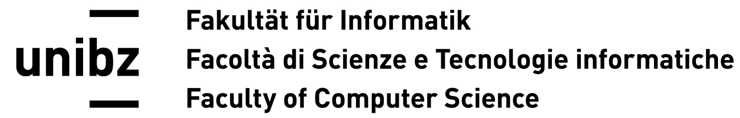


Optimizing the Computation of Parsimonious Temporal Aggregation Queries

Giovanni Mahlknecht

Supervisor Prof. Johann Gamper

Abstract

In database applications time has become an aspect of growing importance. As consequence the amount of data to store increases exponentially. To facilitate evaluation and interpretation of information temporal aggregation is used. Two important aggregation types are Instant Temporal Aggregation (ITA) and Span Temporal Aggregation (STA). Parsimonious Temporal Aggregation (PTA) addresses the weakness of both combining the best properties. The drawback of PTAc is the computational complexity of $\mathcal{O}(n^2c)$ and the space complexity $\mathcal{O}(n^2)$.

In this work two optimizations to the existing PTAc algorithm are introduced. The first one consists in reducing the search space of the dynamic programming scheme adopted by PTAc. This leads to an avoidance of computations reducing the overall complexity of the algorithm. The second optimization addresses the space complexity problem. Through the introduction of a new data structure Split Point Graph memory consumption can be noticeably reduced. Empirical evaluation shows the effectiveness and the limits of the optimizations.

Acknowledgements

I am heartily thankful to my supervisor, Prof. Johann Gamper, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. Without his valuable suggestions and corrections this thesis would not have been possible.

I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Contents

1 Introduction . . . 6
2 Background . . . 7
2.1 Running Example . . . 7
2.2 Overview of Temporal Aggregation . . . 7
2.3 Parsimonious Temporal Aggregation . . . 9
2.3.1 Adjacent Tuples . . . 9
2.3.2 Merging Adjacent Tuples . . . 10
2.3.3 Reduction of the Input Relation . . . 10
2.3.4 Error Function and Efficient Computation . . . 11
2.3.5 Concept of Split Point . . . 12
2.3.6 Error Matrix **E** . . . 13
2.3.7 Split Point Matrix **J** . . . 15
2.3.8 Matrices **E** and **J** of Example . . . 16
3 Optimization of the Dynamic Programming Scheme of PTA . . . 17
3.1 Existing Optimizations . . . 17
3.2 Diagonal Pruning . . . 18
3.2.1 Improvements . . . 19
3.2.2 Space Complexity . . . 20
3.3 Pruning in Last Row . . . 21
4 Split Point Graph as Alternative to Split Point Matrix . . . 22
4.1 Split Path . . . 22
4.2 Graph Representation of the Matrix **J** . . . 23
4.3 Graph Creation and Insertion of new Entries . . . 24
4.4 Search of Destination Nodes . . . 25
4.5 Optimization of SPG . . . 27
4.5.1 Orphaned Elements in Split Point Matrix **J** . . . 27
4.5.2 Elimination of Superfluous Nodes: Path Pruning 28
4.5.3 Path Pruning Algorithm . . . 30
4.6 Complexity Analysis . . . 31
4.6.1 Graph Size . . . 31
4.6.2 Node Pruning Efficiency . . . 31
4.6.3 Memory Requirements . . . 34
5 Algorithms . . . 34
5.1 Data Structure for Split Point Graph . . . 36

5.2 Optimization of Node Insertion 37
6 Implementation 39
7 Experimental Evaluation 42
7.1 Setup and Data 42
7.2 Runtime of Diagonal Pruning Approach 43
7.3 Runtime Split Point Graph Implementation 43
7.4 Memory Requirements 44
7.5 Scalability to Large Datasets 47
8 Conclusion 48

List of Figures

1 Example *patients* and different temporal aggregations 8
2 Splitpoint j=7 12
3 Splitpoints for example 8 13
4 Splitpoints for running example, reduction to $c = 5$ 23
5 Split point graph for running example 24
6 Node insertion in Split Point Graph (example 23) 25
7 Evolution of Split Point Graph of the running example 26
8 Search space when inserting a node $N_{3,5}$ with destination node 3 26
9 Split point graph for running example after computation step k=2 29
10 Split point graph for running example after computation step k=3 29
11 Split point graph for running example after computation step k=4 29
12 Split point graph for running example after computation step k=5 30
13 Split point graph for running example after computation step k=5, only node 9 is computed 30
14 Size of split point graph dataset climacubes, n=870, c=700 for each k-step 32
15 Theoretical Model and real measures of Split Point Graph size . 32
16 Maximum size of Split Point Graph, dataset SYNTH, n=1000 . . 33
17 Memory usage theoretical model, n=1000 36
18 Split Point Graph implementation 37
19 Split Point Graph implementation with auxiliary arrays 40
20 Runtime in function of reduction size c, algorithms PTAc and MP 43
21 Runtime in function of size of the argument relation n, algorithms PTAc and GP 44
22 Runtime in function of reduction size c, algorithms GP and MP . 45
23 Memory consumption in function of argument relation size n, algorithms PTAc and GP 46
24 Memory ratio (GP / PTAc) in function of reduction size c, ITA size n=5000 46
25 Runtime and memory requirements in function of argument relation size n for large scale dataset SYNTH 46

List of Tables

1 Initialization of the **E** matrix 14
2 **E** matrix after computing the first row 14
3 Reutilization of cells in computation of the **E** matrix 15
4 Error matrix **E** of the running example 16
5 Split point matrix **J** of the running example 16
6 **E** matrix diagonal pruning 19
7 Eliminate-able cells with diagonal and last row pruning 20
8 Not reachable split points of the example relation 27
9 Split Point Graph maximum size, dataset SYNTH, n=1000 . . . 33
10 Datasets used for experiments 42
11 Algorithm configuration used for experimental evaluation 43
12 Average reduction of the runtimes GP vs PTAc 44
13 Average reduction of the memory usage for algorithms GP and PTAc 45

1 Introduction

In most database applications time has become an aspect of growing importance. Example are record-keeping applications, financial applications and measurement of phenomena. Nowadays commercial database management systems have only limited support for temporal data. A non temporal database captures only an instantaneous snapshot of the represented mini-world. If such a database is used, management of the temporal aspect has to be modeled at application level and is not really supported by the Database Management System itself.

In the past three decades conceptual models have been studied [BBJ98, JSS94] and different concrete temporal data models have been developed. However, there are advantages and disadvantages in each model and no consensus in the adoption of a concrete model has yet been reached.

Temporal datasets may grow very fast since one single fact is represented by multiple tuples in each relation. Each tuple is associated to a single time point or time interval, depending on whether the point based time model or interval based time model is used.

To summarize the huge amount of data, temporal aggregation can be applied. Non temporal aggregation transforms an argument relation into a summary result relation. The aggregation process is a two step task. The first step is the partitioning of the relation in different groups based on the value of the grouping attributes. The second task is the application of the aggregate functions to these groups. Temporal aggregation, substantially more complex, involves the time dimension as additional grouping attribute. The timeline is partitioned and the argument relation is grouped over these partitions. The two main types of temporal aggregation that have been studied in the past are Instant Temporal Aggregation (ITA) and Span Temporal Aggregation (STA). Parsimonious Temporal Aggregation (PTA) addresses the weakness of ITA and STA combining the best properties of both. According to Gordevičius et al. [GGB12] the drawback of PTA is the computational complexity of $\mathcal{O}(n^2c)$. The space complexity or PTA is $\mathcal{O}(n^2)$.

This work aims to reduce both the runtime and the memory consumption of the PTAc algorithm. On basis of experimental evaluation is shown that both can be reduced substantially. In more detail, the contribution is the following:

Pruning reduction of the runtimes by avoidance of superfluous computations in the dynamic programming scheme adopted by PTAc.

Split Point Graph substitution of a matrix used in the PTAc computation by a graph structure to achieve a reduction of the used memory.

Experimental evaluation on different dataset showing the effectiveness and limits of the proposed optimizations.

In section 2 different forms of temporal aggregation are compared and the Parsimonious Temporal Aggregation operator is explained. In section 3 two optimizations of the PTAc algorithm are proposed. Both approaches are based on the avoidance of unnecessary computations. The second problem, the consumption of memory space, is addressed in section 4. The following two sections are focussing on the algorithms and their implementation. In section 7 some experiments on different datasets are described.

2 Background

2.1 Running Example

As example data for the explanations a constructed dataset containing information about patients in a hospital is used. For each patient the name (pat), the department where he/she was located (dep), the therapy type (therapy) which the patient receives, the daily cost of the therapy and the time period of the hospitalization (T) is given. The single records and a graphical representation emphasizing the temporal distribution of the records and the therapy type is shown in figure 1a.

2.2 Overview of Temporal Aggregation

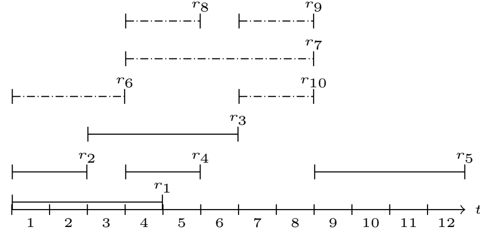
Span Temporal Aggregation and Instant temporal Aggregation differ in how the timeline is partitioned.

STA [BGJ06] partitions on a defined time period, such as week or month. For each of these intervals a result tuple is produced by aggregating all tuples in the input relation that are overlapping the interval. Thus the partitions are not dependent on the data distribution. An example query which can be answered with a STA result set is *For each half-year period and department, what is the number of patients?*.

ITA [KS95, BGJ06] instead identifies time partitions over which the aggregate values are remaining constant. These intervals are called constant intervals. As explained in [BGJ06] because of overlapping tuples in the input relation the output relation can be twice as large as the input relation which contradicts the idea of aggregation. An example query which can be answered with an ITA result set is *For each day and department, what is the number of patients?*. This work does not address the computation of the intermediary ITA relation as multiple algorithms for their computation have already been studied [KS95, MLI03, YW03].

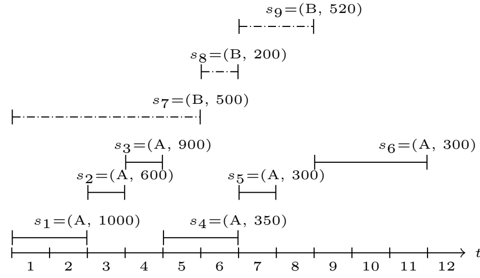
The advantage of ITA over STA is that the grouping interval length is chosen on the basis of the available data, hence it considers the distribution of the tuples

	pat	dep	ther	cost	T
r_1	Bob	Ortho1	A	600	[1,4]
r_2	Mary	Ortho1	A	400	[1,2]
r_3	Mart	Ortho2	A	300	[4,7]
r_4	Joe	Ortho2	A	50	[5,6]
r_5	Max	Ortho1	A	300	[9,12]
r_6	John	Ortho2	B	500	[1,3]
r_7	James	Ortho1	B	200	[4,8]
r_8	Luis	Ortho2	B	300	[4,5]
r_9	Mel	Ortho1	B	20	[7,8]
r_{10}	Luisa	Ortho1	B	300	[7,8]



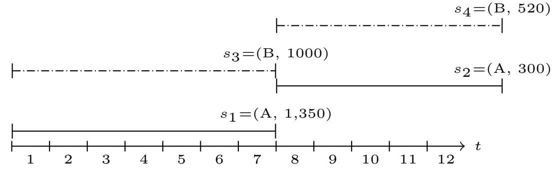
(a) Relation patients

	ther	val	ts	te
s_1	A	1000	1	2
s_2	A	600	3	3
s_3	A	900	4	4
s_4	A	350	5	6
s_5	A	300	7	7
s_6	A	300	9	12
s_7	B	500	1	5
s_8	B	200	6	6
s_9	B	520	7	8



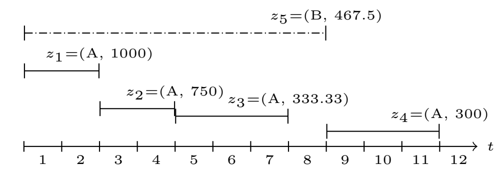
(b) ITA

	ther	val	ts	te
s_1	A	1350	1	7
s_2	A	300	8	14
s_3	B	1000	1	7
s_4	B	520	8	14



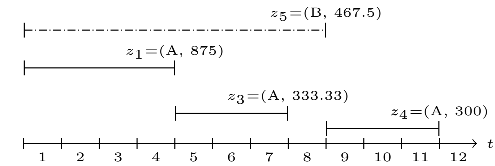
(c) STA, aggregation period *week*

	ther	val	ts	te
z_1	A	1000	1	2
z_2	A	750	3	4
z_3	A	333.3	5	7
z_4	A	300	9	11
z_5	B	467.5	1	8



(d) PTA, size 5

	ther	val	ts	te
z_1	A	875	1	4
z_2	A	333.3	5	7
z_3	A	300	9	11
z_4	B	467.5	1	8



(e) PTA, size = 4

Figure 1: Example *patients* and different temporal aggregations

over the timeline. The disadvantage is the fact that in most cases the input relation increases in size.

Generating the ITA aggregation of the example relation *patients* with grouping attributes *therapy*, aggregation function *sum* over the attribute type *cost* leads to the data reported in figure 1b. As expected the ITA result is about the same size (9) as the argument relation (10). In figure 1c the STA result set is shown. It is evident that the result s_4 with value (B,520) and a duration from day 7 to day 14 results from three tuples (r_7, r_9, r_{10}). Each one of these overlaps for only one day with the aggregation period but it has a rather high contribution to the result.

PTA addresses the weakness of ITA and STA combining the best properties of both. The construction of a PTA result set out of an input relation $\mathbf{r}$ includes two steps. First the construction of an intermediary ITA result and in a second step the approximation of this relation until a specified size or error is reached. The approximation minimizes the introduced error. Therefore in a PTA result the time partitions are depending on the data and the size of the resulting relation can be chosen by the user. In figure 1d and 1e the aggregations using the PTA operator to size $c = 5$ and $c = 4$ are shown.

2.3 Parsimonious Temporal Aggregation

In this section some base concepts of PTA are briefly described.

2.3.1 Adjacent Tuples

Definition 1. Adjacent tuples: given a sequential relation $\mathbf{s}$ with schema $S = (A_1, \ldots, A_k, B_1, \ldots B_p, T)$. Two tuples s_i and s_j are adjacent if the following two conditions hold

$$s_i \prec s_j \iff \begin{cases} s_i.\mathbf{A} = s_j.\mathbf{A} \\ s_i.t_e = s_j.t_b - 1 \end{cases}$$

The attributes $\mathbf{A} = \{A_1, \ldots A_k\}$ are grouping attributes; $B_1, \ldots, B_p$ are the aggregation attributes, i.e. the attributes over which with an aggregation function is applied to coalesce values. The first condition ensures, that the tuples have the same values in the grouping attributes. The second condition ensures that the tuples are not separated by a temporal gap. T_b is the begin of the duration whereas T_e is the end of the duration.

Example 1. In the example the relation tuple s_5 is not adjacent to tuple s_6 ($s_5 \not\prec s_6$) because they are separated by a temporal gap since s_5 ends at time instant 7 while s_6 starts at time instant 9. Also tuples $s_6 \not\prec s_7$ since the values

in the grouping attribute *therapy* differs. For all other tuples the adjacency property holds: $s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5$ and $s_7 \prec s_8 \prec s_9$.

2.3.2 Merging Adjacent Tuples

PTA reduces the size of the ITA input relation by merging adjacent tuples. The merging function is a weighted mean over the aggregation attributes. The weights are the length of the timestamps of the two tuples.

Definition 2. The merge of two adjacent tuples $s_i \prec s_j$ is defined as

$$s_i \oplus s_j = (s_i.\mathbf{A}, v_1, \ldots v_p, [s_i.T_b, s_j.T_e])$$

where $v_d = \frac{|s_i.T|s_i.B_d + |s_j.T|s_j.B_d}{|s_i.T| + |s_j.T|}$

Since the precondition for merging tuples is the adjacency property, the grouping attributes $s_i.\mathbf{A}$ and $s_j.\mathbf{A}$ are value equal. There is no temporal gap between s_i and s_j as well. The duration of the resulting tuple is the combination of the duration of s_i and s_j, equals to $[s_i.T_b, s_j.T_e]$.

Example 2. Merging two tuples of the example relation $s_1 = (A, 1000, [1, 2])$ and $s_2 = (A, 600, [3, 3])$. The resulting tuple $s_z = s_1 \oplus s_2 = (A, 866.67, [1, 3])$. The merge value is computed as $(2 * 1000 + 1 * 600)/(2 + 1) = 866.67$.

2.3.3 Reduction of the Input Relation

To reduce the ITA result set **s** to a specified size adjacent tuples are merged recursively until a specified size is reached.

Definition 3. c is the size of the result of the reduction. The minimal size of the reduction, c_{min} is bound by the number of non adjacent tuples .

$$c_{min} = |\{(s_i, s_{i+1}) | s_i, s_{i+1} \in \mathbf{s} \wedge s_i \not\prec s_{i+1}\}| + 1$$

Example 3. The relation *patients* has minimum size of 3 due to a temporal gab between tuples s_5 and s_6 and a grouping attribute gap between tuples s_6 and s_7.

In the reduction process adjacent tuples are substituted by the merged tuple until the relation reaches size c. Each merge step introduces an error.

PTA defines a non deterministic reduction function ρ which is defined as follows.

Definition 4. The non deterministic reduction function ρ is defined as

$$\rho(\mathbf{s}, c) = \begin{cases} \mathbf{s} & |\mathbf{s}| \leq c \\ \rho(\mathbf{s}\{s_i, s_j\} \cup \{s_i \oplus s_j\}, c) & |\mathbf{s}| > c. \end{cases}$$

The reduction function does not state which tuples are selected for merging.

Example 4. Relation *patients* can be reduced to size $c = 4$. A possible reduction is the relation $\mathbf{z} = \{z_1 = s_1 \oplus s_2, z_2 = s_3 \oplus s_4, z_3 = s_5, z_4 = s_6 \oplus s_7 \oplus s_8 \oplus s_9\}$. However different selection of tuples to merge would give a different result set.

2.3.4 Error Function and Efficient Computation

During the reduction process pairs of adjacent tuples have to be selected for merging. Each one induces an error in respect to the original ITA input set. The selection of tuples is based on the result of a grading function.

Example 5. Assume that the better solution among two possible merges, $s_1 \oplus s_2$ and $s_4 \oplus s_5$, have to be chosen. Tuple s_1 has a value of 1,000 with duration 2; tuple s_2 a value of 600 with duration 1. The resulting merged tuple $z_1 = s_1 \oplus s_2 = (A, 866.67, [1, 3])$ while for merging tuples $z_2 = s_4 \oplus s_5 = (A, 333.33, [5, 7])$. It is intuitively evident, that the merge of s_1 and s_2 induces a higher error than the merge of s_4 and s_5. The aggregation values in the latter pair are closer than in the first, hence the approximation is better.

PTA uses as grading function for the merging selection as error measure the squared sum error between the ITA input relation and the reduction.

$$SSE(\mathbf{s}, \mathbf{z}) = \sum_{z \in \mathbf{z}} \sum_{s \in \mathbf{s}_z} \sum_{d=1}^{p} w_d^2 |s.T| (s.B_d - z.B_d)^2$$

$\mathbf{z}$ is a reduction of the ITA input set $\mathbf{s}$, for every $z \in \mathbf{z}$ let $\mathbf{s}_z$ be the set of ITA tuples that are merged into z. The term w_d is a weighting factor for each aggregation attribute to leverage the impact of the different aggregation attributes.

Example 6. Considering the reductions of example 5. Assume the reduction $\mathbf{z}$ is gained by merging tuples s_1 and s_2. The weight factor w for attribute *cost* is set to 1.

The resulting relation is therefore formed by the tuples $z_1 = s_1 \oplus s_2 \oplus$, $z_2 = s_3$, $z_3 = s_4$, $z_4 = s_5$, $z_5 = s_6$, $z_6 = s_7$, $z_7 = s_8$.

$$\mathbf{z} = \{z_1, \ldots, z_7\} = \{s_1 \oplus s_2, s_3, s_4, \ldots, s_8\}$$

The set $\mathbf{s}_{z_1}$ are the tuples in the ITA relation merged into z_1

$$\mathbf{s}_{z_1} = \{s_1, s_2\}$$

Therefore the error $SSE(\mathbf{s}, \mathbf{z}) = 2(1000-866.67)^2+1(600-866.67)^{=}106666.67$.

It is not necessary to compute the errors for the tuples $z_2, \dots, z_7$ since for each of these tuples the error contribute is 0, because they coincide with the original tuples.

For the second reduction where tuples s_4 and s_5 are merged, the error is $2(350-333.33)^2 + 1(300-333.33)^2 = 1,666.67$, hence the merge of tuples s_4 and s_5 induces a lower error than the merge of tuples s_1 and s_2 and therefore it is the better selection.

For efficient error computation PTA uses a technique introduced by Jagadish et al. [JKM+98] in order to compute the error in constant time.

2.3.5 Concept of Split Point

The reduction n tuples to size c can be divided into two steps: first find a certain point j, and merge all following tuples into 1, $\rho(\{s_{j+1}, \dots, s_n\}, 1)$, second reduce the remaining tuples (the tuples $\{s_1, \dots, s_j\}$) to size $c-1$: $\rho(\{s_1, \dots, s_j\}, c-1)$.

Definition 5. Let be a sequential ITA set $\mathbf{s} = \{s_1, ..., s_n\}$, $\mathbf{s}_j = \{s_1, ..., s_j\}$, $\mathbf{s} \setminus \mathbf{s_j} = \{s_{j+1}, ..., s_n\}$, j is called split point.

Example 7. The split point $j = 7$ splits the sequential ITA relation into two parts, $\mathbf{s}_7 = \{s_1, \dots s_7\}$ and $\mathbf{s} \setminus \mathbf{s}_7 = \{s_8, s_9\}$. Each set is then reduced in a separate step.

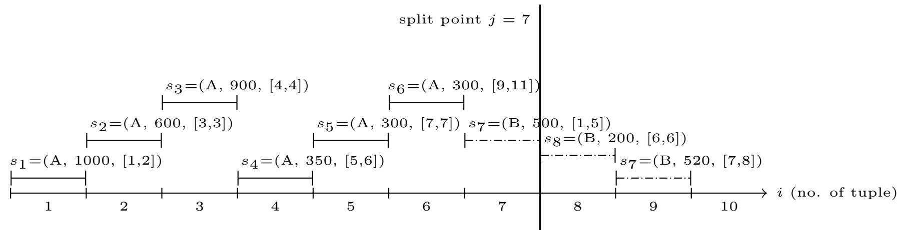


Figure 2: Splitpoint j=7

Function ρ can then be rewritten as

$$\rho(\mathbf{s}, c) = \begin{cases} \rho(\mathbf{s_j}, c-1) \cup \rho(\mathbf{s} \setminus \mathbf{s_j}, 1) & c < |s| \\ \mathbf{s} & \text{if } c = |s| \end{cases}$$

For an optimal reduction, the sum of errors introduced on both sides of j must be minimized at each step.

Example 8. There are 7 possible split points to reduce the example relation to size $c = 3$: 2, 3, 4, 5, 6, 7, 8. However only three splitpoints (6, 7 and 8) are useable because for all split points lower than 6 a merge over non adjacent tuples is necessary. For split point 5 the tuples $\{s_6, \ldots, s_9\}$ have to be merged to size 1 which is not possible because $s_6 \not\prec s_7$.

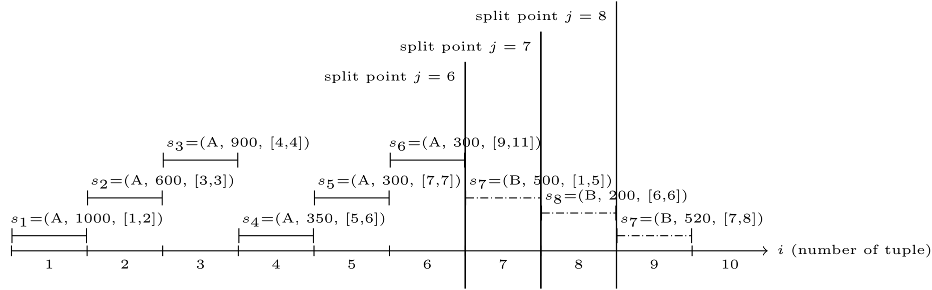


Figure 3: Splitpoints for example 8

2.3.6 Error Matrix E

PTA uses a dynamic programming technique that constructs an error matrix $\mathbf{E}$ and a split point matrix $\mathbf{J}$. Both matrices are of size $c \times n$. In the error matrix each element $\mathbf{E}_{k,i}$ is the smallest error in reducing i tuples to size k.

The matrix is filled incrementally row-wise and each row column-wise, in each row the values computed in the previous row are used.

$$\mathbf{E}_{k,i} = \begin{cases} \min\limits_{k-1 \leq j < i} \{\mathbf{E}_{k-1,j} + SSE(\mathbf{s}_i \setminus \mathbf{s}_j, \rho(\mathbf{s}_i \setminus \mathbf{s}_j, 1))\} & k > 1 \\ SSE(\mathbf{s}_i, \rho(\mathbf{s}_i, 1)) & k = 1 \wedge (s_1 \prec \ldots \prec s_i) \\ \infty & k = 1 \wedge \neg(s_1 \prec \ldots \prec s_i) \end{cases}$$

$\mathbf{E}$, shown in table 1, is initialized to values 0 if $i = k$ (no reduction has to be made at all); infinity if the error has to be computed and is not initialized if the reduction is not possible ($k > i$), that is if the reduction is greater than the input set.

Table 1: Initialization of the **E** matrix

	$i=1$	2	3	4	5	6	7	8	9
$k=1$	0	∞	∞	∞	∞	∞	∞	∞	∞
$k=2$	–	0	∞	∞	∞	∞	∞	∞	∞
$k=3$	–	–	0	∞	∞	∞	∞	∞	∞
$k=4$	–	–	–	0	∞	∞	∞	∞	∞
$k=5$	–	–	–	–	0	∞	∞	∞	∞

For the first row the reduction of $\mathbf{s}_i$ to 1 tuple is computed by evaluating $SSE(\mathbf{s}_i, \rho(\mathbf{s}_i, 1))$. This can be done until reaching the first non adjacent tuple pair.

Example 9. For the example relation in the cell $\mathbf{E}_{1,2}$ the error in reducing the first two tuples to size 1 is found, i.e. $SSE(\{s_1, s_2\}, \{s_1 \oplus s_2\}) = 2(1000 - 866.67)^2 + 1(600 - 866.67)^2 = 106,666.67$. The value of cell $\mathbf{E}_{1,3}$ is set to $SSE(\{s_1, s_2, s_3\}, \{s_1 \oplus s_2 \oplus s_3\}) = 2(1000-875)^2 + 1(600-875)^2 + 1(900-875)^2 = 107,451$. The value of $s_1 \oplus s_2 \oplus s_3 = (A, 875, [1:4])$.

The values for $\mathbf{E}_{1,6}, \ldots, \mathbf{E}_{1,9} = \infty$ because the first gap lies between tuples s_5 and s_6.

The results for $k=1$ are shown in table 2.

Table 2: **E** matrix after computing the first row

	$i=1$	2	3	4	5	6	7	8	9
$k=1$	0	106667	107500	475000	612143	∞	∞	∞	∞
$k=2$	–	0	∞	∞	∞	∞	∞	∞	∞
$k=3$	–	–	0	∞	∞	∞	∞	∞	∞
$k=4$	–	–	–	0	∞	∞	∞	∞	∞
$k=5$	–	–	–	–	0	∞	∞	∞	∞

The **E** matrix is filled row-wise and the results of the previous row are reutilized in the computation of each row. For example in the second row the elements of row 1 are used to compute the minimal error. The value of variable j for which the minimum error is found is the split point for reducing the partial relation.

Example 10. Which is the error in reducing the first 3 tuples to size 2, therefore the value for element $\mathbf{E}_{2,3}$? There are two possibilities: a) merge tuples s_1 and s_2, leave tuple s_3 and b) leave tuple s_1 and merge tuples $s_2 \oplus s_3$. In the first case the error can be computed from $\mathbf{E}_{1,2} + SSE(\{s_3\}, \{s_3\})$. In the second

case the error is $\mathbf{E}_{1,1} + SSE(\{s_2, s_3\}, \{s_2 \oplus s_3\})$. The least of these errors is memorized in the error matrix $\mathbf{E}$. For the merge possibility a) the split point value is 2, for possibility b split point has value 1.

As shown in table 3 the values from $\mathbf{E}_{1,1}$ to $\mathbf{E}_{1,2}$ are used to compute $\mathbf{E}_{2,3}$. The values from $\mathbf{E}_{3,3}$ to $\mathbf{E}_{3,7}$ are used to compute $\mathbf{E}_{4,8}$.

Table 3: Reutilization of cells in computation of the $\mathbf{E}$ matrix

	$i = 1$	2	3	4	5	6	7	8	9
$k = 1$	0	106667	107500	475000	612143	∞	∞	∞	∞
$k = 2$	–	0	45000	107500	109167	612143	∞	∞	∞
$k = 3$	–	–	0	45000	46667	109167	612143	687143	694493
$k = 4$	–	–	–	0	1667	46667	109167	184167	191517
$k = 5$	–	–	–	–	0	1667	46667	109167	129017

More in general the error in reducing the first i tuples to size k, i.e. the value in $\mathbf{E}_{k,i}$ is given by finding the minimum of $\mathbf{E}_{k-1,j} + SSE(\{s_{j+1}, \ldots, s_i\}, \{s_{j+1} \oplus \ldots \oplus s_i\})$ for j ranging from $k-1$ to i.

2.3.7 Split Point Matrix J

It is not sufficient to keep only the information about the least error to reconstruct the output relation. PTA uses a second matrix, the Split Point matrix $\mathbf{J}$. Each element of $\mathbf{J}_{k,i}$ is the first split point to use for the reduction of the first i tuples to size k. The split point values are computed during the computation of the error matrix and are the value of the variable j for which the error is minimal.

Example 11. In example 10 the value of $\mathbf{E}_{2,3}$ has been computed. The two possibilities to reduce the first three tuples to size 2 were: a) split point $j = 2$: tuples s_1 and s_2 are merged, s_3 is left separated b) split point $j = 1$: tuples s_2 and s_3 are merged, tuple s_1 is left separated. Solution b) has a lower error, therefore $\mathbf{J}_{2,3} = 1$. Computing the value for $\mathbf{E}_{3,5}$ the optimal split point is 3. Therefore tuples s_4 and s_5 are merged, the information about further processing of the remaining 3 tuples is retrieved from row 2 of the matrices, thus, the next split point is found at $\mathbf{J}_{2,3}$

When reducing i tuples to size k, at most k split points have to be kept. In the $\mathbf{J}$ matrix all possible ways to reduce i tuples to size k are kept (i ranges from 1 to n, k from 1 to c). This is necessary because only in the last computation step when computing $\mathbf{E}_{c,n}$ and $\mathbf{J}_{c,n}$ the starting point of the optimal reduction is decided, hence only one way for the reduction is decided.

Example 12. The result of the computation of the matrices for $c = 5$ of the running example is the split point matrix $\mathbf{J}$ shown in table 5. The first split point is indicated at $\mathbf{J}_{5,9}$ which is 6, therefore all tuples after tuple number 6, $s_7, \ldots, s_9$ are merged to size 1 and the remaining tuples $s_1, \ldots s_6$ are reduced to size $c-1 = 4$. The necessary information how to reduce the first 6 tuples to size 4 can be found in cell $\mathbf{J}_{4,6}$ with value 5. This means that the tuple s_6 remains and the tuples $s_1, \ldots, s_5$ are reduced to size 3. Following all entries until split point 0 is reached leads to the list $[6, 5, 3, 1, 0]$. Under the assumption that $\mathbf{J}_{5,9}$ has value 5 the list of split points is different: $[5, 3, 2, 1, 0]$ which gives a different reduction.

2.3.8 Matrices E and J of Example

Applying the PTA-Algorithm to the ITA result set of the example data (figure 1b) with parameter $c = 5$ the resulting error matrix $\mathbf{E}$ is shown in table 4 and the split-point-matrix $\mathbf{J}$ in table 5.

Table 4: Error matrix $\mathbf{E}$ of the running example

	$i = 1$	2	3	4	5	6	7	8	9
$k = 1$	0	106667	107500	475000	612143	∞	∞	∞	∞
$k = 2$	–	0	45000	107500	109167	612143	∞	∞	∞
$k = 3$	–	–	0	45000	46667	109167	612143	687143	694493
$k = 4$	–	–	–	0	1667	46667	109167	184167	191517
$k = 5$	–	–	–	–	0	1667	46667	109167	129017

Table 5: Split point matrix $\mathbf{J}$ of the running example

	$i = 1$	2	3	4	5	6	7	8	9
$k = 1$	$\boxed{0}$	0	0	0	0	0	0	0	0
$k = 2$	0	1	$\boxed{1}$	3	3	5	0	0	0
$k = 3$	0	0	2	3	$\boxed{3}$	5	6	6	6
$k = 4$	0	0	0	3	3	$\boxed{5}$	6	6	6
$k = 5$	0	0	0	0	4	5	6	7	$\boxed{6}$

3 Optimization of the Dynamic Programming Scheme of PTA

As explained in the previous section, the PTA algorithm uses a dynamic programming technique to compute two matrices, the error matrix $\mathbf{E}_{c \times n}$ and the split point matrix $\mathbf{J}_{c \times n}$, with c columns and n rows where c is the size of the reduction and n is the number of tuples in the input relation.

In this section, two reductions of the search space of the dynamic programming algorithm are proposed.

3.1 Existing Optimizations

Before presenting the optimizations of this work, the optimizations PTAc already uses to reduce the number of computations are briefly summarized.

The first approach is based on the presence of gaps (either temporal or based on mismatch in grouping attribute values). If a sequential ITA relation is gapped some reductions are not possible since PTA is designed not to merge non adjacent tuples.

To retrieve these tuples a gap vector $\mathbf{G}$ is used. This vector stores the positions of non adjacent tuple pairs in the sorted ITA input relation.

Example 13. In the running example the first 6 tuples of the ITA argument relation can not be reduced to a size of 1 since there is a gap between tuples s_5 and s_6. Therefore value $\mathbf{E}_{1,6}$ will lead to an in finite error. This holds also for $\mathbf{E}_{1,7}, \mathbf{E}_{1,8}, \mathbf{E}_{1,9}$. The gap vector has value $\mathbf{G} =< 5, 6 >$ since $s_5 \not\prec s_6 \not\prec s_7$.

Intuitively if the number of non adjacent tuple pairs in $\mathbf{s}_i$ is greater than k then merging is not possible. Therefore the k-th element of the gap vector indicates the size of the set $\mathbf{s}_i \in \mathbf{s}$ that can be reduced to k. All computations for $E_{k,i}$ where $\mathbf{G}_k > i$ will give an infinite error. In other words $i_{max} = \mathbf{G}_k$ indicates an upper bound to i. If $k > |\mathbf{G}|$, i.e. $\mathbf{G}_k$ does not exist, i cannot be upper bounded and therefore $i_{max} = |\mathbf{s}|$.

Example 14. Assume k has value 2, $\mathbf{G}_2$ has value 6, therefore $\mathbf{s}_6 = \{s_1, \ldots, s_6$ can be reduced to size 2. All computations of $\mathbf{E}_{2,i}$ with $i > 6$ can be avoided and the respective error is ∞. For $k >= 3$, since $|\mathbf{G}| = 2$, $i_{max} = |\mathbf{s}| = 9$

The second approach defines a lower bound for variable j. When computing $\mathbf{E}_{k,i}$ for $k > 1$ the term $SSE(\mathbf{s}_i \backslash \mathbf{s}_j, \rho(\mathbf{s}_i \backslash \mathbf{s}_j, 1))$ has to be evaluated for varying j. This is the error in reducing the tuples $\mathbf{s}_i \setminus \mathbf{s}_j = \{s_j + 1, \ldots, s_i\}$ to size 1. This is only possible if $\mathbf{s}_i \setminus \mathbf{s}_j$ has no gaps. A gap is enclosed if j is lower than the rightmost

non adjacent tuple pair in $\mathbf{s}_i$, i.e. $j_{min} = \max\{\mathbf{G}_l | \mathbf{G}_l < i \wedge l = 1, \ldots, |\mathbf{G}|\}$. If $\mathbf{s}_i$ contains no gaps j_{min} is set to $k-1$.

Example 15. Assume $i = 7$, $k = 4$, j ranges therefore between 3 and 7. A reduction of the first 7 tuples $\mathbf{s}_7$ to size 3 can have the following solutions: $\rho(\mathbf{s}_3, 3) \cup \rho(\mathbf{s}_7 \setminus \mathbf{s}_3, 1)$, $\rho(\mathbf{s}_4, 3) \cup \rho(\mathbf{s}_7 \setminus \mathbf{s}_4, 1)$, $\rho(\mathbf{s}_5, 3) \cup \rho(\mathbf{s}_7 \setminus \mathbf{s}_5, 1)$, $\rho(\mathbf{s}_6, 3) \cup \rho(\mathbf{s}_7 \setminus \mathbf{s}_6, 1)$. In the first three solutions the second part has as argument of the reduction function a relation which includes gaps and can therefore not be merged, only the fourth solution with split point value $j = 6$ is admissible. In the gap vector $\mathbf{G}$ the rightmost element with value < 7 has value 6. Therefore j can be lower bounded to $j_{min} = 6$.

3.2 Diagonal Pruning

During the computation of the error matrix some elements are reused in further computation steps as explained in section 2.3.7. To compute $\mathbf{E}_{k,i}$ the already computed elements $\mathbf{E}_{k-1,j}$ with j ranging from $j_{min} \ldots i$ are reused. It is intuitively evident, that for computing the end result at $\mathbf{E}_{c,n}$, corresponding to the reduction of an input relation of size n to size c it is not necessary to compute the value $\mathbf{E}_{c-1,n}$ since it can never be reused in computing the values of row c. This holds for all elements with a column index $i > n - (c - k)$. Therefore not only the computation of elements in the lower left corner of the $\mathbf{E}$ matrix can be omitted but also the elements in the upper right corner of the matrix.

In addition to the bounds for i and j defined by PTA for variable i can be defined a further bound.

Lemma 1. *For the computation of the matrices* $\mathbf{E}$ *and* $\mathbf{J}$ *the variable* i *has an upper bound which is* $n - (c - k)$.

$$\mathbf{E}_{k,i} = \begin{cases} \min_{k-1 \leq j < i}\{E_{k-1,j} + \\ \quad SSE(\mathbf{s}_i \setminus \mathbf{s}_j, \rho(\mathbf{s}_i \setminus \mathbf{s}_j, 1))\} & k > 1 \wedge (i \leq n - (c - k)) \\ SSE(\mathbf{s}_i, \rho(\mathbf{s}_i, 1)) & k = 1 \wedge s_1 \prec \cdots \prec s_i \\ \infty & k = 1 \wedge \neg(s_1 \prec \cdots \prec s_i) \end{cases}$$

Proof. To compute elements in row c the maximum range for j is $c - 1 \ldots n - 1$. Therefore elements $\mathbf{E}_{c-1,c-1}, \ldots \mathbf{E}_{c-1,n-1}$ are arguments of the minimum function while $\mathbf{E}_{c-1,n}$ is not used. Hence the computation of element $\mathbf{E}_{c-1,n}$ can be avoided. To compute the row $k = (c-1)$ variable j varies between $c-2$ and $n-2$. Therefore the rightmost argument element of the min function is $\mathbf{E}_{c-2,n-2}$. In general this holds for all elements computed in row k and the rightmost used element in row $k-1$ is shifted by 1 in left direction. Reaching at row $k = 1$ the rightmost accessed element is $\mathbf{E}_{1,n-(c-1)}$.

It follows that all elements $\mathbf{E}_{k,i}$ for which $i < n - (c - k)$ are not used in the computation of the next row. □

Example 16. Considering a reduction of the input relation to size $c = 5$ to compute the final element $\mathbf{E}_{5,9}$ the element $\mathbf{E}_{4,9}$ is not needed. This can be extended to the lines above, $\mathbf{E}_{3,8}$ and $\mathbf{E}_{3,9}$ are superfluous. In table 6 all elements that are not strictly needed to compute the end result are shown highlighted by a frame.

Table 6: **E** matrix diagonal pruning

	$i=1$	2	3	4	5	6	7	8	9
$k=1$	0	106667	107500	475000	612143	∞	∞	∞	∞
$k=2$	–	0	45000	107500	109167	612143	∞	∞	∞
$k=3$	–	–	0	45000	46667	109167	612143	687143	694493
$k=4$	–	–	–	0	1667	46667	109167	184167	191517
$k=5$	–	–	–	–	0	1667	46667	109167	129017

3.2.1 Improvements

The number of pruned computations is $\frac{(c-1)*(c-2)}{2}$, thus independent of the size of the input relation n but depends on the size of the reduction c.

The computation effort increases with increasing column index i. For every value with $k > 1$ the term $\min_{k-1 \leq j < i}\{E_{k-1,j} + SSE(\mathbf{s}_i \setminus \mathbf{s}_j, \rho(\mathbf{s}_i \setminus \mathbf{s}_j, 1))\}$ has to be evaluated.

Example 17. For example to compute $\mathbf{E}_{3,4}$ the minimum has to be searched among two values, one for j=2 and one for j=3. In this case it is assumed that no gaps are present in the argument relation, therefore $j_{min} = k - 1$. For $\mathbf{E}_{3,7}$ instead it is necessary to evaluate the term 5 times for j ranging from 2 to 6. Therefore the computational improvement in avoiding computations for elements with a higher i index should be greater than avoiding the computation for lower i values, which is already implemented in the PTA algorithm by limiting $i_{min} = k$.

If the argument relation contains non adjacent tuple pairs the computation of some values of the matrices are avoided by the upper bound i_{max}. These cells are situated on the right end of each row matrix, thus the same cells indicated by diagonal pruning. Therefore for gapped datasets effectiveness of diagonal pruning should decrease since the avoided computations overlaps. As illustrated in table 6 nodes with value ∞ are the nodes for which the computation is avoided by presence of gaps in the dataset.

3.2.2 Space Complexity

For both matrices the elements in the upper right corner and the lower left corner (highlighted as framed cells in table 7) can be pruned and therefore they have not to be stored. Thus, in each row it is necessary to store $n - c + 1$ elements. The size of the **J** matrix in PTAc is given by nc since in each row there are stored n elements. The size of the matrices is in the worst case therefore n^2. With diagonal pruning the size of each row can be reduced to $(n - c + 1)c$. The matrix reaches its maximum number of nodes at $c = \frac{n+1}{2}$ and is $\frac{n^2+2n+1}{4}$
.

Proof.

$$f(n, c) = (n - c + 1)c$$

$$\frac{\partial f}{\partial c} = n - 2c + 1$$

$$\frac{\partial f}{\partial c} = 0 \text{ for } c = \frac{n+1}{2}$$

which is a local maxima in the range $[1, n]$

$$f(n, \frac{n+1}{2}) = \frac{n^2 + 2n + 1}{4}$$

□

Space complexity remains therefore quadratic $\mathcal{S}(\frac{n^2+2n+1}{4}) = \mathcal{S}(n^2)$.

In table 7 the eliminate-able cells are shown which are framed by a solid line. The matrix can be resized if each row k is shifted by k cells to the left.

Table 7: Eliminate-able cells with diagonal and last row pruning

	$i = 1$	2	3	4	5	6	7
$k = 1$	0	26,666	67,500	208,333	269,285	∞	∞
$k = 2$	−	0	5,000	41,666	49,166	269,285	∞
$k = 3$	−	−	0	5,000	6,666	49,166	269,285
$k = 4$	−	−	−	0	1,666	6,666	49,166

The error matrix **E** does not require the same amount of memory as matrix **J**. To compute the k-th row only the results of row $k-1$ are reused. Therefore only the last two rows of the matrix have to be stored and the matrix can be replaced by two arrays. Therefore the space complexity to store **E** is linear with the size of the input relation. Unfortunately this does not hold for the **J** matrix where only after the last computation the split path is decided, hence the whole matrix has to be kept until the last element for $\mathbf{E}_{k,n}$ and $\mathbf{J}_{k,n}$ is computed.

3.3 Pruning in Last Row

PTA computes all values from $i = k$ to n where i has an upper bound as described in the previous section or by evaluation of the gap vector $\mathbf{G}$. Both bounds do not take effect for computation of row $k = c$. The size of the gap vector $\mathbf{G}$ is always lower than c, therefore $i_{max} = n$. Also the upper bound for i in diagonal pruning is equal to $n - (c - k) = n$.

Since row c is the last computed row the values can not be reused in further computation steps. The only cell that is needed is element $\mathbf{J}_{c,n}$ and therefore $\mathbf{E}_{c,n}$, which is the starting point of the reduction.

As shown in table 7 the nodes framed by a dotted line are not needed to compute any further result but are computed by PTAc without optimization.

Therefore for the computation of the row $k = c$ the variable i can be lower bounded at value n.

Lemma 2. *To compute the error and split point matrix in the last row ($k = c$ only the element $\mathbf{E}_{c,n}$ has to be computed, for all elements $\mathbf{E}_{c,i} | i < n$ the computation can be omitted.*

$$\mathbf{E}_{k,i} = \begin{cases} \min_{k-1 \leq j < i}\{E_{k-1,j} + \\ \quad SSE(\mathbf{s}_i \setminus \mathbf{s}_j, \rho(\mathbf{s}_i \setminus \mathbf{s}_j, 1))\} & k > 1 \wedge k < c \wedge (i \leq n - (c - k)) \\ SSE(\mathbf{s}_i, \rho(\mathbf{s}_i, 1)) & k = 1 \wedge s_1 \prec \cdots \prec s_i \\ \infty & k = 1 \wedge \neg(s_1 \prec \cdots \prec s_i) \\ \min_{c-1 \leq j < n}\{E_{c-1,j} + \\ \quad SSE(\mathbf{s}_n \setminus \mathbf{s}_j, \rho(\mathbf{s}_n \setminus \mathbf{s}_j, 1))\} & k = c \wedge i = n \end{cases}$$

The proof is similar to the proof of the previous lemma and is therefore omitted.

Both optimizations described reduce the computational effort of the algorithm. However the number of avoided computations for diagonal pruning depends only on c, the size of the reduction. For small reductions only a very limited number of computations is avoided. For last row pruning instead the number of avoided computations depends on the size of the relation n and on the reduction size c. The number of non computed cells is $n - c - 2$.

4 Split Point Graph as Alternative to Split Point Matrix

In this section the second problem of PTAc, the quadratic space complexity is addressed. First the concept of split point is analyzed and the concept of split path is introduced. A crucial point is then the description of a new representation of the matrix $\mathbf{J}$ the Split Point Graph (SPG).

4.1 Split Path

The end result of a PTAc query is the reduction of the intermediary ITA relation. The points indicating where merging steps occur is therefore the crucial information to generate the reduction itself. These points are given by the split points stored in the matrix $\mathbf{J}$. After computation of the element $\mathbf{J}_{c,n}$ a list of split points can be obtained. In the matrix $\mathbf{J}$ are stored all possible split points a reduction may encounter. The construction of the output relation begins with split point at $j_c = \mathbf{J}_{c,n}$ and indicates the first split for the reduction. All tuples following the split point j_c point are merged to size 1. The remaining tuples (all tuples with index $i \leq j_c$, $\{s_1, \dots s_{j_c}\}$) are then reduced to size $c - 1$. The next split point $j_{c-1} = \mathbf{J}_{c-1,j_c}$. The iterations are repeated until a split point with value 0 is reached. The obtained sequence is named *Split Path.*

Example 18. In the example relation for $c = 5$ the element $\mathbf{J}_{c,n}$ has value 6. This means that in order to reduce n tuples to size $c = 5$, all tuples after s_6 are merged into one tuple. In the next step the remaining tuples $(s_1 \dots s_6)$ have to be reduced to size $c = 4$. Therefore the next split point is found at $\mathbf{J}_{(4,6)}$ with value 5, followed by $\mathbf{J}_{(3,5)}$ with value 3. In table 5 is illustrated the split point matrix for the example data. The split points are highlighted, the split path is $[6, 5, 3, 1]$. The output relation is therefore $\mathbf{z} = (s_1, s_2 \oplus s_3, s_4 \oplus s_5, s_6, s_7 \oplus s_8 \oplus s_9)$.

The output relation is then reconstructed by merging all tuples lying between the discovered split points.

Example 19. The split path [6, 4, 3, 1] of the running example can be visualized as depicted in figure 4. The reduction is obtained by merging tuples $s_2 \oplus s_3$, tuples $s_4 \oplus s_5$ and tuples $s_7 \oplus s_8 \oplus s_9$.

For the computation of a PTAc query it is necessary to keep the whole matrix with all intermediary results. This is motivated by the computation of the starting point in the last computed element $\mathbf{J}_{c,n}$. A different value in this element would lead to different split paths.

Example 20. Assume that in the example the last split point was 5 instead of 6, this would lead to different merging steps, i.e. $[5, 3, 2, 1]$ equivalent to

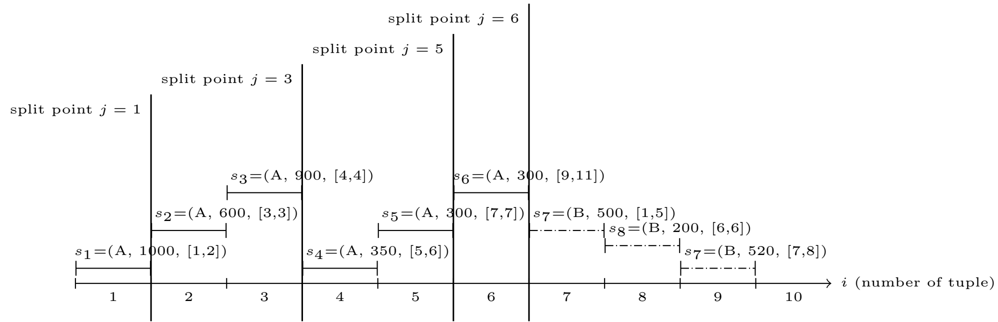


Figure 4: Splitpoints for running example, reduction to $c = 5$

$\mathbf{z} = (s1, s_2, s_3, s_4 \oplus s5, s_6 \oplus s_7 \oplus s_8 \oplus s_9)$.

4.2 Graph Representation of the Matrix J

Another way to represent the split points of the partial reductions is a graph. The split points are not indicated by the value of a matrix element but by edges between nodes. The nodes are organized in levels that are equivalent to the rows in the split point matrix. Each node is labeled with the number of the tuple in the sequential relation which therefore ranges between 1 and n. A node $N_{k,i}$ indicates node with index i at level k. Each node on level $k = 1$ has no outgoing edges. These nodes are called the terminating nodes of the split path. The information the graph captures is the same of the split point matrix $\mathbf{J}$. In order to reduce the first i tuples to size k the edges from the starting node $N_{k,i}$ have to be followed until reaching one of the terminating nodes at level 1. The final split path is then obtained by following the edges from source node $N_{c,n}$.

Example 21. The graph for the example relation is shown in figure 5. Each node highlighted by a dotted pattern can be eliminated through *diagonal pruning*, see section 3. An edge between node $N_{5,9}$ and node $N_{4,6}$ indicates that in order to reduce 9 tuples from size 5 to size 4 the optimal split point is after tuple s_6. The remaining tuples from $s_1 \dots s_6$ are reduced to size $c = 4$. The necessary split point is indicated by the edge outgoing from node $N_{4,5}$ which points to node $N_{3,5}$. Following all edges from $N_{5,9}$ to level 1 the resulting split path is $[6, 5, 3, 1]$.

If a node on level $k > 1$ has no outgoing edge this means that a reduction indicated by the node itself is not possible. This is the case for all nodes where $i < k$, i.e. all reductions where the size k is greater than the number of tuples. Another case for non possible reduction is founded due to gaps. A subset of the sequential argument relation with size j can not been reduced to a size k if the

subset contains more than $k-1$ non adjacent tuple pairs.

Example 22. For example node $N_{2,1}$ has no outgoing edge because it is not possible to reduce the first 1 tuples to a size of $c = 2$. It is neither possible to reduce the first 7 tuples to size $c = 2$ due to gaps in the ITA result set. Therefore $N_{2,7}$ has no outgoing edge. Remember that $s_5 \not\prec s_6$ and $s_6 \not\prec s_7$. Therefore the minimum reduction size of tuples $s_1 \dots s_7$ is 3.

Each node can have zero or multiple incoming edges. No incoming edges denotes split points that cannot be reached from any node of the next level. Therefore these nodes cannot make part of any split path.

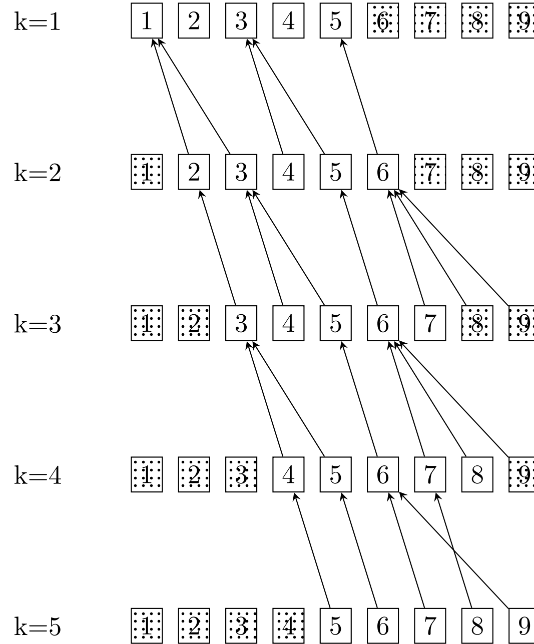


Figure 5: Split point graph for running example

4.3 Graph Creation and Insertion of new Entries

The graph is created by inserting the nodes in each level for $k = 1 \dots c$. For $k = 1$ nodes with label 1 to $n - c + 1$ are created. Each of these nodes has no outgoing link, hence no edge has to be added.

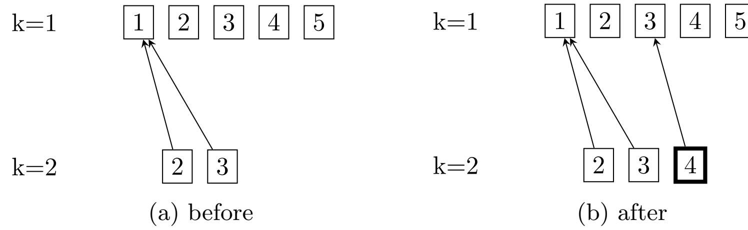
For $k > 1$ nodes with labels from k to $n-(c-k)$ are created and after the node creation the edge corresponding to the split point has to be inserted. Since each node has exactly one outgoing edge it is appropriate to unify in a single step node creation and edge insertion. Three parameters are needed for this step:

number of the level the node is inserted (k), the index of the node and the index of the destination of the edge, i.e. the split point. The destination is always on the adjoining upper level $k-1$.

Example 23. The necessary steps to insert a node with parameters $k=2$, index 4, destination 2, i.e. node $N_{2,4}$ with edge pointing to $N_{1,2}$ are:

1. *newnode* = add new node $N_{2,4}$ to level $k=2$
2. *destnode* = search node with index *destination* on level $k=1$, the result is $N_{1,2}$
3. add edge between *newnode* and *destnode*

Figure 6 shows the graph status before and after inserting the node and edge.



(a) before (b) after

Figure 6: Node insertion in Split Point Graph (example 23)

Figure 7 shows the complete evolution of the split point graph for the running example.

4.4 Search of Destination Nodes

At each level i the nodes are inserted sequentially from index $j = i \dots n-(c-i)$. For each newly inserted node $N_{i,j}$ the destination node has to be retrieved among the nodes of the level $i-1$. Not all nodes have to be included in the search space. The edge outgoing from the inserted node $N_{i,j}$ points to node $N_{i-1,l} | i-1 < l \leq j$. If $l < i-1$ a non possible reduction of the first $i-1$ tuples to a size greater than $i-1$ is requested in the next step. The second condition is motivated by the fact that the split point following split point j is necessarily lower than j.

Example 24. Considering the example relation. To insert node $N_{3,5}$ the nodes at $i=2$ from index $j=2$ to index $j=4$ have to be searched, see figure 8.

The nodes at each level are inserted in ascending order and therefore the set is sorted. The complexity for searching nodes in this set is logarithmic to the number of elements. For inserting nodes at a certain level k the index of the

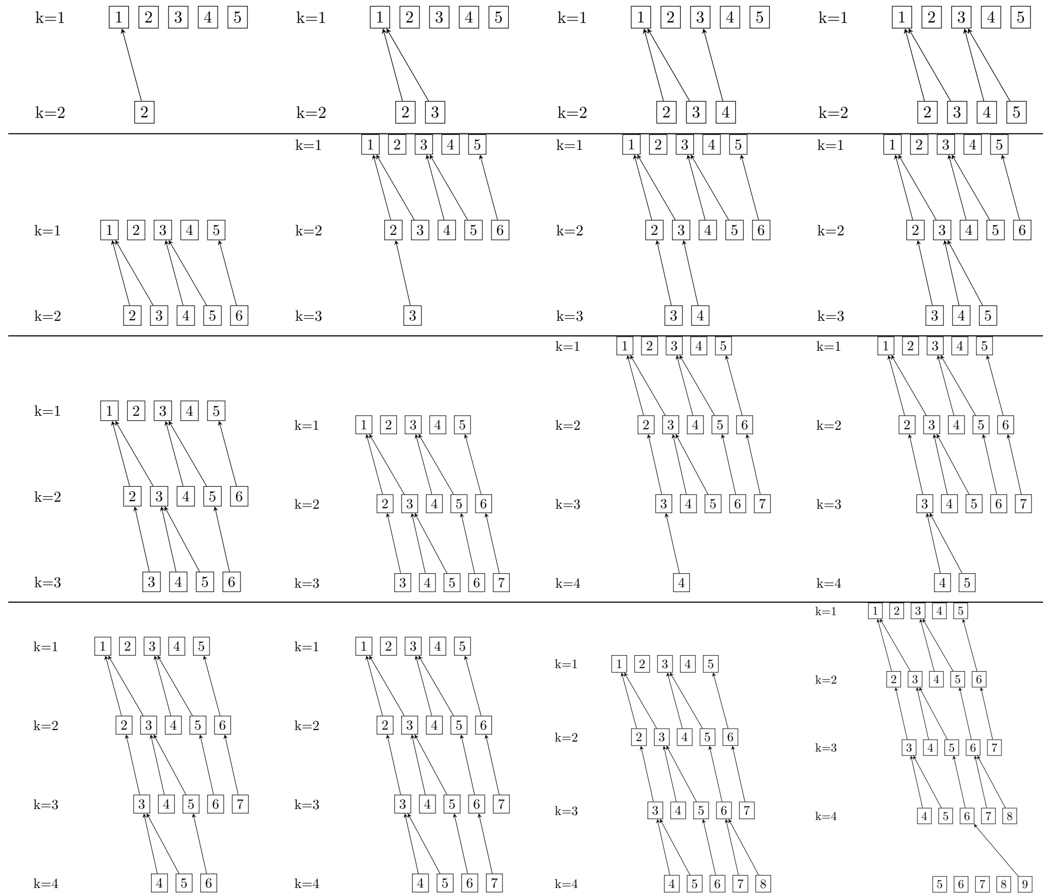


Figure 7: Evolution of Split Point Graph of the running example

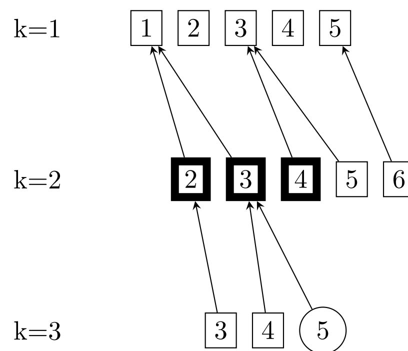


Figure 8: Search space when inserting a node $N_{3,5}$ with destination node 3

node i ranges from k to $n-(c-k)$, therefore $n-c$ nodes have to be inserted in the graph. For the first node with index $i=k$ the search space at level $k-1$ is only the node $N_{k-1,k-1}$. For the second node with index $i=k+1$ the search space at level $k-1$ is given by nodes $N_{k-1,k-1}\ldots N_{k-1,k}$.

The computational complexity of searching step in node inserting is for each level $\sum_{i=1}^{n-c}\mathcal{O}(\log i)=\mathcal{O}(n\log n))$.

Since at most n levels have to be computed the worst case complexity for the searching step over all levels is equal to $\mathcal{O}(n^2\log n)=\mathcal{O}(n^2)$.

In section 5.2 a solution is proposed in order to reduce this quadratic complexity to linear.

4.5 Optimization of SPG

In the following optimizations for the split point graph are described. In particular the size of the graph can be reduced after all nodes of a level are inserted.

4.5.1 Orphaned Elements in Split Point Matrix J

Not all split points contained in the matrix **J** are useful for the computation of the final result. Non-used elements which are superfluous for the generation of the split path are called orphaned. A split path is constructed by following the split points starting at element $\mathbf{J}_{c,n}$ until reaching the element in column 1. It is noticeable that some elements can never be reached.

Example 25. In the running example the split point $\mathbf{J}_{2,4}$ cannot be reached by any element in row 3. In the whole row $k=3$ no element with value 4 is contained. Therefore the element is orphaned. Table 8 highlights by framing all orphaned nodes.

Table 8: Not reachable split points of the example relation

	$i=1$	2	3	4	5	6	7	8	9
$k=1$	0	0	0	0	0	0	0	0	0
$k=2$	0	1	1	3	3	5	0	0	0
$k=3$	0	0	2	3	3	5	6	6	6
$k=4$	0	0	0	3	3	5	6	6	6
$k=5$	0	0	0	0	4	5	6	7	6

4.5.2 Elimination of Superfluous Nodes: Path Pruning

The orphaned nodes, as explained in section 4.5.1, are equivalent to the nodes in the split point graph that have no incoming edges. If a node has no incoming edge it can never be part of a valid split path and therefore it can be removed from the graph. The nodes that can be removed by diagonal pruning, i.e. the nodes $N_{k,i}$ with $i > (n - (c - k))$ have no incoming edge. An exception is the start node $N_{c,n}$.

Lemma 3. *The elimination of some nodes $N_{k,i}$ with edge pointing to $N_{k-1,j}$ can have as a consequence the elimination of nodes $N_{k-1,j}$.*

Proof. If a node $N_{k,i}$ is removed, the outgoing edge to node $N_{k-1,j}$ has to be removed, too. Each node can have at most 1 outgoing edge but multiple incoming edges. Therefore node $N_{k-1,j}$ can lose all incoming edges iff $N_{k-1,j}$ has only one incoming edge. If $N_{k-1,j}$ has no incoming edge it can be deleted, otherwise not. □

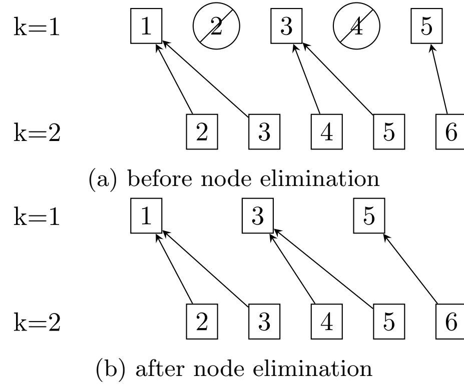
Example 26. In the example graph shown in figure 5 each node without incoming edge can be removed. This is the case for nodes $N_{1,2}, N_{1,4}, N_{2,4}, N_{3,4}, N_{4,8}$, $N_{5,5}, \ldots, N_{5,8}$. It is also shown that the nodes removed by diagonal pruning have no ingoing edge.

The deletion of node $N_{k-1,j}$ can lead to a deletion of a node $N_{k-2,l}$ by following the edges. This repetition terminates if either a terminating node at level 1 is reached or if a node is not deleted, hence no changes to the lower level are made. The elimination of some nodes and, if existing also their connected parents, is named *path pruning*. If after computing all nodes of a certain level k the graph is resized through path pruning, the graph contains only nodes that make part of a valid partial split path and the size is therefore minimal. After computation of the last node $N_{c,n}$ the split path of the result can be obtained.

Example 27. Figure 9 shows the split point graph after computation of level $k = 2$. Nodes $N_{1,3}$ and $N_{1,3}$ are not connected to any node at level $k = 2$ and they therefore be removed.

After computation for $k = 3$ the nodes at $k = 2$ without incoming edge can be eliminated. In the running example this is the case for node $N_{2,4}$ as shown in figure 10. Since the deletion of this node also can have an influence on the nodes at level $k = 1$ it is necessary to descend the graph to $k = 1$ and check if the node with incoming edge from the deleted node $N_{2,4}$ has no more incoming edges. If this is so, the node can also be removed. In the example node $N_{1,3}$ cant be removed since it has another incoming link from node $N_{2,5}$

After computation for $k = 4$ the nodes $N_{3,4}$ and $N_{3,7}$ can be removed. Deletion of $N_{3,7}$ leads to deletion of node $N_{2,6}$ and descending the graph also node $N_{1,5}$ is removed (figure 11).



(a) before node elimination

(b) after node elimination

Figure 9: Split point graph for running example after computation step k=2

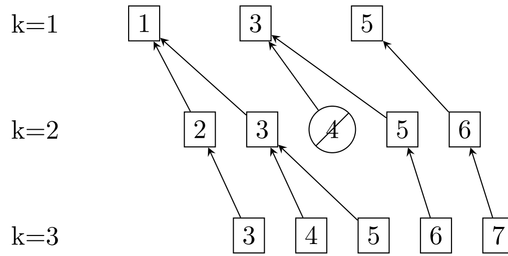


Figure 10: Split point graph for running example after computation step k=3

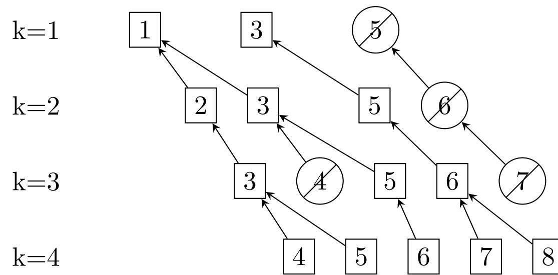


Figure 11: Split point graph for running example after computation step k=4

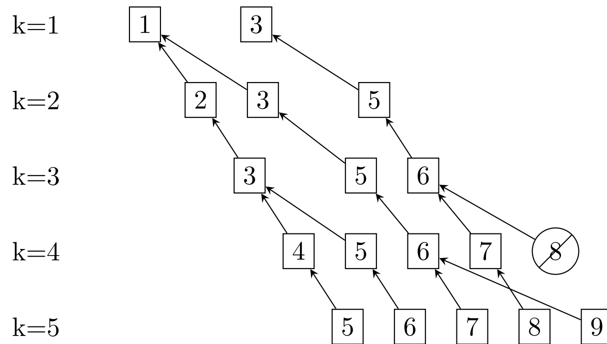


Figure 12: Split point graph for running example after computation step k=5

After insertion of all elements in the last level $k = c$ only node $N_{4,8}$ can be removed. In figure 12 the graph after insertion of all nodes is shown. If only node $N_{c,n}$ is computed which is the necessary element to obtain the split path as described in section 3.3 the graph reduces after path pruning step to only one split path as shown in figure 13. Since this step is not necessary for determining the split path of the result, this step can be omitted.

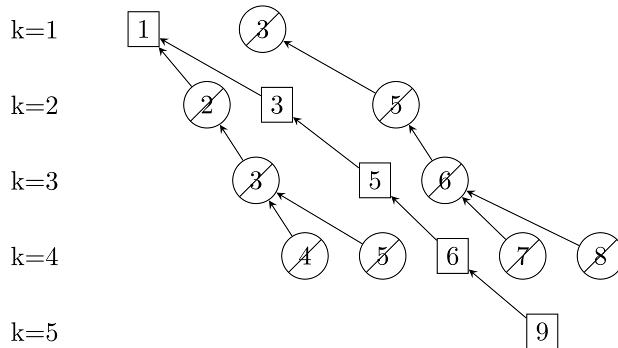


Figure 13: Split point graph for running example after computation step k=5, only node 9 is computed

4.5.3 Path Pruning Algorithm

After the insertion of all elements of a k-level the path pruning step can be executed. The nodes on the level $k-1$ have to be accessed directly and are the starting nodes of path pruning. Path pruning can be formulated as a recursive function, see algorithm 1, called for every node on level $k-1$.

Algorithm 1: Function pathPrune

```
1 Function pathPrune(node)
2     if node has no incoming edges then
3         next = follow edge from node;
4         delete node;
5         if next exists then
6             pathPrune (next);
```

4.6 Complexity Analysis

In this section the dependence of the graph size on the parameters n and c of the reduction is analyzed. Also the effectiveness of path pruning is discussed.

4.6.1 Graph Size

In the graph only the edges and therefore also the split points which are computed are inserted. If a node is not reached by the next level the edge is not inserted, hence no additional space is used in the graph. In the split point matrix **J** instead all possible split points are stored, even if they can never be reached from the higher levels. Avoiding unnecessary computations of **E** and therefore values for **J** limits the size of the split point graph.

The number of nodes in the split point graph depends on n, the size of the input relation, and on c, size of the reduction. As evaluated in section 3.2.2 the theoretical size of the split point matrix is $(n-c+1)c$, reaching its maximum with $\frac{n^2+2n+1}{4}$ at $c = \frac{n+1}{2}$. The number of nodes in the split point graph is growing on every step for variable k, reaching its maximum at $k = c$. Figure 14 shows the size of the graph for the reduction of an example dataset "climacubes" with size $n = 870$ to a reduction of $c = 700$. As expected it grows linear with increasing k-value. The size of the split point graph at k=700 is equal to the theoretical size, i.e. $(870 - 700) * 700 = 119,000$.

Figure 15 shows the measured size of the split point graph in a reduction of the same dataset for different reduction sizes (c). As expected the size of the graph has its maximum at $c = 436$ and is very low for c near to 1 and c near to n.

4.6.2 Node Pruning Efficiency

Node pruning reduces the size of the graph. The reduction efficiency depends on c, if $c \ll n$ the reduction is more efficient. The worst case for the path pruning efficiency is given if c is equal to n. In this case also the number of split

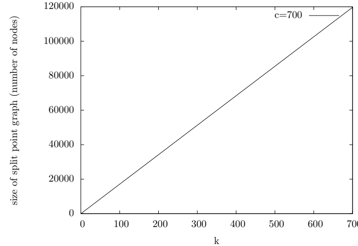


Figure 14: Size of split point graph dataset climacubes, n=870, c=700 for each k-step

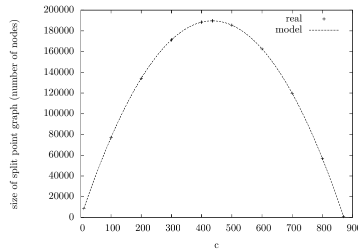


Figure 15: Theoretical Model and real measures of Split Point Graph size

points is equal to n, therefore no nodes in the split point graph with multiple incoming edges can exist. Therefore no nodes can be pruned. Despite this fact the number of nodes in the graph is equal to n because most insertions are avoided by diagonal pruning.

If c is very low some nodes have more incoming edges than others. These nodes are split points between tuples in the ITA relation where a merge would introduce a high error. In presence of non adjacent tuples the nodes this phenomena is even more evident. In fact a node is a split point between two tuples of the sequential ITA relation. Therefore it is always contained in the split path and this node has certainly an incoming edge. Therefore since the number of edges between two levels is fixed to $n - c + 1$, i.e. the number of nodes, other nodes must lose their incoming edges and therefore are eliminated.

This behavior is shown in figure 16. The graph depicts the maximum number of nodes in the graph at different reduction sizes of an example dataset. The number of nodes without path pruning is symmetrically distributed with maximum at $c = n/2 + 1$ as shown in figure 15. The distribution of nodes with path pruning is skewed to the right as expected.

Table 9: Split Point Graph maximum size, dataset SYNTH, n=1000

c	nodes without pruning	nodes with pruning	ratio
100	90100	12261	13.61%
200	160200	31901	19.91%
300	210300	53153	25.27%
400	240400	72838	30.30%
500	250500	87713	35.02%
600	240600	94952	39.46%
700	210700	91808	43.57%
800	160800	72877	45.32%
900	90900	45392	49.94%
mean			33.60 %

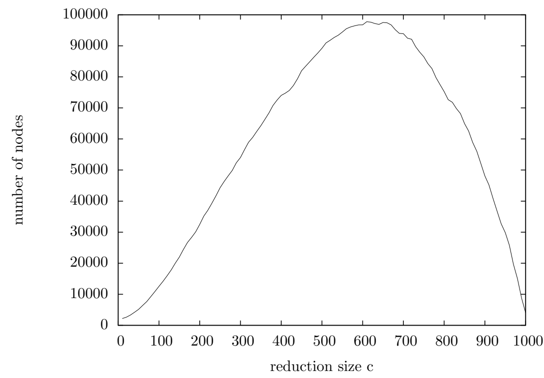


Figure 16: Maximum size of Split Point Graph, dataset SYNTH, n=1000

4.6.3 Memory Requirements

PTAc uses a matrix with dimension $|n \times c|$ of integer values to store the split points of the reduction. The matrix is instantiated before the computation is executed and the size of the matrix is data independent.

In contrast to this the split point graph solution uses a data structure that requires more space for storing a single split point (node with incoming edge) but the number of nodes are not fixed a priori. The graph is growing during the computation and the growing rate is data dependent due to the possibility of node pruning.

In the split point matrix each element requires 32 bits per single split point[1]. In an exemplary split point graph implementation each node requires 128 bits of memory as described in section 6.

If in the worst case the split point graph has $(n - c + 1)c$ elements. Under the assumption that in each of the path pruning steps none of the paths is pruned the total amount of memory necessary with matrix implementation is $32nc$ bits while with split point graph implementation $128(n - c + 1)c$ bits.

Therefore split point graph implementation uses less memory if $c > \frac{3}{4}n + 1$. Since the reduction size c is normally much smaller than n, $c \ll n$, the usage of split point graph implementation is mainly dependent on the effectiveness of the path pruning step.

Experiments showed that in average 65% of the nodes are removed in the path pruning step. By taking into consideration this node reduction rate the break even point for memory usage lowers to $\frac{4}{14}n + 1 = 0.28n + 1$. For $c < 0.28n$ the memory consumption is slightly greater but the difference is negligible. Figure 17 shows the memory consumption of matrix solution and split point graph with and without average path pruning of a dataset with $n = 1000$ in function of the reduction size c.

Since the node pruning rate for small values of c is higher than for large c-values as described in section 4.6.2 the break even point is reduced furthermore.

5 Algorithms

In the PTA algorithm the insertion of a split point was a simple assignment of a value in the **J**-matrix. This step is concerns lines 9, 15 and 22 of the PTAc Algorithm (see algorithm 2)

To substitute the split point matrix **J** with a split point graph only few modifications to the algorithm are necessary. The assignments to the matrix **J** have to

[1] It is assumed that the number of elements in the input relation is less than 2^{32}, otherwise int64 has to be used.

Algorithm 2: Algorithm PTAc

```
Input: r, A, F, c
1  s ← 𝒢^ITA[A, F]r;
2  Initialize G, L, S, SS;
3  Initialize E, J to ∞ and 0, respectively;
4  for k = 1 to c do
5      if k ≤ |G| then i_max = G_k else i_max = |s|;
6      for i = k to i_max do
7          if k=1 then
8              E_{1,i} ← SSE(S_i, ρ(S_i, 1));
9              J_{1,i} ← 0;
10         else
11             j_min ← max{k − 1, G_l | G_l < i ∧ l = 1, ..., |G|};
12             if G_{k−1} = j_min then
13                 j ← j_min;
14                 E_{k,i} ← E_{k−1,j} + SSE(s_i \ s_j, ρ(s_i \ s_j, 1));
15                 J_{k,i} ← j;
16             else
17                 for j = i − 1 to j_min do
18                     err_1 = E_{k−1,j};
19                     err_2 = SSE(s_i \ s_j, ρ(s_i \ s_j, 1));
20                     if err_1 + err_2 < E_{k,i} then
21                         E_{k,i} ← err_1 + err_2;
22                         J_{k,i} ← j;
23                     if err_2 > E_{k,i} then break;
24 z ← ∅, n ← |s|;
25 while c > 0 do
26     j ← J_{c,n};
27     z ← z ∪ {s_{j+1} ⊕ ... ⊕ s_n};
28     n ← j; c ← c − 1;
29 return z
```

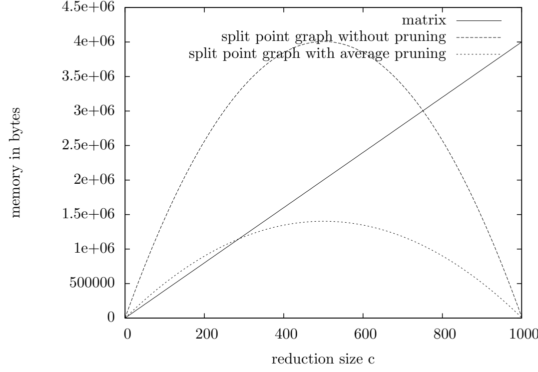


Figure 17: Memory usage theoretical model, n=1000

be modified, the graph pruning step has to be introduced and the reconstruction of the reduced output relation (lines 24 to 28) has to be adapted. Algorithm 5 shows the modified version of PTAc.

The function SplitPointGraphInsert (algorithm 3) is used for inserting new nodes into the split point graph. In this function the search of destination nodes is the most expensive step (section 4.4). In section 5.2 a solution to improve the search of nodes is proposed. Function SplitPointGraphPrune (algorithm 4) shows the procedure for the path pruning step. In both functions it is assumed that each node $N_{k,i}$ has three properties: i - the label of the element, *nextsplit* - edge to the element on the lower level (splitpoint), *numchilds* - a counter indicating how many incoming edge the node has. The whole graph is denoted with **SPG**, the set of nodes at level k is denoted as $\mathbf{SPG}_k$.

Algorithm 3: Function SplitPointGraphInsert

Input: index, nextsplit, k

1 newnode = add new node to $\mathbf{SPG}_k$ index *index*;
2 destnode = $node \in \mathbf{SPG}_{k-1} | node.index = nextsplit$;
3 add edge from newnode to destnode;

5.1 Data Structure for Split Point Graph

A possible data structure for the implementation of the split point graph can use pointers between single nodes. A node is represented by a triple (index, nextsplit, counter). Index is the label of the node, nextsplit is the edge to the next node at the next lower level. Counter is used for counting the number of incoming links and is necessary to facilitate the node pruning phase. Figure 18 shows the split point graph for the running example. All nodes, also theses

Algorithm 4: SplitPointGraphPrune

Input: k

1 **foreach** *node in* $\mathbf{SPG}_{k-1}$ **do**
2   startnode = node;
3   **while** *startnode!=NULL and startnode.numchilds==0* **do**
4   nextnode=startnode.nextsplit;
5   **if** *nextnode!=NULL* **then**
6   startnode.pathparent.numchilds–;
7   remove startnode; startnode=nextnode;

which are not are inserted in the graph, no paths have been pruned.

At each computation step for variable k the nodes are inserted sequentially and edges between the last two levels are inserted. Thus only the last two levels are modified. For the path pruning step the elements at level $k-1$ will be accessed sequentially and the path pruning step is started at the found nodes.

When all nodes of level k have been inserted and the graph pruning step has been executed all nodes at level $k-1$ have no longer to be accessed directly, each access can be done by following the incoming edges starting at level k.

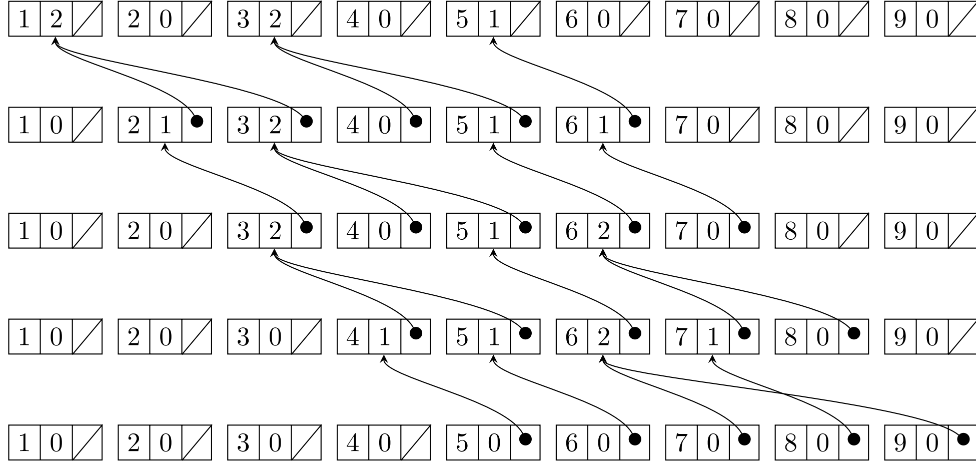


Figure 18: Split Point Graph implementation

5.2 Optimization of Node Insertion

As explained in section 4.4 the most complex computation step in inserting a new node is the search of the destination at level $k-1$. Since the elements in

Algorithm 5: Algorithm PTAc, modified for split point graph

Input: $\mathbf{r}, \mathbf{A}, \mathbf{F}, c$

1. $\mathbf{s} \leftarrow \mathscr{G}^{\mathrm{ITA}}[\mathbf{A}, \mathbf{F}]\mathbf{r}$;
2. Initialize $\mathbf{G}, \mathbf{L}, \mathbf{S}, \mathbf{SS}$;
3. Initialize $\mathbf{E}$ to ∞; **SPG** to empty graph;
4. **for** $k = 1$ **to** c **do**
5.   **if** $k \leq |\mathbf{G}|$ **then** $i_{\max} = \mathbf{G}_k$ **else** $i_{\max} = |\mathbf{s}|$;
6.   **for** $i = k$ **to** $i_{\max}$ **do**
7.   **if** k=*1* **then**
8. $\mathbf{E}_{1,i} \leftarrow SSE(\mathbf{S}_i, \rho(\mathbf{S}_i, 1))$;
9.   SplitPointGraphInsert(i, 0, 1);
10.   **else**
11. $j_{\min} \leftarrow \max\{k-1, \mathbf{G}_l | \mathbf{G}_l < i \wedge l = 1, \ldots, |\mathbf{G}|\}$;
12.   **if** $\mathbf{G}_{k-1} = j_{min}$ **then**
13. $j \leftarrow j_{min}$;
14. $\mathbf{E}_{k,i} \leftarrow \mathbf{E}_{k-1,j} + SSE(\mathbf{s}_i \setminus \mathbf{s}_j, \rho(\mathbf{s}_i \setminus \mathbf{s}_j, 1))$;
15.   SplitPointGraphInsert(i, j, k);
16.   **else**
17.   **for** $j = i - 1$ **to** j_{min} **do**
18. $err_1 = \mathbf{E}_{k-1,j}$;
19. $err_2 = SSE(\mathbf{s}_i \setminus \mathbf{s}_j, \rho(\mathbf{s}_i \setminus \mathbf{s}_j, 1))$;
20.   **if** $err_1 + err_2 < \mathbf{E}_{k,i}$ **then**
21. $\mathbf{E}_{k,i} \leftarrow err_1 + err_2$;
22.   SplitPointGraphInsert(i, j, k);
23.   **if** $err_2 > \mathbf{E}_{k,i}$ **then** break;
24.   SplitPointGraphPrune();
25. $\mathbf{z} \leftarrow \emptyset$, $n \leftarrow |\mathbf{s}|$;
26. startnode $\leftarrow \mathbf{SPG_{k,n}}$;
27. **while** *startnode.nextsplit != null* **do**
28. $j \leftarrow$ startnode.index;
29. $\mathbf{z} \leftarrow \mathbf{z} \cup \{s_{j+1} \oplus \ldots \oplus s_n\}$;
30.   startnode = startnode.nextsplit;
31. **return z**

this level are inserted in ascending order and therefore are sorted one solution to improve the search step is to use binary search. At each level at most n elements have to be inserted. The maximum number of levels is n, therefore the complexity is $\mathcal{O}(n^2 \log n)$.

A solution to reduce this complexity is to use two auxiliary arrays. One containing links to the nodes of level k and one containing links to the nodes of level $k-1$. This permits random access to the nodes of the last two levels.

At each computation step for level k nodes with index k to $n-c-k$ are inserted. Therefore at every level $n-c$ elements are inserted. The first element of the auxiliary array points to node k, the second element to node $k+1$. The nodes can therefore be accessed randomly through the auxiliary array by shifting the index position by k. Figure 19 shows the split point graph with auxiliary array.

The insertion of a new node can now be redefined as in algorithm 6.

Algorithm 6: Function SplitPointGraphInsert with auxiliary arrays

Input: index, nextsplit, k

1 newnode = add new node to $\mathbf{SPG}_k$ index *index*
2 last[index-k].node = newnode
3 destnode = ntl[nextsplit-k-1].node
4 add edge from newnode to destnode

An example of the used structure for the running example after computation for $k = 3$ is shown in figure 19. When computing a value for $k = 4$ the pointer to the destination node can be retrieved accessing the auxiliary array *ntl* (abbr. next to last). When all elements at level $k = 4$ are computed the array *ntl* is not needed any more and can be discarded, the array *last* becomes the new *ntl* and a new set of node objects for $k + 1$ as well as the auxiliary array *last* has to be instantiated.

6 Implementation

In the implementation used for the experiments a c++ standard vector of references to a node structure is used to store the references to the last two k-levels. These levels can be considered as the root nodes. After the computation of every level the vector instance levelkminus1 is destroyed, the node objects remain still valid because the destructor frees memory only for the vector object, the destructor for the node element objects is not called and has to be executed explicitly. This is done during the node pruning phase. The vector levelk is initialized after every computation step for variable k with a new set of $n-c+1$ elements.

Listing 1: Declaration of the auxiliary vectors

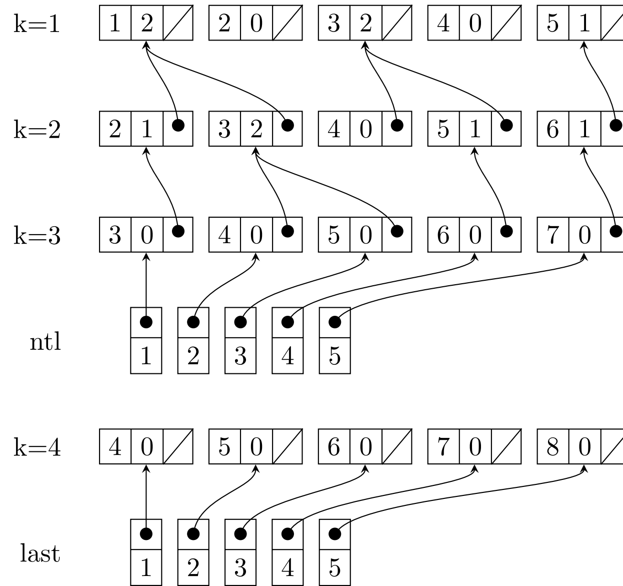


Figure 19: Split Point Graph implementation with auxiliary arrays

```
1  struct spgNode {
2          long indexNr;
3          spgNode* nextSplit;
4          long numChilds;
5  };
6  std::vector<spgNode*> levelk;
7  std::vector<spgNode*> levelkminus1;
```

Listing 2: Swap and initialization of the auxiliary vectors

```
1  levelkminus1 = levelk;
2  levelk = *new std::vector<spgNode*>;
3  for(long i=0;i<=(n-c+1);i++){
4          spgNode* newNode = new spgNode;
5          newNode->indexNr=i;
6          newNode->numChilds=0;
7          newNode->nextSplit = NULL;
8          levelk.push_back(newNode);
9  }
```

In the path pruning phase the nodes referenced by the vector at level $k-1$ are processed sequentially. Each referenced node with no incoming edges, i.e. with value for the variable numChilds equal to zero, is deleted. Before the deletion the counter of the incoming edges of the pointed node is decremented. The loop ensures that all connected nodes are deleted.

Listing 3: Path pruning function

```
1  void pathPruning(){
2      for(long i=0;i<=(n-c);i++){
3          spgNode* startNode = levelkminus1[i];
4          while(startNode->numChilds==0 && startNode!=NULL){
5              spgNode* nextNode=startNode->nextSplit;
6              if(nextNode!=NULL){
7                  startNode->pathParent->numChilds--;
8              }
9              delete startNode;
10             startNode=nextNode;
11         }
12     }
13 }
```

7 Experimental Evaluation

In this section investigations on the runtime performance and space requirements of the proposed modifications to the PTAc algorithm are presented.

7.1 Setup and Data

The experiments run on a OS X machine with one Core i5 1700MHz processor and 8GB of ram. The algorithms implemented in c++ used only one core of the dual core processor. A PostgreSQL 9.2 database running on the same machine is used as data storage medium.

For the experiments three different datasets were used: the synthetic employee temporal data set (ETDS) donated by F. Wang [Wan09], a synthetic dataset (SYNTH) and subset a dataset of temperature measurements of a building in Meran "climacubes" (CC).

The ETDS relation reports the evolution of employees in a company and contains 2875697 records. Each record stores employee number, sex, department, title, salary, and contract validity interval in months. The ITA queries over this relation are summarized in Table 10.

The SYNTH dataset consists of only one aggregation group, the values of the aggregation attribute are random values in the range [1;1000]. The interval duration of the single tuples are random values with range [1;40]. No temporal gaps are in the data.

CC relation reports temperature measurements of a building in Meran. Each record stores the temperature of 5 sensors, humidity and the value of the consumed energy in [kw/h] used for heating the building.

Table 10 reports the details of the used datasets.

Table 10: Datasets used for experiments

name	grouping attributes	aggregation function	ITA size	c_{min}
ETDS1	none	avg(salary)	6,393	1
ETDS2	empno,deptno	avg(salary)	2,844,050	331,551
ETDS3	deptno	avg(salary)	57,408	9
SYNTH	none	avg(value)	500,000	1
CC	none	avg(temp)	870	1

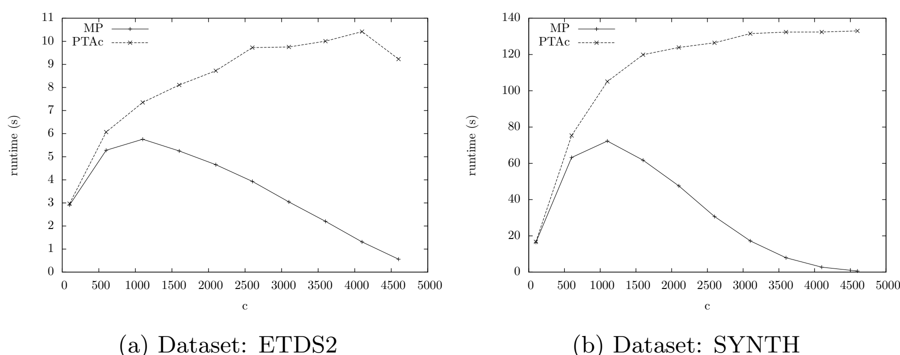
The unmodified PTAc algorithm is compared to the two introduced optimization. PTAc with matrix implementation and pruning (MP), PTAc with graph implementation and pruning (DP). The implemented algorithms used for the experimental evaluation are reported in table 11.

Table 11: Algorithm configuration used for experimental evaluation

Label	split point implementation	diaonal pruning	pruning last row
PTAc	matrix	no	no
MP	matrix	yes	yes
GP	graph	yes	yes

7.2 Runtime of Diagonal Pruning Approach

To evaluate the influence of the pruning optimization the algorithms PTAc and MP are compared. Figure 20 reports the runtimes of the two algorithms on the datasets SYNTH and ETDS2. The size of the input relation is 5000, the reduction size varies between 100 and 4600. As expected the improvement for low values for the reduction size c is negligible. Starting from a reduction to about 10% of the size of the argument relation the runtimes of MP are slightly lower than those of PTAc. For reduction sizes greater than 20% of the argument relation size the runtimes of MP decreases until approximating to 0 for reduction sizes c near to n.



(a) Dataset: ETDS2

(b) Dataset: SYNTH

Figure 20: Runtime in function of reduction size c, algorithms PTAc and MP

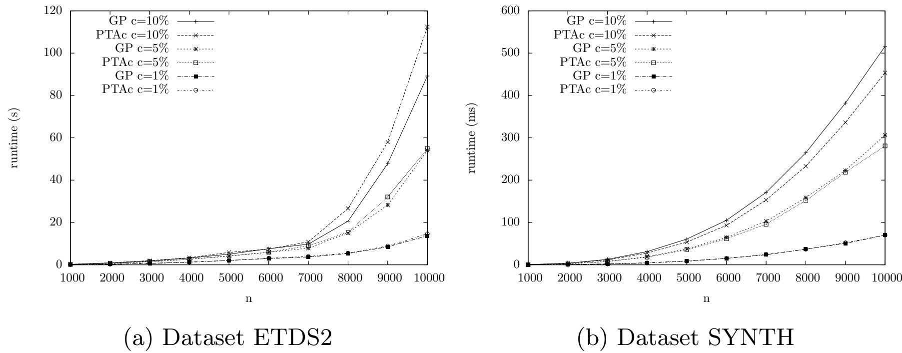
7.3 Runtime Split Point Graph Implementation

In this experiment the influence of the graph implementation on the runtimes of the aggregation is evaluated. As expected runtimes of the algorithm with split point graph implementation (GP) are not changed in respect to the unmodified algorithm (PTAc). Figure 21 reports the comparison of the algorithms GP and PTAc on the datasets ETDS2 and SYNTH at reduction rates of 1%, 5% and 10% for different sizes of the argument relation. At 1% the runtimes do not differ. At higher reduction rates due to diagonal pruning the GP algorithm is slightly faster. The experiments for the two datasets differ because of the data

dependence of PTA's optimizations. ETDS2 contains non adjacent tuples. Thus the search space reductions do partially overlap with the already defined bounds in the original PTAc algorithm causing a lower runtime reduction.

Table 12: Average reduction of the runtimes GP vs PTAc

Dataset	Reduction size in % of sequential relation		
	1%	5%	10%
ETDS2	-0.30%	-2.49%	-8.34%
SYNTH	-2.86%	-5.44%	-10.28%



(a) Dataset ETDS2

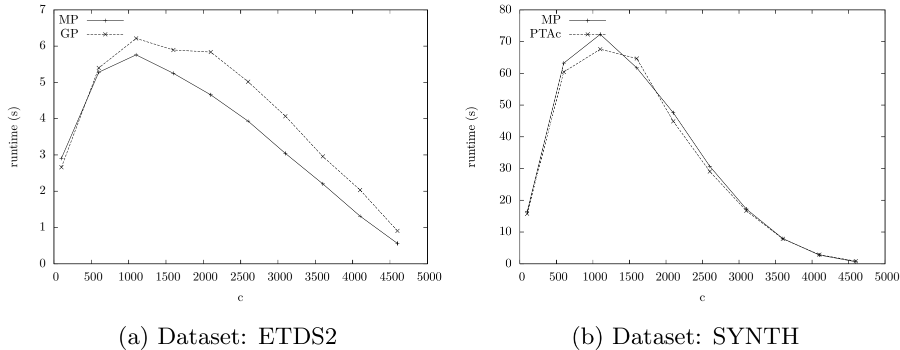
(b) Dataset SYNTH

Figure 21: Runtime in function of size of the argument relation n, algorithms PTAc and GP

The graph implementation (GP) has nearly the same runtimes as the algorithm with matrix implementation (MP). The graph implementation is slightly more complex than the matrix implementation since it requires the additional path pruning step. Figure 22 shows the comparison of the runtime of algorithm MP and GP. For ETDS2 a difference of average 25% is measured, for SYNTH instead the runtimes differ in average by only 1%. This difference is caused by the data dependence of the path pruning step. For the SYNTH dataset the path pruning phase is less effective, hence the time required for path pruning is negligible in relation to the overall computation.

7.4 Memory Requirements

The evaluation of the memory requirements considers the same dataset and algorithms as the previous evaluations. Algorithm DP reduces the memory consumption in respect to PTAc. For the comparison a node size of 128 bits and 32 bits / entry in split point matrix $\mathbf{J}$ is assumed. Table 13 reports the reduction rates of memory consumption for experiments on ETDS2 and SYNTH. The average value of the reduction is 67.6% for the SYNTH dataset and 76.52% for



(a) Dataset: ETDS2

(b) Dataset: SYNTH

Figure 22: Runtime in function of reduction size c, algorithms GP and MP

the ETDS2 dataset. The higher rate for the dataset ETDS2 is expected due to the presence of temporal gaps in the ITA argument relation as explained in section 4.6.2. Figure 23 shows the memory space used by the compared algorithms GP and PTAc on the datasets ETDS2 and SYNTH for different reduction rates.

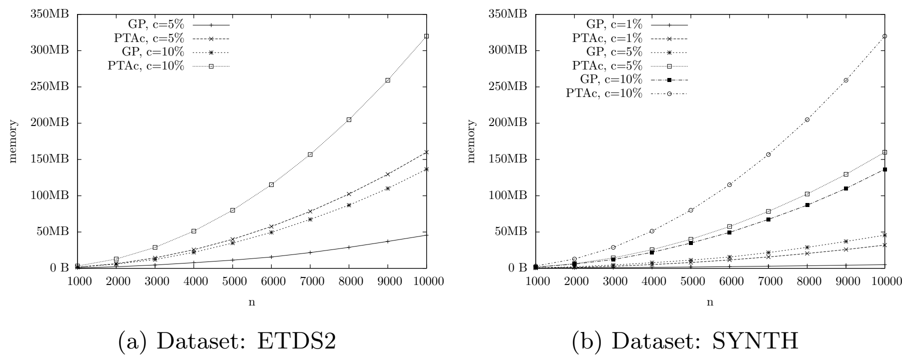
Table 13: Average reduction of the memory usage for algorithms GP and PTAc

Dataset	Reduction size in % of sequential relation			
	1%	2%	5%	10%
ETDS2	-70.8%	-80.5	-80.95%	-73.9%
SYNTH	-69.5%	-75.6%	-69.6%	-56.3%

Figure 24 shows the ratio between the memory of the graph implementation GP in respect to the matrix implementation MP. As expected for very small reduction sizes ($c < 10$) the graph implementation requires more memory. The minimum graph size is given by the number of nodes of the two last levels, $|SPG|_{min} \geq 2 * (n - c + 1)$. Each node requires 4 times more memory then an entry in the split point matrix. Therefore for small reduction sizes the overhead of the graph implementation is greater than the node reduction through path pruning. The reductions for very small values of c induce a high error and therefore are normally not used for approximation of the argument relation.

For reductions $c > 10$ the ratio between GP and MP decreases markedly. As expected the ratio increases for $c > 10$ until it reaches a local maximum at a reduction for $c = n/2$. In this case the theoretical number of nodes is maximal. Due to the data dependence of path pruning the maximum is not reached at $c = n/2$ for all datasets. For the SYNTH dataset the maximum is reached at $c = 1500$.

In nearly all cases, with expect to very high reduction rates to $c < 0.2\%$ of the



(a) Dataset: ETDS2 (b) Dataset: SYNTH

Figure 23: Memory consumption in function of argument relation size n, algorithms PTAc and GP

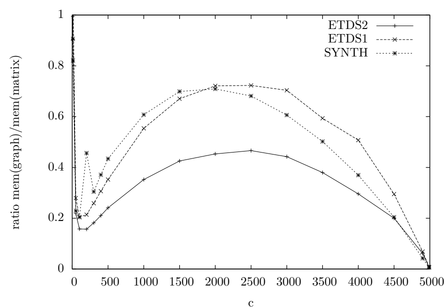
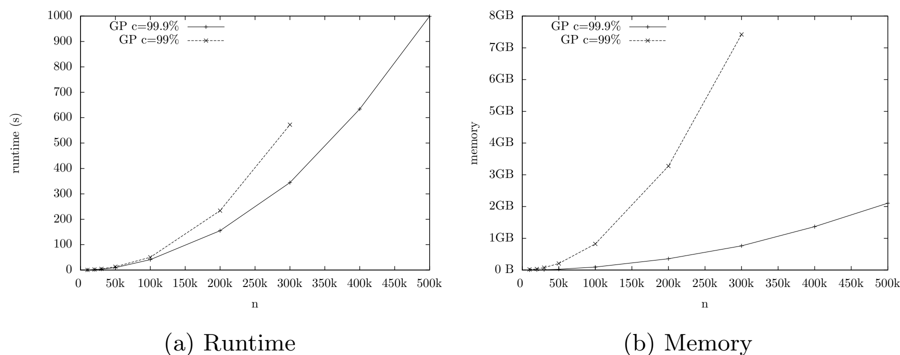


Figure 24: Memory ratio (GP / PTAc) in function of reduction size c, ITA size n=5000



(a) Runtime (b) Memory

Figure 25: Runtime and memory requirements in function of argument relation size n for large scale dataset SYNTH

size of the argument relation, the graph approach consumes less memory than the matrix implementation. As Gordevičius states for reductions up to 10% of the ITA argument relation size the error remains still very low. This is also the range where Split Point Graph outperforms the matrix implementation in terms of memory usage.

7.5 Scalability to Large Datasets

In the following the ability of algorithm GP for reduction of large datasets is analyzed. The experiments does not consider the PTAc algorithm motivated by the memory limitations of the test system. For a sequential relation with size $n = 40000$ and a reduction to $c = 36000$ the size of the $\mathbf{J}$ matrix exceeds 5.3 GB of memory.

The experiments performed to analyze the possibility to reduce large datasets to 99% and 99.9% of its original size. Although the reduction concept is usually used for smaller reductions also these reductions could be useful. An example is the smoothing the ITA result set. Small changes between tuples were eliminated while higher changes remains.

The elimination of a small number of tuples can also be useful in elimination of outliers. The error function considers not only the value difference but also the duration of the merged tuples. If the duration of some outlier tuple is short a merge of long tuples with a similar value could induce a higher error than a merge of a very short outlier tuple where the value differs strongly.

In figure 25a is shown the runtime of the reduction to 99% and 99.9% of its original size of the dataset SYNTH for varying subsets of the argument relation. The reduction to 99% was limited to 300,000 tuples due to memory limitations. In figure 25b is reported the memory consumption for the same experiment. As expected for reduction sizes close to the size of the argument relation both the required space and runtimes decrease slightly.

8 Conclusion

In this work two optimizations of the computation of Parsimonious Temporal Aggregation Queries are introduced. The first optimization decreases the runtime of the computation. This is achieved by reducing the search space of the dynamic programming scheme adopted by PTAc as described by Gordevičious et al. The second improvement regards the memory consumption which in PTAc is quadratic. Experiments showed that the memory requirements with this approach can be reduced to about one third of the space used by the original PTAc algorithm. Effectiveness of both optimizations are mainly depending on the compression rate. Best memory reduction is achieved for the usually adopted reduction size up to 10% and for reduction sizes greater than 80% of the size of the argument relation. Runtime improvements affects all reduction sizes reaching its maximum value for reductions with size is close to the size of the argument relation. Experimental evaluation showed that with the described optimizations PTAc can be adopted also for large scale datasets if only a small reduction is aimed.

Bibliography

[BBJ98] Michael H. Böhlen, Renato Busatto, and Christian S. Jensen. Point- versus interval-based temporal data models. In Susan Darling Urban and Elisa Bertino, editors, *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 192–200. IEEE Computer Society, 1998.

[BGJ06] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Multidimensional aggregation for temporal data. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2006.

[GGB12] Juozas Gordevicius, Johann Gamper, and Michael H. Böhlen. Parsimonious temporal aggregation. *VLDB J.*, 21(3):309–332, 2012.

[JKM+98] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 275–286. Morgan Kaufmann, 1998.

[JSS94] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. Unifying temporal data models via a conceptual model. *Inf. Syst.*, 19(7):513–547, 1994.

[KS95] Nick Kline and Richard T. Snodgrass. Computing temporal aggregates. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 222–231. IEEE Computer Society, 1995.

[MLI03] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. Knowl. Data Eng.*, 15(3):744–759, 2003.

[Wan09] F. Wang. Employee temporal data set, 2009.

[YW03] Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, 2003.