last tuple of *l.* Having the partition with the smallest end time at the tail as the first partition, *r* is again compared with the last tuple of the new first partition. If they still overlap, a new partition has to be created. Otherwise, we append *r* to the first partition and fetch a new *r* from *R*. This algorithm goes on until every *r* from *R* is fetched. The function returns the first partition, which gives us access to the whole partition list.
In figure 1, the pseudocode of the partitioning is shown.
After the *DIP*, we have a list of partitions. Inside each partition, no tuple is overlapping. This is shown in figure 2. In this figure, a tuple r_i is represented as a black line. Interpret a line as an interval, which has a start and an end time. As we can see, all the tuples inside one partition are not overlapping.

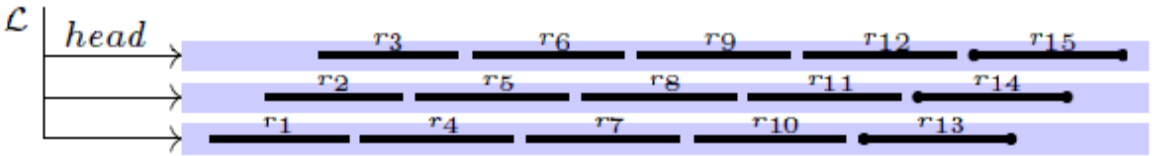


Figure 2: List of partitions

2.3 Partitioning for memory

For memory, the partitions are built as a list. This list allows us to create as many partitions as we need and we are not bound to a limit anymore (as we had it before when the partitions were stored in arrays). Each partition contains a pointer to the next partition and the blocks, in which the tuples are stored. A block has fixed memory size allocated. The head of the partition list is returned by the *DIP* function.

2.4 Partitioning for disk

The process flow for disk partitioning is the same. The main difference to the memory partitioning is that the partitions are not stored in lists but in file-nodes which behave similarly. A file-node mainly consists of a pointer to the next file-node, a pointer to a file, the name of the file and an indicator to see whether a file is open or not. The function does not expect a list as parameters, since the input relation is on disk. Thus, the function requires a relation of the type heap as a parameter. Through a heap data structure, we can read data very efficiently and write it into a binary file. On the one hand, binary files are more suitable for storing data than common text files. On the other hand, if you open a binary file, the symbols shown are not interpretable, whereas common text files, i.e. relations in text files, display the actual relation. To read from a binary file, the data has to be stored in a heap again. As already said, file-nodes behave similarly to list nodes, used in memory partitioning. Also, the same pseudo code is applied again. One point that needs to be highlighted is the creation of a new partition. Creating a new partition goes along with creating a new file-node. This file-node consists of an actual file that needs to be created and opened. When a file is opened, another file needs to close. In addition, we can only write into an open file. The first partition-file is always open, whereas all other partition-files are closed. This process is very inefficient, which I noticed when I ran my tests. If a relation has many partitions - independent of the size of the relation - the partitioning is very slow, since many file openings and closings are taking place. Therefore, an improvement of the disk partitioning is provided in section 2.5.
The actual files of the partitioning are stored in the folders */Partitions1* and */Partitions2*. The first folder stores the partitions from the outer relation, and the second folder stores the partitions from the inner relation. As in section 2.3, the *DIP* function returns the head of the partition list, which gives us access to all partitions.

2.5 Solution for the file I/O bottleneck

As already mentioned, opening and closing files is very expensive. Since a file is closed before the partition list makes the partition with the smallest end time the first partition, and opening that file afterwards, there will be a lot of file openings and closings. Therefore, the algorithm needs to be reworked. It is not necessary to close every file before we enter the ordering section. In fact, every file could be kept open. Unfortunately, there is a limit of open files depending on the current operating system. Whenever we had more files open than the operating system allows, the algorithm fails. The majority of the operating systems allow 256 open files by default. On Mac OS, this can be changed through the command line by typing *ulimit -n 1000*, where 1000 is the amount of files that can be open at once on my Macbook Pro. In our approach, a constant *k* is defined which limits the amount of open files. At the moment, *k* is set to 240. If there is any need to change this constant, modify the variable *k* in the *run_Partitioning_Ordered_DISK.* Now, we only need to keep track of how many files are currently open. Whenever we reach 240, the algorithm starts closing the files again. This happens while we order the partitions. When the 240^{th} partition is swapped with the 241^{th} partition, partition 240 is closed and partition 241 is opened. This approach allows us to create up to 240 partitions very efficiently (everything above slows the algorithm down a lot).

3. Temporal operators

3.1 Temporal join

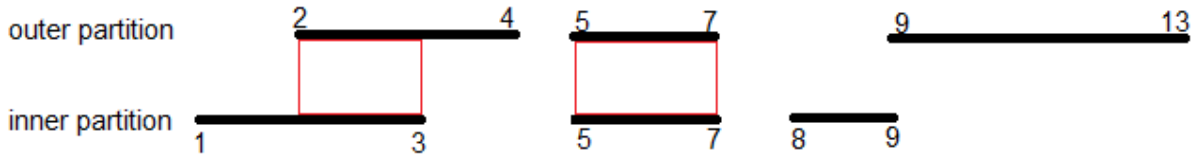
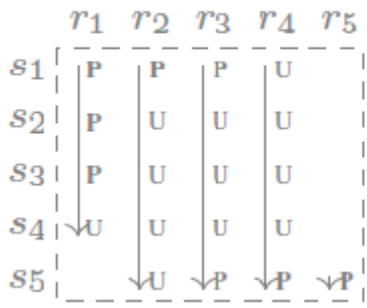


Figure 3: Overlap conditions

Consider the tuples in figure 3. A join between two tuples is defined as an overlap between these two tuples. A tuple is represented as a line and the two numbers of a tuple indicate its start and end time. The first tuple of the outer partition starts at 2 and ends at 4. The first tuple of the inner partition starts at 1 and ends at 3. The tuples overlap between 2 and 3.
Joining two relations without the *DIP* is expensive. Having a join applied on two relations where the tuples in the relation overlap with each other forces the algorithm to do backtracking. The first tuple *r* from *R* needs to be compared to all tuples in *S* until the start time of *s* is bigger than the end time of *r*. Since the relation is sorted, there will be no productive join match with any other tuples following *s*. If there was a join match, for the next tuple *r*, we have to consider all the tuples *s* back to the first join match. Consider the tuples r_2 and r_3 in Figure 4. Tuple r_2 has to be compared to all tuples in *S*. As we can see, there are 4 unproductive join matches. For tuple r_3, all tuples from *S* have to be checked again because there was a join match between r_2 and s_1 and therefore, we need to *backtrack* the tuples from *S*. Now, we get 3 unproductive join matches. This is an example we try to resolve in this thesis. With partitions where the tuples do not overlap, the backtracking can be avoided.

R	T	#	$
r_1	$[1, 5)$	1	80
r_2	$[6, 8)$	1	60
r_3	$[7, 10)$	2	60
r_4	$[10, 12)$	3	70
r_5	$[10, 12)$	1	80



S	T	#	$
s_1	$[0, 10)$	8	60
s_2	$[1, 2)$	2	70
s_3	$[3, 4)$	2	80
s_4	$[5, 6)$	3	60
s_5	$[8, 11)$	2	90

Figure 4: Join of two overlapping relations

In figure 5, a join between four partitions is performed. Note that there are many less unproductive matches than in figure 4. Each partition of *R* is joined with each partition of *S*. Consider the join between R_1 and S_2. The tuples s_2 *and* s_3 resulted into a productive join match with r_1, but not s_4. All of the previous productive join matches can be ignored for the next tuple r_2, since there was an unproductive join match between r_1 and s_4. Therefore, we can immediately start at s_4.

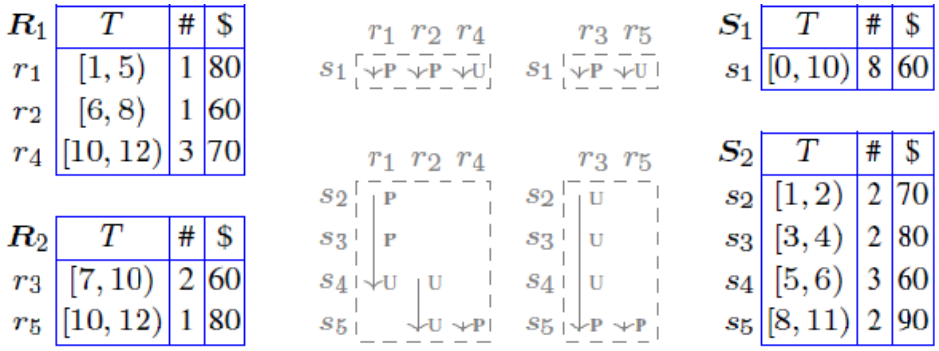


Figure 5: Temporal join of four partitions

In order to perform a temporal join, every outer partition R_i must be joined with every inner partition S_j. The merge function *DIPMerge* is implemented for a memory and a disk version and returns a counter of the join matches. Outside the function, we catch the counter of joins and call the function for every outer and inner partition. Since the function expects only one partition of each relation, every partition R_i has to be scanned for every partition S_i. Thus, the *DIPMerge* function for the partition list R_n and S_m, where *n* and *m* represent the number of partitions of each relation, will be called *n*m* times. Notice that an outer partition has to be scanned multiple times. Therefore, an enhanced algorithm is provided and presented in section 4. In figure 6, the invocation of the *DIPMerge* function is shown.

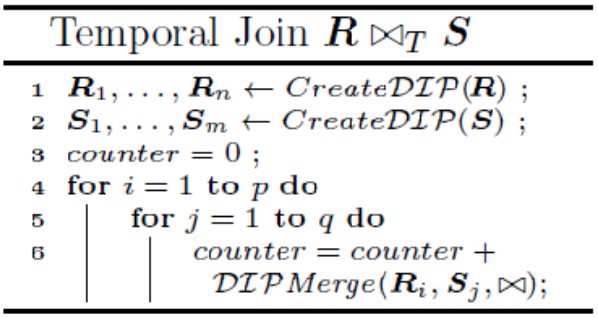


Figure 6: Temporal join

3.2 DIPMerge for a temporal join

Inside the *DIPMerge* function, 5 significant variables are used:

- A counter, which increments whenever a join match appears.
- A tuple *r*, in which a tuple from the outer partition will be stored.
- A tuple *s*, in which a tuple from the inner partition will be stored.
- A variable *r.X*, which stores the interval between the current and the last fetched tuple of *s*. This interval can be seen as a lead.
- A variable *s.X*, which stores the interval between the current and the last fetched tuple of *r*. This interval can be seen as a lead (Red interval in figure 8).

The last two variables are mainly meant for the anti-join. They are initialized with $[-\infty, s.T_s)$ for *s.X* and $[-\infty, r.T_s)$ for *r.X*. *r* is initially fetched from the outer partition and *s* is fetched from the inner partition. Now, the algorithm is channeled into a while loop, where it processes every tuple. Since there are already two accessible tuples, the algorithm first checks if they meet the join condition. The condition checks that both tuples are not *null* and that they overlap. If so, the counter is incremented. Otherwise, it will count as an unproductive match. The next step is to decide which tuple, *r* or *s*, will be advanced. There are two cases where *r* is advanced as long as *r* is not *null*:

1. *r.Te* is smaller than *s.Te* or
2. *s* is *null*

In the first case, tuple *s* ends later than *r* ends. In the second case, all tuples from *S* have already been fetched so we need to fetch the remaining tuples of *R*. Even though *s* is null and cannot result in a join with any tuple *r*, it can result in an anti-join match, which we will see later. These two cases are shown in figure 7. Whenever this appears, *r* will be advanced. This condition is shown in the pseudocode of the *DIPMerge* function in figure 9 in line 15.

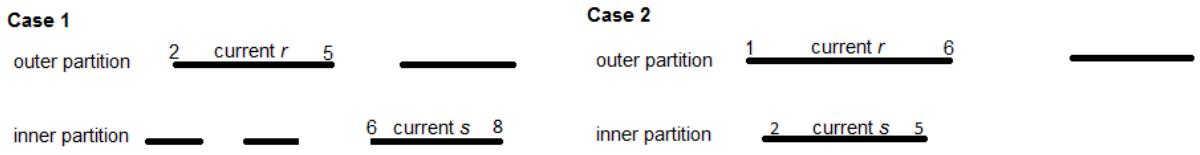


Figure 7: Conditions for advancing r

There was a small bug in the pseudocode of my supervisor F. Cafagna. His condition in line 15 was implemented as follows:

if(!null(r) && (s==NULL || (r.T_e *<= x.*T_e*)*

This resulted in less join matches since some tuples of *r* will not be advanced when needed. In figure 8, this case is shown. The red line represents the lead *s.X*. At the current *r*, the end time of the lead is smaller than the end time of *r*. Although *r* should be advanced, the algorithm does not see that there is still a remaining tuple of *r* in the same interval of the rather big tuple *s* and *r* will not be advanced.

Figure 8: Bug

```
DIPMerge(O, I, Operator, parallel, s.X, r.X)
1  s ← fetchRow(I);
2  r ← fetchRow(O) ;
3  s.X = [−∞, s.T_s];  //tail of s;
4  r.X = [−∞, r.T_s)  //tail of r;
5  while !null(r) ∨ !null(s) do
6      if Operator ∈ {▷, ⋉⊏} then
7          if len(s.X) > 0 ∧ overlap(r.T, s.X) then
8              countJoin + + ;
9      if Operator = ⋉⊏ then
10         if len(r.X) > 0 ∧ overlap(s.T, r.X) then
11             countJoin + + ;
12     if Operator ∈ {⋈, ⋉⊏} then
13         if overlap(r.T, s.T) then
14             countJoin + + ;
15     if (!null(r) ∧ (null(s) ∨ r.T_e < s.T_e)) then
16         if r.T_e ≥ r.X_s then r.X = [r.T_e, ∅);
17         r ← FetchRow(O) ;
18         if !null(r) then r.X = [r.X_s, r.T_s) else
             r.X = [r.X_s, ∞);
19     else
20         if s.T_e ≥ s.X_s then s.X = [s.T_e, ∅);
21         s ← FetchRow(I) ;
22         if !null(s) then s.X = [s.X_s, s.T_s) else
             s.X = [s.X_s, ∞);
23 return  countJoin + +
```

Figure 9: DIPMerge pseudocode

In my algorithm, the lead of *r*, namely *r.X*, will be updated afterwards. If the end time of the previous *r* is bigger than the start time of *r.X*, it will be assigned to the start time of *r.X*, and the start time of the current *r* is assigned to the
end time of *r.X* (if r is not null). When *r* is *null*, the algorithm reached the last tuple of the current partition and the end time of *r.X* is set to infinite.
In all other cases, *s* will be advanced. As *r.X* was updated after *r* had been advanced, the same happens to the lead of *s,* which is *s.X*. Note that this procedure lacks backtracking and many unproductive matches are avoided.
At the end of the function, the counter is returned. *DIPMerge* is mainly the same on memory and disk. For disk, the files are opened before it goes into the *DIPMerge* function. As parameters, the two partitions of the type heap are passed. After the function, the inner file is closed and the outer file initialized again, since every outer partition must be rescanned for every inner partition. Since file I/O is very expensive, an improved algorithm is provided in section 4.6 and 4.7.

3.3 Temporal Anti-join

A temporal anti-join between two relations *R* and *S* returns all intervals in *R* where no tuple in *S* is valid. It is computed by joining each outer partition from *R* with the relation *S*. Outside the *DIPMerge* function, all outer partitions are scanned for the whole relation *S,* as you can see in figure 10. Having a tuple *r* from an outer partition, a join match appears if an interval between two tuples in *S* is overlapping with *r*. The algorithm already stores this interval between two tuples. It is the lead of a tuple, *s.X* and *r.X*. For the anti-join, the tuple *s.X* is used, since we want to know the interval between two tuples in *S*. Whenever an overlap between *r* and *s.X* appears, it counts as an anti-join match.

Temporal Anti-join

$\boldsymbol{R} \triangleright_T \boldsymbol{S}$

```
1  R_1, ..., R_n ← CreateDIP(R)
   counter = 0 ;
2  for i = 1 to p do
3  |   counter = counter +
   |     DIPMerge(R_i, S, ▷) ;
```

Figure 10: Temporal anti-join

The anti-join condition contains two sub-conditions:

- Tuple *r* overlaps with tuple *s.X* and
- the length of *s.X* is greater than zero.

This condition is shown in line 7 in figure 9.
The process of deciding when to advance which tuple, *r* or *s*, is the same as for the join, described in section 3.2.

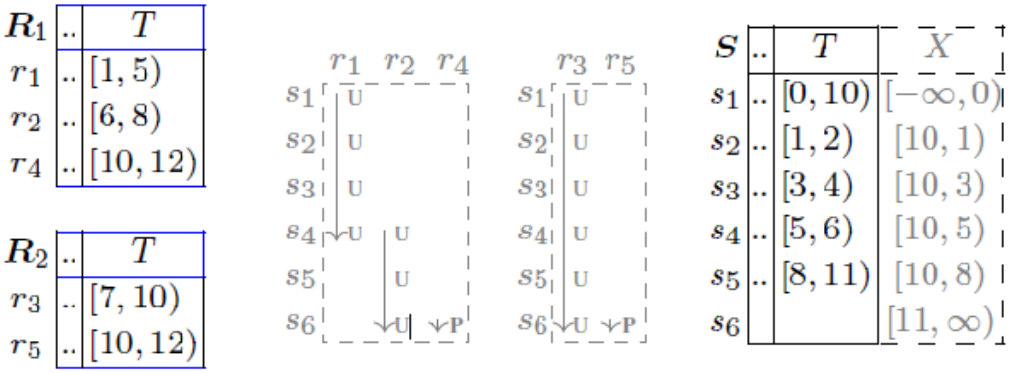


Figure 11: Temporal anti-join and lead of S

In figure 11, there is an example of how the anti-join looks like. In relation *S*, each lead *s.X* is also shown. As you can see, no tuple in *S* overlaps with the lead because it starts after the previous tuple *s* ends and ends before the current tuple *s* starts. During this period, no tuple of *s* is valid. If we get any join matches between *r* and these intervals, it is considered to be an anti-join match. From s_2 until s_5, we have this circumstance that no lead exists. The first tuple s_1 has an end time of 10, which is bigger than all end times until s_5. This behavior is checked in line 20 in figure 9. We always keep the biggest end time of s_i as the start time of *s.X*. However, if the start time is bigger than the end time, there is no way that any tuples overlap with this *s.X,* which is what we want. The situation changes here when we reach the last tuple of *S*. Then, *s.X* never ends. The lead goes to infinite. If any tuple in *R* is valid during this period, it can be counted as an anti-join match. This is the case for the tuple r_5. It is valid until 12, which is later than *s.X* [11,∞) starts.

3.4 Temporal Aggregation

The temporal aggregation returns a set of maximal intervals in which tuples are valid. The aggregation refers to only one relation *R*. Since the tuples overlap in this relation, multiple values of tuples can be valid at the same time. The goal is to identify which values are valid at which time intervals, shown in the tables 1. The column *$* represents the value of a tuple. This can be done by joining all partitions. But we also need to know when only one value for a certain time period is valid. This can be done by combining a join with two anti-joins. So far, we only had the anti-join for *R* and *S* in which no tuples in *S* overlap with the partitions of *R*. Additionally, the opposite is required as well. In conclusion, for every partition R_i in *R*, a join, an anti-join between partition R_i and R_{i+1} and an anti-join between R_{i+1} and R_i will provide all the needed intervals, also known as a full outer join. When we use a full outer join on two partitions, the resulting intervals, which can be seen as a temporary, must be stored. This temporary partition will again be full outer joined with the next outer partition and so on. Outside the *DIPM*erge function, every outer partition is iterated. The *DIPMerge* function is invoked between a temporary partition, in which the intervals of the previous *DIPMerge* run are stored, and the next outer partition. Initially, the first partition is assigned to this temporary partition. Afterwards, whenever a *DIPMerge* starts, this partition will be overwritten with the current join matches.
Now, only the second anti-join needs to be implemented in order to perform a full outer join. The condition for an anti-join between *S* and *R*, where no tuples from *R* are valid, contains two sub-conditions:

- Tuple *s* overlaps with the lead of *r,* which is stored in *r.X* and
- the length of *r.X* is greater than zero.

This condition is shown in line 9 in figure 9. Notice that the two other join conditions in line 12 and 6 also contain the full outer join variable, which means that all three conditions will be checked. After the last iteration of R_{last}, the tuples from the full outer join are stored in the list *Z.* An additional projection (i.e. scan) of the result is needed for computing the result of the aggregation. This can done by evaluating an aggregation function (e.g. min) on the data field where all values have been inserted.

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

```
1 R_1, ..., R_p ← CreateDIP(R) ;
2 Z ← R_2 ;
3 for i = 2 to p do
4 |   Z = DIPMerge(Z, R_i, ⋈^T) ;
```

Figure 12: Temporal aggregation

R_1

T	*$*
[1, 5)	10
[6, 10)	20

R_2

T	*$*
[3, 7)	30
[8, 11)	40

Z_1

T	*Sum($)*
[1, 3)	10
[3, 5)	40
[5, 6)	30
[6, 7)	50
[7, 8)	20
[8, 10)	60
[10, 11)	40

Tables 1: Creation of Z_1

In figure 13, there is a representation of how the list *Z* is created. First of all, we have two partitions R_1 and R_2 on which we apply a full outer join. For every single join match (join, anti-join between R_1 and R_2, anti-join between R_2 and R_1), a new tuple is created and added to Z_1. Z_1 will be full outer joined with R_3, and a new list *Z* evolves. In this particular example, a tuple is always valid during the considered period. It is also possible that no tuple from R_i and Z_i is valid at the same time. So don't get confused by the list *Z* always being valid.

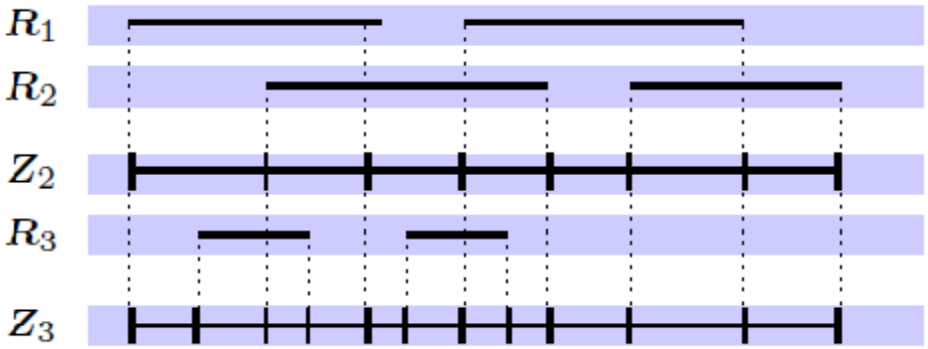


Figure 13: Creation of the Z-list

Now, all three goals for this thesis are implemented. A temporal join, anti-join and aggregation. We do not compare the performance of these algorithms with any other temporal operator approaches, but an enhancement of this implementation will be shown. The following sections describe how our approach is being improved.

4. M-Way *DIPMerge*

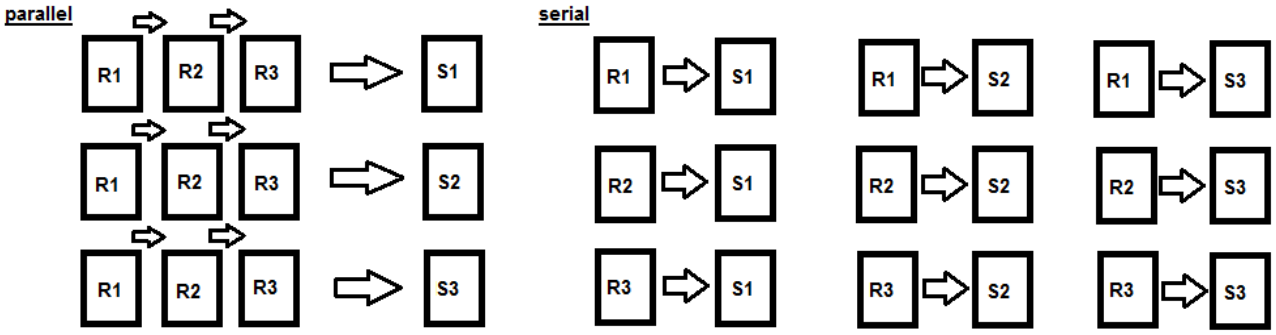


Figure 14: Parallelizing 3 outer partitions

So far, no performance improvement has been done. A problem that can be recognized very quickly is the multiple scanning of the same inner partition. Every time a new outer partition is fetched in the *DIPMerge* function, an inner partition is scanned multiple times, until every outer partition has been compared to the inner partition. Since nothing is changing in these inner partitions, there needs to be a way to avoid these redundant processes. Therefore, a new merge function has to be implemented. Contrary to the old merge function, this one takes multiple outer partitions as arguments instead of only one. Thus, if all outer partitions would be passed in this function, every inner partition has to be scanned only once. We are talking about parallelizing the outer partitions because they are all fetched at the same time. Inside the function, the fetched tuples *r* from each passed outer partition is stored. In addition, there is also an array for the lead of *r,* in this case an array for *r.X*. For this approach, only small changes, compared to the old *DIPMerge* function, have to be made. For once, all the used *r* and *r.X* tuples are going to be replaced with the arrays where r[i] . Initially, a new variable called *parallel* is required. We need to know how many outer partitions we want to call in parallel. This variable is also used for initializing both arrays. If there are *n* outer partitions called in parallel, the arrays for tuple *r* and *r.X* only need to store *n* tuples. Unfortunately, we do not know how many outer partitions there are at the time we call a new temporal operator. However, the number of partitions which are going to be called in parallel is limited to the amount of outer partitions that are stored during the *DIP* process. In the merge function, we iterate through the array step by step. A variable called *i* indicates in which outer partition we are at the moment. It increments every time the conditions for advancing *s* (shown in figure 15 in line 20) are met. For each *s* tuple, we advance the current outer partition until all tuples overlapping with s are found. We then, instead of advancing s, process the next outer partition by incrementing *i*. Only after all outer partitions are processed (if *i*= parallel -1), we advance *s* (and set *i*=0).
A join match is outputted as an incrementation of the *countJoin* variable shown in line 13.
An anti-join match is also outputted as an incrementation of the countJoin variable shown in line 10.
In figure 14, there is an example where 3 outer partitions are joined with 3 inner partitions. If they are merged serially, an inner partition is scanned 3 times per outer partition. Overall, there are 9 scanned inner partitions. If the partitions are merged in parallel with 3 outer partitions simultaneously, every inner partition is scanned once and overall, there are 3 scanned inner partitions.

$\mathcal{DIP}Merge(\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_p\}, \boldsymbol{S}, Operator, parallel, s.X, r.X[])$

```
1  for i = 0 to parallel do
2  |  r[i] ← fetchRow(R_i);
3  |  r.X[i] = [−∞, r[i].T_s];  //tail of r[i]
4  s ← fetchRow(S) ;
5  s.X = [−∞, s.T_s)  //tail of s;
6  i = 0 ;
7  while !null(r[i]) ∨ !null(s) do
8  |  if Operator ∈ {▷} then
9  |  |  if len(s.X) > 0 ∧ overlap(r[i].T, s.X) then
10 |  |  |  countJoin + + ;
11 |  if Operator ∈ {⋈} then
12 |  |  if overlap(r[i].T, s.T) then
13 |  |  |  countJoin + + ;
14 |  if (!null(r[i]) ∧ (null(s) ∨ r[i].T_e < s.T_e)) then
15 |  |  if r[i].T_e ≥ r.X[i]_s then r.X[i] = [r[i].T_e, ∅);
16 |  |  r[i] ← FetchRow(O_i) ;
17 |  |  if !null(r[i]) then r.X[i] = [r.X[i]_s, r[i].T_s) else
   |  |    r.X[i] = [r.X[i]_s, ∞);
18 |  else
19 |  |  if i < parallel − 1 then
20 |  |  |  i + +;
21 |  |  else
22 |  |  |  if s.T_e ≥ s.X_s then s.X = [s.T_e, ∅);
23 |  |  |  s ← FetchRow(I) ;
24 |  |  |  i = 0;
25 |  |  |  if !null(s) then s.X = [s.X_s, s.T_s) else
   |  |  |    s.X = [s.X_s, ∞);
26 return  countJoin + +
```

Figure 15: DIPMerge optimized

4.1. Optimized Temporal join

In figure 15, the new *DIPMerge* is shown. The argument *{R₁,.., Rₚ}* and *S* represents the list of outer and inner partitions. The parameter *parallel* indicates how many outer partitions are called in parallel. From line 1 to 3, the first tuple from each outer partition is fetched in the array called *r*, and for every tuple in *r*, a lead will be created and stored in the array called *r.X.* The amount of outer partitions to process depends on the parameter *parallel*. A lead consists of a start and end time, which will initially be minus infinite and the start time of the corresponding *r[i]* tuple. Tuple *s* and lead *s.X* are not changed regarding the original *DIPMerge* function. The two leads, *r.X and s.X* are passed as parameters to *DIPMerge*. They are initially empty, but the memory allocation for them has already been made outside the function. This is done because of efficiency reasons. If this allocation took place inside the *DIPMerge* function, every time we call

the function, we would allocate new memory for those tails. The variable *i*, which indicates in which outer partition the algorithm is at the moment, is set to zero because the first tuple of the first outer partition is stored at position zero in array *r[i]*. At this point, the algorithm is ready to enter the loop where it iterates through all tuples and all outer partitions respectively. The join condition is shown in lines 11 and 12. In contrast to the original *DIPMerge* function, instead of comparing tuple *r* to tuple *s*, the current tuple in the array *r[i]* is compared to *s*. Afterwards, we need to decide which tuple should be advanced. If the condition in line 14 is true, we advance *r[i]*. If the condition for the current *i* is not true, we try to advance *s*. However, there might be other outer partitions that meet the condition at line 14. Therefore, *s* will only be advanced if all current tuples of the array *r[i]* are checked. The *i* variable gives us information about which partition we are currently checking. Since a tuple of each outer partition is stored in the array *r,* we can compare *i* to the *parallel* variable, which indicates how many outer partitions there actually are. If *i* is equal *parallel-1*, all the fetched tuples of each outer partition are checked and *s* can be advanced. In addition, *i* is set to zero, meaning that we start with the tuple of the first outer partition again. We need to subtract minus 1 from *parallel,* since an array starts at zero. If not all outer partitions have been checked, which is shown at line 19, the variable *i* is incremented which goes along with skipping to the tuple of the next outer partition.
The *DIPMerge* function is called *n* times, where *n* is the number of inner partitions if the value of *parallel* is the amount of outer partitions. If we only pass half of the maximum number of outer partitions as arguments, every inner partition will be scanned twice. One time for the first half of outer partitions, and one time for the second half of outer partitions. In figure 16, the invocation of the *DIPMerge* function is shown. Note that we pass *k* outer partitions, which is the amount of outer partitions we want to call in parallel.

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

```
1  R_1, ..., R_p ← CreateDIP(R)
2  S_1, ..., S_q ← CreateDIP(S)
3  counter = 0
4  for i = 1 to p do
5  |  for j = 1 to q do
6  |  |  k = min(k, numOuterPartitions)
7  |  |  counter = counter +
   |  |    DIPMerge({R_i, ..., R_k}, S_j, ⋈)
8  |  |  i = k
```

Figure 16: Invocation of optimized DIPMerge

5. Performance tests

5.1. Performance tests for temporal join on memory

So far, the partitions are stored in a list. Lists are not optimal when it comes to memory allocation. One node can be far away from another node, which leads to bigger jumps in the memory and this has a direct influence on the performance. Even though our parallelized algorithm works with nodes, we are interested in a data structure where the tuples are allocated close to each other. Arrays are very suitable for this. Therefore, the tuples are stored in arrays after the *DIP.* Each partition is an array, and those arrays are passed as parameters.
As we are interested how much faster the parallel merge function is, the time for multiple scenarios has been recorded.
For *parallel*=1, every inner partition has to be scanned with every outer partition. For *parallel*=*numOuterpartitions*, every inner partition has to be scanned once. In this case, however, we still need to scan all outer partitions for each inner partition. Let us take the relation from figure 14. There are 3 inner and 3 outer partitions. If they are merged sequentially, each inner partition is scanned 3 times, and each outer partition is scanned 3 times, which results in 18 scans overall. If they are merged parallelly, it only takes 12 scans: 9 for the partitions which are called in parallel and 3 for the inner partitions. As one can imagine, the runtime of a merge with 12 scans cannot be more than twice as fast as the runtime of a merge with 18 scans. During the tests, we noticed that the scan of small partition costs almost nothing. The costs evolve through jumps between the partitions, when some partitions are in the cache and others are in the memory. Hence, big partitions will not show any big runtime differences since all of the partitions are stored in the memory. However, F. Cafagna and I found some scenarios where the caching effect is highly noticeable and the runtime improves dramatically. In this scenario, the relation does have a lot of partitions. Therefore, all tuples in the relation overlap with each other. We can call this a worst case scenario since every tuple needs a partition for itself. The first worst case relation that we used had 30k tuples, which resulted in 30k partitions. In order to see performance differences, the time was recorded for a join. We tested different configurations of the variable *parallel,* which indicates how many outer partitions are called in parallel. The following values for the variable were passed:

parallel := {1, 10, 20, 30, 50, 60, 100, 1000, 10000, 20000, 30000}

Having value 1 as the *parallel* parameter will make the algorithm behave like there is no parallelism. On the contrary, by passing 30k outer partitions, an inner partition has to be scanned only once.
Now, we need to know the cost of the *DIPMerge*. In formula 1, the costs are shown. B_r *and* B_s stand for the number of blocks. *P* represents the amount of outer partitions we want to call in parallel and *M* is the size of the cache per partition. Once some partitions are loaded into the cache, they will be reused afterwards and almost no costs emerge. Therefore, we can subtract *M* from the block of each outer partition. The expression (B_r *- M)* approximates to zero for small partitions.

$$cost(DIPMerge) = \frac{numOuterpartitions}{n} * B_s * numInnerpartitions + (B_r - M) * numOuterpartitions$$

formula 1: Cost of a temporal join

As we can see in figure 17, the runtime for a serial execution (*parallel* =1) takes 195 seconds. What happens is that an outer partition gets fetched into the cache, as well as an inner partition. Those two are compared with the *DIPMerge* function. After this, the next inner partition gets loaded into the cache and compared with the same outer partition. This process goes on for each inner partition. After each inner partition has been compared to the first outer partition, the next outer partition is loaded into the cache and the process starts again with the first inner partition.

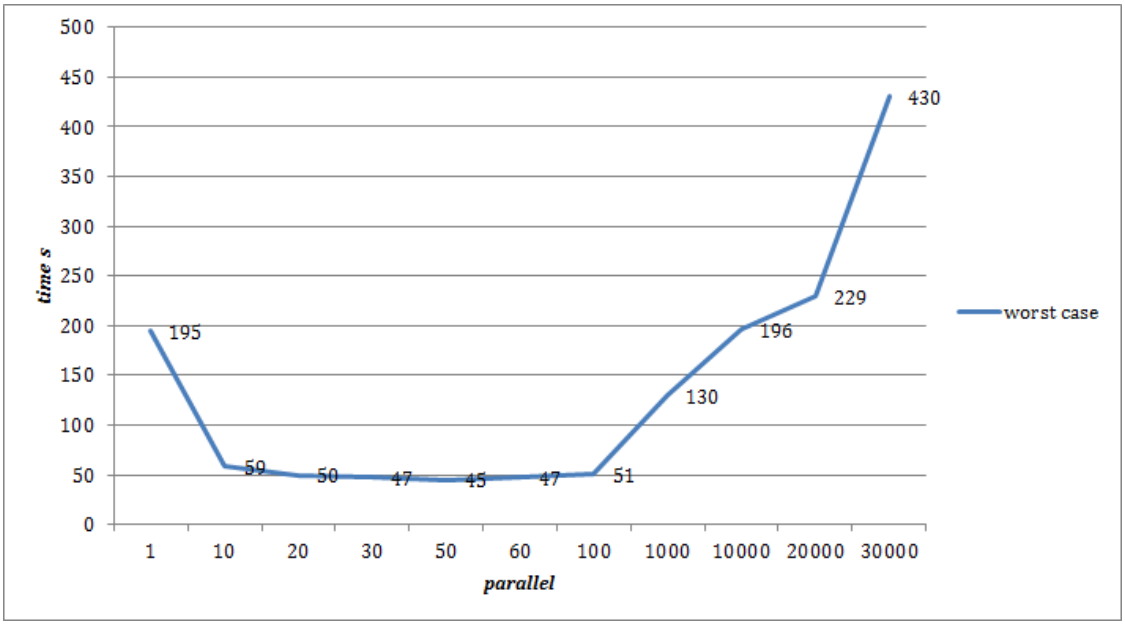


Figure 17: Temporal join with small partitions on 3 MB cache

If we only call 10 outer partitions in parallel, there is already a drastic change in the runtime behavior. Those 10 outer partitions are used in the *DIPMerge* function for each inner partition, which are 30k inner partitions. Afterwards, the next 10 outer partitions are loaded into the cache. Since these 10 outer partitions will always stay in the cache for each inner partition, there are no jumps between memory and cache which generate the main costs. And since the outer partitions are all in the cache, the jumps between partitions are very short. An inner partition is scanned 300 times having 10 outer partitions called in parallel. After passing more than about 1'000 outer partitions to be calculated in parallel, the runtime begins to grow again. How can this be explained? Even though there are less jumps overall, the execution time somehow grows. The Mac that was given to me by the *IfI* has a level-2 cache of 256 *kB* and a level-3 cache of only 3 *MB*. With more than 1'000 outer partitions, the cache becomes stuffed to a point that no outer partition will fit in anymore. This is also showed in our formula 1, where the term ($B_r - M$) does not approximate to zero anymore, because we pass that many blocks that the cache will overflow. So we lose the advantage of small jumps which we had in the cache, since we need to fetch data from the memory again. Some outer partitions are now in the cache, others are in the memory. For 30'000 outer partitions, it will take as many outer partitions as possible in the cache, scan them, and then take the next outer partitions. We have less scans of a partition, but the loading between cache and memory becomes more frequent. The jumps are now fairly big and the runtime becomes bigger and bigger. How can we prove this? We cannot be 100% certain that the cache is full. In addition, there are other processes using the cache, independent of this *DIPMerge* function. Therefore, we tried the same experiment on a different device, with more cache capacity. The MacBook from F. Cafagna has a level-three cache of 8*MB*, so there is more than twice of the memory size in the cache than on my device.

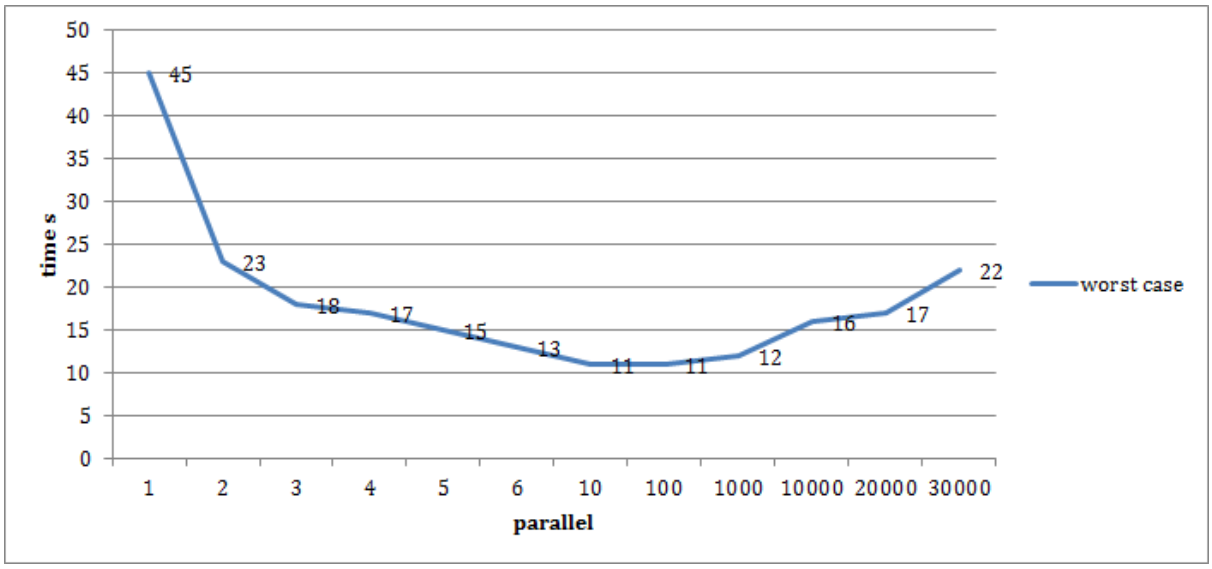


Figure 18: Temporal join with small partitions on 8 MB cache

As we can clearly see, the cache does not get overflowed like in the first example. Each inner and outer partition has to be fetched from the memory into the CPU cache. These jumps from memory into cache are the actual initiator of a big runtime, not the multiple scans of an inner partition. If we pass 10 outer partitions, the runtime is much smaller because they can be used for each inner partition. Since 10 partitions do not require much space, they will certainly fit into the cache. The runtime is faster because there are way less jumps from memory into the cache.

We showed that the parallelization speeds up the join for small partitions. With big partitions, this effect is not noticeable because the partitions will not fit entirely into the cache. Therefore, the performance difference between a parallel and a serial execution only differs in the amount of partition scans. Since the partitions are stored in an array, the tuples are very close to each other. A scan costs almost nothing, and we will not see any performance improvement here. We would only see an effect if we could reduce the scans of the outer partitions, which are our main cost. So far, we have only considered the memory implementation. For the disk implementation, this effect can be shown as well.

5.2 Performance test for temporal join on disk

In this case, the partitions are taken from the binary files. The binary files are converted into a heap data structure, where the partitions are stored in blocks. First of all, each block needs to be loaded into the memory, which is an additional step that produces costs. Therefore, we can say that a disk execution is slower than a memory execution. As already mentioned, the amount of open files is limited, depending on the operating system. Applying the same worst case scenario on about 32k partitions and calling them in parallel will not work, since my MacBook only allowed 1'000 files open. Therefore, we reduce the worst case relation to 1k tuples.

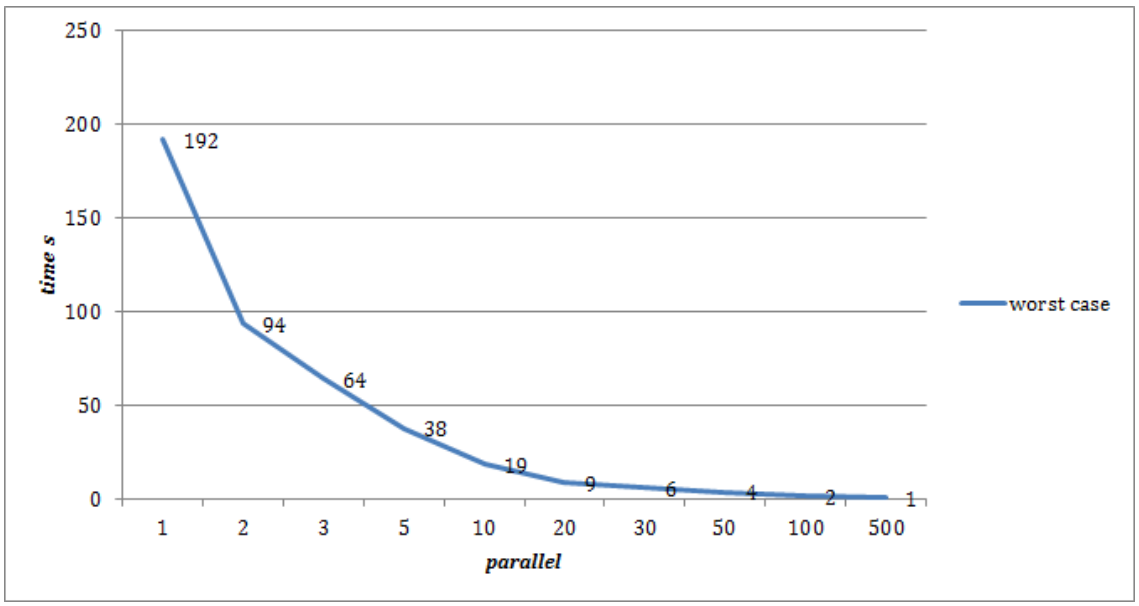


Figure 19: Temporal join with small partitions on disk

As expected, the runtime is much slower on disk than on memory. Even though we have about the same runtime for *parallel*=1 in memory and on disk, the relation for disk is about 30 times smaller. The main costs are opening a file, loading the file into the memory, and closing a file. The speeding up can be observed very well. For *parallel*=500, an inner partition has to be scanned twice, whereas an inner partition for *parallel*=1 is scanned 1'000 times. Every partition fits into the cache.

As mentioned, these additional costs are only existent for the disk implementation. The costs are reduced if we increment *parallel*. They are reduced because we close each file after a *DIPMerge* run and we have to load each partition into the memory again if not all partitions have been compared. This leads us to the assumption that the performance gain through the cache can be ignored for a moment. We want to see how the algorithm behaves if we pass big relations. Since we are able to reduce the amount of file openings and closings, we should be able to see a performance difference between *parallel*=1 and *parallel*=max.

Initially, a relation with 1 million tuples and with 10 million tuples was given to me. I wrote a small java program to get a relation with 3, 5 and 7 million tuples. It takes the input relation, copies it into a new file, and generates the same amount of tuples in addition. The start time of these tuples are calculated by adding up the start time of the last tuple from the original relation with the current data value. The end time is calculated similarly, but with the end time of the last tuple. The data value is the sum of the current position and the biggest data value. This way, we will get a relation with more tuples, but the amount of partition will always stay the same, which, in this case, is 6. I tested 4 relations with the modifications of *parallel*=1 and *parallel*=6.

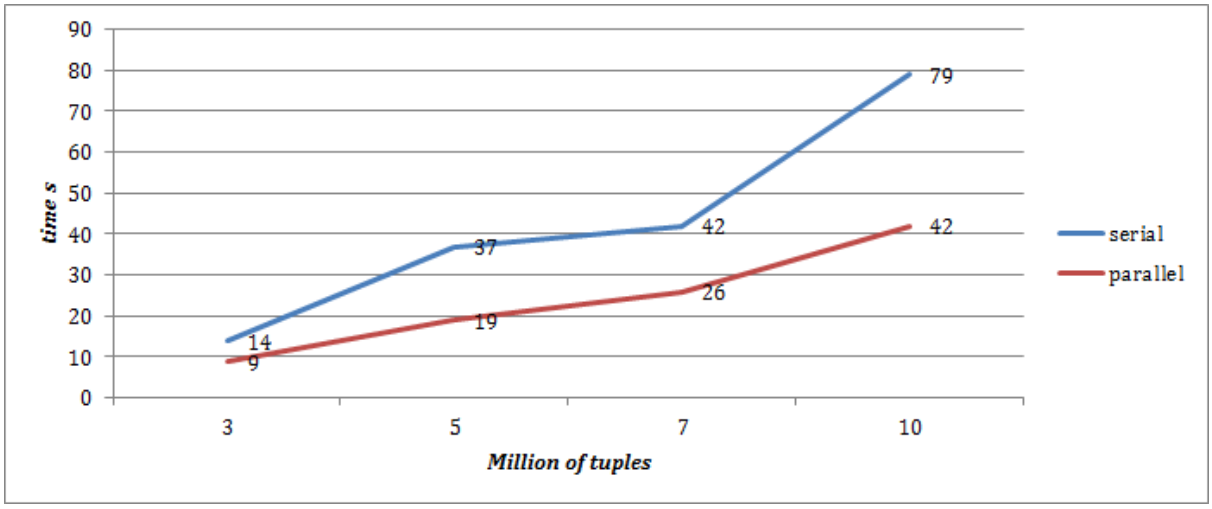


Figure 20: Temporal join with big partitions on disk

At this stage, we need to know the cost of the *DIPMerge* again. If we remember formula 1 from section 5.1, the term ($B_r - M$) approximates to zero for small partitions. But in this scenario, we have big partitions. Therefore we know, that this caching effect with big partitions is not even noticeable and we will not see any runtimes more than twice as fast as a serial execution. Overall, we have 72 scans for a serial execution and 42 scans for a parallel execution. The amount of scans goes along with the amount of file openings and closings. However, we cannot reach a result like we had with small relations. The part of the caching becomes not noticeable anymore. Since 42 scans are not half of 72 scans, the execution time cannot be more than twice as fast, showed in figure 20. Even though we do not get an improvement such as in in the previous scenario, the runtime difference is still noticeable. The graph grows in general because the relations are getting bigger. The problem that still remains is the dominant cost produced by the outer partitions. In this scenario, an outer partition is scanned 6 times. Preferably, we want each outer partition to be scanned only once. This is not possible. As we saw in section 4.1, the advancing of *r* and *s* is strictly regulated. The tuples from the outer partitions are stored in the array *r*. By Advancing *r,* we check the current tuple at position *i* and compare it to the current tuple *s*. If we wanted full parallelization, *s* would be stored in an array too. Then, our condition at line 41 would not work anymore, because it requires a tuple from a single partition. The problem is that we don't know which tuple to advance in this case. Having multiple tuples for *s* would ruin our approach, since the condition might be met for the current *r[i]* and *s[i]* partition, but not for the others, and we would miss many join matches.
The circumstance where we scan each partition only once can be achieved in the anti-join which will be shown in the following section.

5.3 Performance test for temporal anti-join on disk

What an anti-join is and how it is implemented is explained in section 3.3. The inputs for the *DIPMerge* function are the outer partitions and the full inner relation. The inner relation is scanned for each outer partition and since there is only one inner relation and not multiple inner partitions, each outer partition is scanned only once. With the parallelizing of the outer partitions, we can pass multiple outer partitions instead of only one, such as we did in the temporal join. By passing *parallel=max,* the inner relation is scanned only once. In this case, nothing is scanned multiple times and we can expect a runtime that is more than twice as fast as it was for a serial execution. To demonstrate this, I took an outer relation with 48 partitions and 800k tuples and an inner relation with 6 million tuples. Afterwards, I measured the time, as we already did before, and tested the anti-join with different configurations of *parallel.*

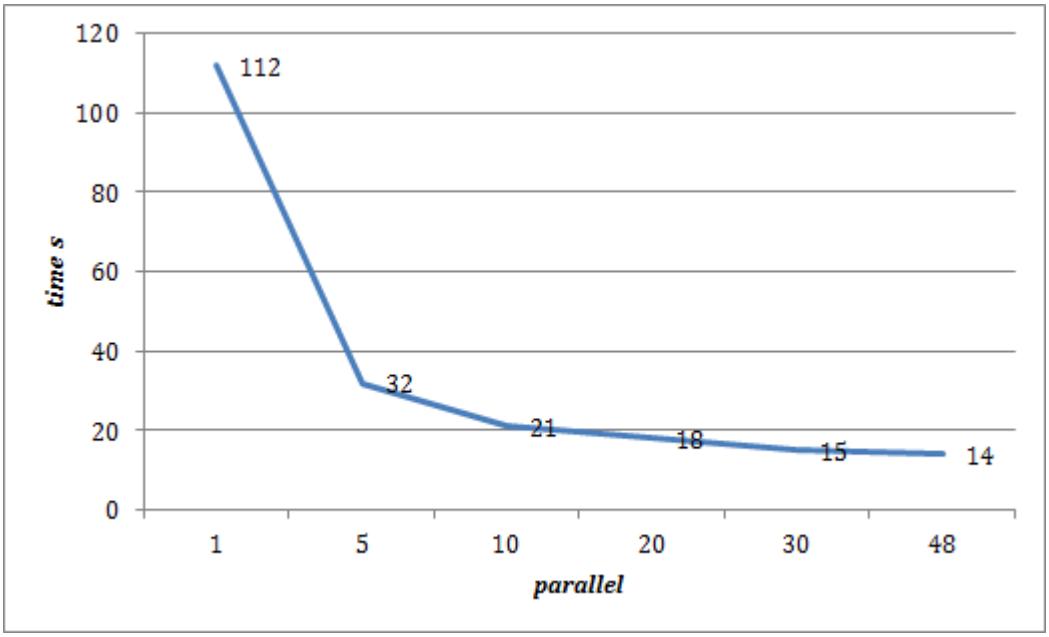


Figure 21: Temporal anti-join with big inner relation

For *parallel*=1, it took 112 seconds. The relation was scanned 48 times and each outer partition was also scanned 48 times overall. Again, the scan itself is not the expensive part. Opening the file, loading it from memory into disk and closing it produces the highest costs. Having *parallel*=48, the file of the inner relation and each outer partition are opened, loaded from disk into memory and closed only once. Therefore, we get a runtime that is about 9 times faster.

4.5 Parallelized temporal aggregation

For the aggregation, we are not able to use a parallelization. As explained in section 3.4 how the temporal aggregation works, we cannot aggregate multiple partitions simultaneously. It is specifically necessary that the first outer partition is going to be merged with the second outer partition and the result will be a partition that is going to be merged with the next outer partition and so on. Therefore, a parallel execution cannot be implemented without destroying our whole structure of the *DIPMerge* function which we want to be as consistent as possible.

5. Summary and conclusions

The task for this thesis was to implement an algorithm that takes a relation with overlapping tuples and creates several partitions in which no tuple overlaps with any tuples in the same partition. This procedure is called *Disjoint Interval Partitioning* (*DIP*). The thesis is based on a part of F. Cafagna's PhD thesis in which he theoretically proves that temporal operators perform better with disjoint interval partitions [1]. The approach of the *DIP* was taken from there. In this thesis, a relation only consists of 3 fields: a start time, an end time and a data value. The first version of the *DIP*, implemented by F. Cafagna, stored the partitions in an array of partitions which is limiting because we do not know how many partitions can arise from a relation. Therefore, the first task was to implement a *DIP* algorithm where the partitions are stored as a list. In addition, I had to implement a version for disk, where the partitions are stored on a file. The first version of the disk implementation was rather slow, because we had too many unnecessary file openings and closings. This was quickly fixed by allowing the algorithm to have up to 200 files open simultaneously.
The second task of the thesis was to implement temporal operators, which takes the partitions from the *DIP* as an input. A temporal join, anti-join and aggregation was implemented for a memory and a disk version. But what is the advantage of having multiple partitions in which no tuple overlaps with another instead of having the raw relation as an input for the temporal operators? Due to the *DIP*, we are able to avoid the so called backtracking that we had to do for a relation with overlapping tuples. We call it backtracking because for a current tuple *r* (from the outer relation) and *s* (from the inner relation), we have to consider all previous *s* tuples which resulted in a productive join match. With the *DIP*, we got rid of this procedure. To answer the question, through the *DIP*, we will have many less unproductive join matches as we will have them with raw relations.
I started my thesis on January 14, 2016 and implemented all requirements from the task sheet that have been declared by F. Cafagna after about three and a half months. At this point, F. Cafagna came up with the idea of optimizing the temporal operator functions. We both agreed that this would be an interesting addition to my thesis. The idea was to parallelize the outer partitions, which reduces the amount of multiple scans of the same partition. The improvement was strongly noticeable. For a join with small relations, we got runtimes up to 4 times faster than the original implementation. For disk, we showed that the parallelism can be applied on the anti-join, with runtimes up to 8 times difference to the original implementation. The experiments and our formula 1 also proved that we cannot get a runtime more than twice as fast for a join with big relations. The reason for that is that we never scan half of the amount of partitions with full parallelization. Overall, we got about ½ less scans with the parallelism (for temporal joins). The dominant costs are the outer partitions, which are still scanned multiple times, and unfortunately, this circumstance cannot be avoided. For small relation, the parallelized partitions stay in the cache, which speeds up the runtime up to four times.
Even though the optimization was the most challenging part, in my opinion, considering the results that we achieved, it was absolutely worth it.
Future work might address the implementation of the aggregation projections. In addition, the comparison to other approaches such as *OIP* (Overlap Interval Partitioning [2]), the *Temporal Alignment* [3] and the *TimeLine Index* [4] can be considered.

Bibliography

[1] Francesco Cafagna, Michael H. Böhlen, Annelies Bracher. Disjoint Interval Partitioning. University of Zurich, 2015.

[2] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014,* pages 1459–1470, 2014.

[3] A. Dignös, M. H. Böhlen, and J. Gamper. Temporal alignment. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012,* pages 433-444, 2012.

[4] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data,* SIGMOD '13, pages 1173-1184, New York, NY, USA, 2013. ACM.

Installation Guidelines

On the provided CD, there is a folder called *DIP*. Copy this folder to your desktop, open the command line and navigate to this folder. For p, you can choose the amount of outer partitions called in parallel. The text files “tfdb100k.txt” are the input relations. There are many others of them in the *DIP* folder if you want to choose any others. In this case, unzip them first.

For a temporal join on disk, use the command:
./main -o 2 --of ../dip/tfdb100k.txt --if ../dip/tfdb100k.txt -a DipDisk --on-disk --Join --p 6

For a temporal anti-join on disk, use the command:
./main -o 2 --of ../dip/tfdb100k.txt --if ../dip/tfdb100k.txt -a DipDisk --on-disk --AntiJoin --p 6

For a temporal aggregation on disk, use the command:
./main -o 2 --of ../dip/tfdb100k.txt --if ../dip/tfdb100k.txt -a DipDisk --on-disk --Aggregation

For a temporal join on memory, use the command:
./main -o 2 --of ../dip/tfdb100k.txt --if ../dip/tfdb100k.txt -a DipMem --on-array --Join --p 6

For a temporal anti-join on memory, use the command:
./main -o 2 --of ../dip/tfdb100k.txt --if ../dip/tfdb100k.txt -a DipMem --on-array --AntiJoin --p 6

For a temporal aggregation on memory, use the command:
./main -o 2 --of ../dip/tfdb100k.txt --if ../dip/tfdb100k.txt -a DipMem --Aggregation

Content of the CD

- Abstract file on English
- Abstract file on German
- Report.pdf contains the thesis
- DIP.zip contains the source code