

Parsimonious Temporal Aggregation

Juozas Gordevičius

Johann Gamper

Michael Böhlen

Free University of Bozen-Bolzano, Italy
{gordevicius,gamper,boehlen}@inf.unibz.it

ABSTRACT

Temporal aggregation is a crucial operator in temporal databases and has been studied in various flavors, including instant temporal aggregation (ITA) and span temporal aggregation (STA), each having its strengths and weaknesses. In this paper we define a new temporal aggregation operator, called parsimonious temporal aggregation (PTA), which comprises two main steps: (i) it computes the ITA result over the input relation and (ii) it compresses this intermediate result to a user-specified size c by merging adjacent tuples and keeping the induced total error minimal; the compressed ITA result is returned as the final result. By considering the distribution of the input data and allowing to control the result size, PTA combines the best features of ITA and STA. We provide two evaluation algorithms for PTA queries. First, the oPTA algorithm computes an exact solution, by applying dynamic programming to explore all possibilities to compress the ITA result and selecting the compression with the minimal total error. It runs in $O(n^2pc)$ time and $O(n^2)$ space, where n is the size of the input relation and p is the number of aggregation functions in the query. Second, the more efficient gPTA algorithm computes an approximate solution by greedily merging the most similar ITA result tuples, which, however, does not guarantee a compression with a minimal total error. gPTA intermingles the two steps of PTA and avoids large intermediate results. The compression step of gPTA runs in $O(np \log(c + \delta))$ time and $O(c + \delta)$ space, where δ is a small buffer for “look ahead”. An empirical evaluation shows good results: considerable reductions of the result size introduce only small errors, and gPTA scales to large data sets and is only slightly worse than the exact solution of PTA.

1. INTRODUCTION

Temporal aggregation is a crucial operator in temporal databases that aims to summarize large sets of time-varying information. It has been studied in various flavors, most importantly as instant temporal aggregation (ITA) [2, 7, 8, 12,

13, 16]. The value of ITA at time instant t is computed from the set of tuples that hold at t . Consecutive time points with identical aggregate values are then coalesced into so-called constant intervals, i.e., tuples over maximal time intervals during which the aggregate results are constant. Thus, ITA works at the smallest time granularity and produces a result tuple whenever an argument tuple starts or ends. Its main drawback is that the result size depends on the argument relation. Due to temporally overlapping argument tuples the result relation is often larger than the argument relation, and can get up to twice as large [2]. This behavior is in conflict with the very idea of aggregation, which is to provide a summary of the data. Moreover, many applications do not need the fine-grained result of ITA, but require a concise overview of the data, i.e., a small set of result tuples that represent the most significant changes over time.

Span temporal aggregation (STA) [2, 7] allows to control the result size by permitting an application to specify the time intervals for which to report a result tuple, e.g., for each year from 2000 to 2005. For each of these intervals a result tuple is produced by aggregating over all argument tuples that overlap that interval. STA might not always provide good summaries of the data, since the intervals are specified a priori without considering the distribution of the data.

In this paper we define *parsimonious temporal aggregation* (PTA), which comprises two main steps: (i) it computes the ITA result over the input relation and (ii) it compresses this intermediate result to a user-specified size c by merging adjacent tuples and keeping the induced total error minimal; the compressed ITA result is returned as the final PTA result. By considering the distribution of the input data and permitting control over the result size, PTA combines the best features of ITA and STA. We use the sum squared error over all tuples and aggregate values between the ITA and PTA result to compute the error that is induced by the merging step. Only adjacent tuples, i.e., tuples that belong to the same aggregation group and are not separated by a temporal gap, are considered for merging. This is a constraint that many ITA results naturally satisfy.

Example 1. As a running example, we consider a temporal relation PROJECTS that records the name of an employee (E), the project he/she works for (P), the contracted hours per week (H), the monthly salary (S), and a time period (T) that represents the time interval (in months) during which the project contract is effective. An instance of the PROJECTS relation is given in Figure 1 and graphically illustrated in Figure 2(a), where the timestamps of the tuples are drawn as horizontal lines.

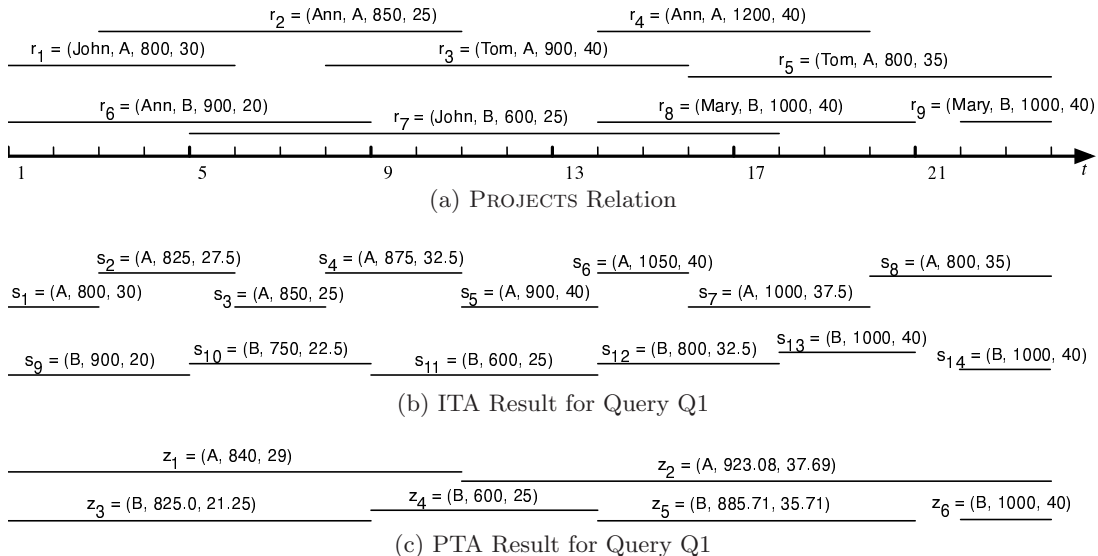


Figure 2: Temporal Aggregation over the PROJECTS Relation.

	<i>E</i>	<i>P</i>	<i>H</i>	<i>S</i>	<i>T</i>
r_1	John	A	800	30	[1, 5]
r_2	Ann	A	850	25	[3, 10]
r_3	Tom	A	900	40	[8, 15]
r_4	Ann	A	1200	40	[14, 19]
r_5	Tom	A	800	35	[16, 23]
r_6	Ann	B	900	20	[1, 8]
r_7	Tom	B	600	25	[5, 17]
r_8	Mary	B	1000	40	[14, 20]
r_9	Mary	B	1000	40	[22, 23]

Figure 1: Temporal Relation PROJECTS.

Consider the following ITA query Q1: “What are the average monthly salary and the average weekly working hours for each project?”. The result of this query is graphically illustrated in Figure 2(b) and contains two aggregation groups for the two projects. It has more tuples than the input relation, but many tuples represent only small fluctuations in the aggregate values, e.g., s_1, \dots, s_4 .

To reduce the size of the ITA result we propose to iteratively merge pairs of adjacent tuples that belong to the same aggregation group, have similar aggregation values, and are not separated by a temporal gap. For example, the tuples s_1 and s_2 are good candidates for merging. In contrast, s_1 and s_9 cannot be merged since they are about different projects, and s_{13} and s_{14} cannot be merged since they are separated by a temporal gap. Figure 2(c) shows a possible compression of the ITA result to six tuples, which is obtained by applying eight consecutive merging steps. For instance, the tuples s_1, \dots, s_4 are merged in three steps to produce the PTA result tuple z_1 , stating that in the interval [1, 10] the average salary and working hours for project A are 840 and 29, respectively.

The contributions of this paper can be summarized as follows:

- We define a new, data-driven temporal aggregation operator, termed PTA, that allows to control the result size by computing ITA as an intermediate result and

compressing it to a user-specified size c such that the induced total error is minimal.

- We provide oPTA, an exact evaluation algorithm for PTA queries, which adopts dynamic programming to investigate all possible compressions of the ITA result to size c . With an input relation of size n and p aggregation functions, the algorithm runs in $O(n^2pc)$ time and $O(n^2)$ space.
- We provide gPTA, an evaluation algorithm that computes an approximate solution to PTA queries by greedily merging the most similar ITA result tuples. By tightly integrating the two PTA steps, the algorithm avoids large intermediate results. It runs in $O(c + \delta)$ space and performs the merging step in $O(np \log(c + \delta))$ time, where δ is a small buffer for “look ahead”.
- An experimental evaluation of PTA on synthetic and real-world data revealed the following results: considerable reductions of the result size introduce only small errors; gPTA is scalable for large data sets, it has an error that is very close to the exact PTA solution, and it is consistently better than the approximate temporal coalescing approach in [1].

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we introduce and formally define the parsimonious temporal aggregation operator. Section 4 presents an exact evaluation algorithm for PTA, and Section 5 presents an algorithm that computes an approximate solution. Section 6 reports the results of an experimental study, and Section 7 concludes the paper and points to future work.

2. RELATED WORK

Various forms of temporal aggregation have been studied in the past, including instant temporal aggregation (ITA), moving-window temporal aggregation (MWTA), and span

temporal aggregation (STA) [2, 7, 8, 12, 13, 16]. They differ mainly in how the time line is partitioned.

ITA [7, 8, 13] works at the smallest time granularity. The time line is partitioned into time instants, and for each time instant, t , the aggregate functions are evaluated over all tuples that hold at t . Then, identical aggregate results for consecutive time instants are coalesced into tuples over maximal time intervals. While ITA reports the most detailed result, the main drawback is that the result relation is typically larger than the argument relation and can be up to twice of its size.

Moving-window temporal aggregation (MwTA) (first introduced in TSQL [9] and later also termed cumulative temporal aggregation [11, 16]), extends ITA aggregation by computing for each time instant t the aggregate functions over all tuples that hold in a window “around” t . Just like ITA it is prone to returning large result relations.

Span temporal aggregation (STA) [11] allows to control the result size by partitioning the time line into predefined intervals. For each such interval, a single result tuple is produced by evaluating the aggregate functions over all argument tuples that overlap that interval. However, the timestamps of the result tuples are specified by the application and are independent of the argument data. Most approaches consider only regular time spans expressed in terms of granularities, e.g., years or months.

In [13] temporal aggregation is formalized in a uniform framework that enables the analysis and comparison of the different forms of temporal aggregation. In a similar vein, the multi-dimensional temporal aggregation operator [2] generalizes previous temporal aggregation operators. Two different semantics are distinguished. Constant interval semantics computes ITA with support for lineage information. With fixed interval semantics, which covers span temporal aggregation (STA), the user specifies the time intervals for which to report result tuples.

All of the frameworks and approaches discussed above address known forms of temporal aggregation (i.e., ITA, MwTA, and STA) and hence suffer from the shortcomings of these operators. A first approach to combine the best features of ITA and STA was presented in [4] and was termed greedy parsimonious temporal aggregation. The operator greedily merges pairs of the most similar, adjacent tuples in the ITA result until an application-specific size constraint is met. The greedy strategy does not guarantee that the compressed result relation is optimal with respect to the induced total error. In this paper we proceed and extend the work in [4] in several directions: we provide a more general definition of parsimonious temporal aggregation which minimizes the total error; we provide a new evaluation algorithm based on dynamic programming for an exact solution of PTA; we provide an algorithm for an approximate solution of PTA, which extends the evaluation algorithm presented in [4] and tightly integrates the computation of the ITA result with the merging step.

The work of Berberich et al. [1] deals with the problem of reducing a temporal relation, where each tuple is assigned a real value and no temporal gaps or aggregation groups exist. The authors propose a technique called approximate temporal coalescing (ATC) which merges adjacent tuples if the local merge error does not exceed a user-specified threshold. The approach scales linearly with the size of the input relation and suits well for online processing of streaming data.

However, the overall error of this approximation tends to be rather high. Similar approaches have been proposed in [10, 14].

Keogh et al. [6] investigate online summarization of large time series data. They observe that only moving-window approaches provide scalable solutions, however, being unable to “look ahead”, the approximations are poor. Offline bottom-up approaches provide significantly better approximations as they consider the whole dataset at once. The authors suggest to combine the two approaches and to process with a bottom-up algorithm each intermediate result of the moving-window algorithm, achieving errors and scalability comparable to bottom-up and moving-window approaches, respectively.

Our algorithm for computing an exact solution of PTA is inspired by the work of Jagadish et al. [5]. They propose a dynamic programming approach to compute a histogram of c buckets from a set of n item frequencies. The algorithm finds the optimal solution in $O(n^2c)$ time. We extend this approach to consider temporal tuples instead of frequencies where different aggregation groups and temporal gaps act as predefined bucket boundaries.

3. PARSIMONIOUS TEMPORAL AGGREGATION

In this section we introduce and provide a formal definition of the *parsimonious temporal aggregation* (PTA) operator, which is comprised of two main steps:

1. Compute ITA over the argument relation
2. Compress the ITA result to a user-specified size c by merging adjacent tuples and keeping the induced total error minimal.

After introducing a few preliminary notations we discuss the two steps in detail.

3.1 Preliminaries

We assume a discrete *time domain*, Δ^T , where the elements are termed *chronons* (or time points), equipped with a total order, $<^T$ (e.g. calendar months with the order $<$). A timestamp (or time interval) T is a convex set over the time domain and is represented by two chronons, $[TS, TE]$, denoting its inclusive starting and ending points, respectively.

A *relation schema* is a three-tuple $R = (\Omega, \Delta, dom)$, where Ω is a non-empty, finite set of attributes, Δ is a finite set of domains, and $dom : \Omega \rightarrow \Delta$ is a function that associates a domain with each attribute. A *temporal relation schema* is a relation schema with at least one timestamp valued attribute, i.e., $\Delta^T \in \Delta$. A *tuple* r over schema R is a finite set that contains for every $A_i \in \Omega$ a pair A_i/v_i such that $v_i \in dom(A_i)$. A *relation* over schema R is a finite set of tuples over R , denoted as \mathbf{r} .

To simplify the notation we assume an ordering of the attributes and represent a temporal relation schema as $R = (A_1, \dots, A_m, T)$ and a corresponding tuple as $r = (v_1, \dots, v_m, T)$. For a tuple r and an attribute A we write $r.A$ to denote the value of the attribute A in r . For a set of attributes A_1, \dots, A_k , $k \leq m$, we define $r[A_1, \dots, A_k] = (r.A_1, \dots, r.A_k)$, and use as a shorthand notation $\mathbf{A} = \{A_1, \dots, A_k\}$ and $r[\mathbf{A}]$.

3.2 Instant Temporal Aggregation

For each combination of grouping attribute values, ITA computes an aggregation result at each time point, t , by considering all argument tuples that hold at t and have the same grouping attribute values.

Definition 1. (Instant Temporal Aggregation) Let \mathbf{r} be a temporal relation with schema $R = (A_1, \dots, A_m, T)$, where $\mathbf{A} = \{A_1, \dots, A_k\}$ is a set of grouping attributes, and $\mathbf{F} = \{f_1/B_1, \dots, f_p/B_p\}$ is a set of aggregate functions. Further, let $\mathbf{r}_{g,t} = \{r \mid r \in \mathbf{r} \wedge r[\mathbf{A}] = g \wedge t \in r.T\}$ be the aggregation group that contains all tuples of \mathbf{r} with identical grouping attribute values and whose timestamp contains t . Then the result of *instant temporal aggregation* is defined as

$$\mathcal{G}^{ITA}[\mathbf{F}][\mathbf{A}][T]\mathbf{r} = \{g \circ f \mid t \in \Delta^T \wedge g \in \pi[\mathbf{A}]\mathbf{r} \wedge \mathbf{r}_{g,t} \neq \emptyset \wedge f = (f_1(\mathbf{r}_{g,t}), \dots, f_p(\mathbf{r}_{g,t}), [t, t])\}$$

and has the schema $S = (A_1, \dots, A_k, B_1, \dots, B_p, T)$.

The variable t ranges over the temporal domain, and g ranges over all combinations of grouping attribute values in \mathbf{r} . For each combination of g and t , the set $\mathbf{r}_{g,t}$ collects the argument tuples that are valid at time t and have the same grouping attribute values as g . A result tuple, x , is then produced by extending g with the result of the aggregate functions f_i that are computed over the non-empty aggregation groups $\mathbf{r}_{g,t}$ and a timestamp that represents the time instant t . Each f_i is some aggregation function (e.g., *sum* or *count*) that takes a temporal relation as argument and applies aggregation to one of the relation's attributes. The resulting value is stored as the value of an attribute named B_i . For instance, the pair $avg(S)/B_1$ computes the average of the salary attribute, which is then stored as a value of attribute B_1 .

To obtain the final result, value-equivalent tuples over consecutive time points are typically *coalesced* into tuples over maximal time periods during which the aggregate values do not change. Though this step is not included in the above definition, we assume coalesced result tuples throughout the rest of the paper.

Example 2. The ITA query Q1 over the PROJECTS relation has a single grouping attribute, $\mathbf{A} = \{P\}$. The aggregate functions are $\mathbf{F} = \{avg(S)/AS, avg(H)/AH\}$. Thus, the schema of the result relation is (P, AS, AH, T) , where AS and AH store the aggregation results. The ITA result relation is graphically illustrated in Figure 2(b).

For the rest of the paper we denote an argument relation and its schema \mathbf{r} and R , respectively. The result of applying ITA aggregation on \mathbf{r} is a relation \mathbf{s} with schema $S = (A_1, \dots, A_k, B_1, \dots, B_p, T)$, where $\mathbf{A} = \{A_1, \dots, A_k\}$ are the grouping attributes and $\mathbf{B} = \{B_1, \dots, B_p\}$ represent aggregate values. Applying PTA aggregation yields the result relation \mathbf{z} with the same schema S .

A fundamental property of ITA aggregation is that the timestamps of the result tuples within a single aggregation group do not intersect. We term such temporal relations *sequential*.

Definition 2. (Sequential Relation) Let \mathbf{s} be a temporal relation with schema $S = (A_1, \dots, A_k, B_1, \dots, B_p, T)$. The relation \mathbf{s} is termed *sequential* with respect to a set of attributes $\mathbf{A} = \{A_1, \dots, A_k\}$ if the following holds true:

$$\forall s_i, s_j \in \mathbf{s} (s_i \neq s_j \wedge s_i[\mathbf{A}] = s_j[\mathbf{A}] \implies s_i.T \cap s_j.T = \emptyset)$$

Obviously, the ITA result in Figure 2(b) is sequential with respect to the grouping attribute P , i.e., the timestamps of all tuples with identical P -values are temporally disjoint.

3.3 Merging Adjacent Tuples

The second step of PTA is to merge adjacent tuples in the ITA result relation, i.e., tuples that belong to the same aggregation group and are not separated by another tuple or by a temporal gap.

Definition 3. (Adjacent Tuples) Let \mathbf{s} with schema $S = (A_1, \dots, A_k, B_1, \dots, B_p, T)$ be the result of ITA aggregation with grouping attributes $\mathbf{A} = \{A_1, \dots, A_k\}$. Two tuples $s_i, s_j \in \mathbf{s}$ are termed *adjacent*, $s_i \prec s_j$, iff the following conditions hold true:

- (1) $s_i[\mathbf{A}] = s_j[\mathbf{A}]$
- (2) $s_i.TE = s_j.TS - 1$

Example 3. Consider the ITA result in Figure 2(b) where $s_1 \prec s_2 \prec \dots \prec s_8$ and $s_9 \prec s_{10} \prec \dots \prec s_{13}$. Examples of non adjacent tuples are $s_1 \not\prec s_3$ and $s_{13} \not\prec s_{14}$ due to the temporal gap between the two tuples and $s_5 \not\prec s_{12}$ due to different values for the grouping attribute P .

We merge pairs of adjacent tuples into a single tuple by means of a merge operator that is defined next.

Definition 4. (Merge Operator) Let \mathbf{s} with schema $S = (A_1, \dots, A_k, B_1, \dots, B_p, T)$ be the result of ITA over some input relation \mathbf{r} , where $\mathbf{A} = \{A_1, \dots, A_k\}$ are the grouping attributes and $\mathbf{B} = \{B_1, \dots, B_p\}$ store the aggregation results. The *merge operator*, \oplus , of two adjacent tuples, $s_i, s_j \in \mathbf{s}$, $s_i \prec s_j$, is defined as

$$s_i \oplus s_j = (s_i.A_1, \dots, s_i.A_k, v_1, \dots, v_p, [s_i.TS, s_j.TE])$$

where $v_l = \frac{|s_i.T|s_i.B_l + |s_j.T|s_j.B_l}{s_j.TE - s_i.TS + 1}$ for $1 \leq l \leq p$.

The merge operator takes two adjacent tuples and produces their common representation, say tuple z . The grouping attributes of z equal to the ones of s_i and s_j . The aggregate values are computed as weighted average of the values in s_i and s_j where the weights are the respective lengths of temporal intervals.

Example 4. Consider the ITA result in Figure 2(b) and assume to merge the two adjacent tuples s_1 and s_2 which have a timestamp of length 2 and 3, respectively. The average salary is then computed as $(2 \cdot 800 + 3 \cdot 825) / (5 - 1 + 1) = 815$. The final result of merging these two tuples is $s_1 \oplus s_2 = (A, 815, 28.5, [1, 5])$.

To reduce the ITA result to a specific size, the merge operator needs to be applied iteratively. However, there is a lower bound for the size of the PTA result, which is determined by the number of adjacent tuples. Each merge of two tuples reduces the size of the relation by 1. For an ITA result relation, \mathbf{s} , the minimal size, c_{min} , of the reduced result relation is therefore given as the difference between the cardinality of \mathbf{s} and the number of adjacent tuples in \mathbf{s} , i.e.

$$c_{min} = |\mathbf{s}| - |\{(s_i, s_j) \mid s_i, s_j \in \mathbf{s} \wedge s_i \prec s_j\}|.$$

In the running example the minimal size of any reduced relation is $c_{min} = 3$, since the ITA contains 14 tuples with 11 tuples being adjacent.

Finally, we define a non-deterministic function that takes as input a sequential relation \mathbf{s} and returns its reduction (or compression) to size c .

Definition 5. (Reduction) Let \mathbf{s} be the result of ITA over an input relation \mathbf{r} , $s_i \prec s_j$ be two adjacent tuples of \mathbf{s} , and c with $c_{min} \leq c \leq |\mathbf{s}|$ be a size constraint. A *reduction*, ρ , of \mathbf{s} to size c is defined as follows:

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

Example 5. There are many ways to reduce the ITA result over the PROJECTS relation depicted in Figure 2(b) to 6 tuples. One solution is depicted in Figure 2(c).

3.4 Error Measure

The merging step introduces an error with respect to the ITA result (except for the trivial case that adjacent tuples are value-equivalent). We use the following measure to quantify this error.

Definition 6. (Error Measure) Let \mathbf{s} be the result of ITA with schema $S = (A_1, \dots, A_k, B_1, \dots, B_p, T)$, where the grouping attributes are $\mathbf{A} = \{A_1, \dots, A_k\}$ and the aggregation attributes are $\mathbf{B} = \{B_1, \dots, B_p\}$. Furthermore, let $\mathbf{z} = \rho(\mathbf{s}, \cdot)$ be a reduction of \mathbf{s} and $w_1 > 0, \dots, w_p > 0$ be a set of positive weights. The *error*, $E(\mathbf{s}, \mathbf{z})$, induced by compressing \mathbf{s} to \mathbf{z} is defined as

$$E(\mathbf{s}, \mathbf{z}) = \sum_{z \in \mathbf{z}} \sum_{\substack{s \in \mathbf{s}, \\ s[\mathbf{A}] = z[\mathbf{A}], \\ s.T \subseteq z.T}} \sum_{i=1}^p w_i^2 |s.T| (s.B_i - z.B_i)^2.$$

The above measure is the well-known sum squared error, which is given as the total sum of the squared difference between the tuples in the ITA result relation, \mathbf{s} , and the tuples in the compressed result, \mathbf{z} . More specifically, for each tuple $z \in \mathbf{z}$ the error measure computes the squared distance over all aggregation results B_1, \dots, B_p between z and the ITA result tuples $s \in \mathbf{s}$ that have been merged to produce z . The weights w_i are necessary to leverage the impact of different attributes.

Example 6. Consider to merge the tuples $s_1 \oplus s_2 = (A, 815, 28.5, [1, 5])$ in the ITA result in Figure 2(b). The resulting relation is $\mathbf{z} = (\mathbf{s} \setminus \{s_1, s_2\}) \cup \{s_1 \oplus s_2\}$. The values of the H attribute in the ITA result range from 22.5 to 40, whereas the values of S attribute are between 600 and 1050, making the latter overly influential. Thus, we assign to the S and H attributes the weights $w_S = 1$ and $w_H = 1050/600 = 26.25$, respectively. Then the error is computed as follows:

$$\begin{aligned} E(\mathbf{s}, \mathbf{z}) &= 1^2 \cdot 2 \cdot (815 - 800)^2 + 26.25^2 \cdot 2 \cdot (28.5 - 30)^2 + \\ &\quad 1^2 \cdot 3 \cdot (815 - 825)^2 + 26.25^2 \cdot 3 \cdot (28.5 - 27.5)^2 \\ &= 5918. \end{aligned}$$

For comparison, the error of merging s_4 with s_5 would be 59077.

Having introduced all building blocks that are required for the PTA operator, we go on to provide its exact definition.

3.5 Defining PTA

Given a result of ITA over an argument relation, our aim is to find its reduction to size c that would induce the minimal error. The following definition introduces the parsimonious temporal aggregation operator as an optimization problem.

Definition 7. (Parsimonious Temporal Aggregation) Let \mathbf{r} be a temporal relation with schema $R = (A_1, \dots, A_m, T)$, $\mathbf{A} = \{A_1, \dots, A_k\}$ be the grouping attributes, and $\mathbf{F} = \{f_1/B_1, \dots, f_p/B_p\}$ be a set of aggregate functions. Furthermore, let $\mathbf{s} = \mathcal{G}^{ITA}[\mathbf{F}][\mathbf{A}][T]\mathbf{r}$ be the result of ITA over \mathbf{r} , c with $c_{min} \leq c \leq |\mathbf{s}|$ be an application-specific size constraint, and $\mathbf{Z} = \{\mathbf{z} | \mathbf{z} = \rho(\mathbf{s}, c)\}$ be the set of all possible reductions of \mathbf{s} to size c . The *parsimonious temporal aggregation* operator is defined as

$$\mathcal{G}^{PTA}[\mathbf{F}][\mathbf{A}][T][c]\mathbf{r} = \arg \min_{\mathbf{z} \in \mathbf{Z}} \{E(\mathbf{s}, \mathbf{z})\}.$$

PTA is a two-step process. First, the ITA over the input relation \mathbf{r} is computed. Then, all possible reductions of the ITA result to size c are considered, choosing the one that yields the smallest total error.

Example 7. Figure 2(c) shows the best possible reduction of ITA relation in Figure 2(b) to $c = 6$ tuples.

4. EXACT PTA EVALUATION

In this section we provide an evaluation algorithm to compute exact solutions for PTA queries, by using dynamic programming to explore all possible compressions of the intermediate ITA result.

4.1 Basic Dynamic Programming Scheme

We assume that the ITA result relation, \mathbf{s} , is sorted first on the aggregation groups and, within each aggregation group, along the time line. This is how many ITA aggregation algorithms return the result relation. We enumerate the sorted ITA result tuples as $\mathbf{s} = \{s_1, \dots, s_n\}$, and we use $\mathbf{s}_i = \{s_1, \dots, s_i\}$, $\mathbf{s}_i \subseteq \mathbf{s}$ to refer to the subset of the first i tuples in \mathbf{s} . A pair of non-adjacent tuples, $s_i \not\prec s_{i+1}$, in \mathbf{s} marks a temporal gap or group boundary and hence is not allowed to be merged. Thus, we set the error of merging a subset $\mathbf{s}' \subseteq \mathbf{s}$, which contains a pair $s_i \not\prec s_{i+1}$, into one tuple to be infinite. This avoids that a subset \mathbf{s}' is selected for merging as long as other subsets which contain only adjacent tuples exist.

The optimal compression of \mathbf{s} to size c , $\rho(\mathbf{s}, c)$, is the one that minimizes the error $E(\mathbf{s}, \rho(\mathbf{s}, c))$. This error equals to the minimal error of compressing the first i tuples, $\mathbf{s}_i = \{s_1, \dots, s_i\}$ to $c - 1$ tuples plus the error of merging the remaining tuples $\mathbf{s} \setminus \mathbf{s}_i = \{s_{i+1}, \dots, s_n\}$ into a single one. The choice of i must minimize the total error. This observation leads to the following dynamic programming scheme:

$$E^*(\mathbf{s}_i, c') = \begin{cases} \min_{1 \leq j < i} \{E^*(\mathbf{s}_j, c' - 1) + E^*(\mathbf{s}_i \setminus \mathbf{s}_j, 1)\} & c' \geq 2 \\ E(\mathbf{s}_i, \rho(\mathbf{s}_i, 1)) & c' = 1 \end{cases}$$

$E^*(\mathbf{s}_i, c') = \min E(\mathbf{s}_i, \rho(\mathbf{s}_i, c'))$ represents the smallest error of compressing \mathbf{s}_i to c' tuples, and $E(\mathbf{s}_i, \rho(\mathbf{s}_i, 1)) = \infty$ if not $s_1 \prec \dots \prec s_i$. When $i < c'$, the error is not defined.

To find $E^*(\mathbf{s}, c)$ we compute $E^*(\mathbf{s}_i, c')$ in increasing order of c' for all $1 \leq c' \leq c$, and for any fixed c' in increasing order of i for all $1 \leq i \leq |\mathbf{s}|$. At each step the error $E^*(\mathbf{s}_j, c' - 1)$ is known from the previous step.

Example 8. The evaluation of the dynamic programming scheme can be illustrated in a matrix, where the number of rows corresponds to the number of tuples in the STA result and the number of columns corresponds to c . The cell (i, c') stores the minimum error of merging the subset \mathbf{s}_i into c' tuples. Table 1 shows the matrix for the running example, where the ITA result has 14 tuples and shall be compressed to 6 tuples. We start filling it by setting $c' = 1$ and computing the first column. To fill the second column, $c' = 2$, we use the data from the first one, and so on. Eventually, the cell $(14, 6)$ contains the error of the optimal compression of the ITA result to six tuples.

Table 1: Dynamic Programming Matrix.

i	$c' = 1$	$c' = 2$	$c' = 3$	$c' = 4$	$c' = 5$	$c' = 6$
1	0	—	—	—	—	—
2	5 918	0	—	—	—	—
3	19 727	5 918	0	—	—	—
4	61 152	19 727	5 918	0	—	—
5	261 867	61 152	19 727	5 918	0	—
6	414 074	88 152	46 727	19 727	5 918	0
7	510 381	101 278	59 852	28 802	14 993	5 918
8	550 937	217 658	101 278	59 852	28 802	14 993
9	∞	550 937	217 658	101 278	59 852	28 802
10	∞	604 550	271 271	154 891	101 278	59 852
11	∞	790 134	456 855	271 271	154 891	101 278
12	∞	1 004 461	671 183	446 293	271 271	154 891
13	∞	1 555 458	925 151	591 872	406 287	271 271
14	∞	∞	1 555 458	925 151	591 872	406 288

There are several ways to improve the performance of the basic algorithm. First, we show that the error of merging a set of tuples into one can be computed very efficiently. Second, we show that many cells of the matrix do not have to be computed to obtain the final result.

4.2 Computing the Error

To efficiently compute the error of merging a set of tuples into a single tuple, we exploit the fact that the error measure introduced in Definition 6 is decomposable. Consider an ordered ITA result \mathbf{s} and a subset of adjacent tuples $\mathbf{s}' = \{s_i, \dots, s_j\}$, $\mathbf{s}' \subseteq \mathbf{s}$, which is merged into a single tuple $z = s_i \oplus \dots \oplus s_j$. The error of this compression, $E(\mathbf{s}', \{z\})$, is equivalent to

$$\sum_{l=1}^p \left[\sum_{\mathbf{s} \in \mathbf{s}'} |s.T| w_l^2 s.B_l^2 - \frac{1}{|z.T|} (w_l \sum_{\mathbf{s} \in \mathbf{s}'} |s.T| s.B_l)^2 \right].$$

To speed up the computation of this expression, we precompute the following arrays of length $|\mathbf{s}|$:

$$\begin{aligned} \mathbf{S}_l[i] &= \sum_{j=1}^i |s_j.T| w_l s_j.B_l \\ \mathbf{SS}_l[i] &= \sum_{j=1}^i |s_j.T| w_l^2 s_j.B_l^2 \\ \mathbf{L}[i] &= \sum_{j=1}^i |s_j.T| \end{aligned}$$

The value at position i in each of these arrays is computed over the subset $\mathbf{s}_i = \{s_1, \dots, s_i\}$. The arrays \mathbf{S} and \mathbf{SS} are needed for each aggregate function in the query, $l = 1, \dots, p$.

Using the precomputed arrays, we can transform the above error formula into

$$E(\mathbf{s}', \{z\}) = \sum_{l=1}^p \left[\mathbf{SS}_l[j] - \mathbf{SS}_l[i-1] - \frac{(\mathbf{S}_l[j] - \mathbf{S}_l[i-1])^2}{\mathbf{L}[j] - \mathbf{L}[i-1]} \right]$$

which can be computed in $O(p)$ time.

4.3 Limiting the Search Space

By limiting the range of the variables i and j in the dynamic programming algorithm, many unnecessary evaluations of $E^*(\mathbf{s}_i, c')$ that would anyway return infinity can be avoided. For that purpose we use an array \mathbf{G} that stores the positions of the non-adjacent tuple pairs. If $s_k \not\prec s_{k+1}$ is the l -th pair of non-adjacent tuples in the sorted relation \mathbf{s} , then $\mathbf{G}[l] = k$. In the running example we have $\mathbf{G} = \{8, 13\}$, representing that $s_8 \not\prec s_9$ and $s_{13} \not\prec s_{14}$. The array \mathbf{G} helps us to find tighter limits for i and j .

We consider first the range of the variable i . Given a fixed c' we compute $E^*(\mathbf{s}_i, c')$ for all values of i between 1 and $|\mathbf{s}|$. For the lower bound of i we have that the error is not defined if $i < c'$. For the upper bound we have that the error is infinite if the number of non-adjacent pairs in \mathbf{s}_i is greater than c' , which is also the case for every $s_{i'}$ with $i' > i$. Thus, when $E^*(\mathbf{s}_i, c')$ evaluates to infinity, we can stop to loop over i and proceed by augmenting c' . The range of i can therefore be limited to $i = c', \dots, \mathbf{G}[c']$. That is, when computing $E^*(\mathbf{s}_i, c')$, for each fixed c' we need to consider only those subsets $\mathbf{s}_i \subseteq \mathbf{s}$ that can be compressed to c' tuples. The smallest such subset is $\mathbf{s}_{c'}$ (i.e., the first c' tuples), and the largest contains all tuples up to the c' -th pair of non-adjacent tuples.

Example 9. Evaluating the dynamic programming matrix in the previous example we get $E^*(s_9, 1) = \infty$ since $s_8 \not\prec s_9$. Thus, the iteration over i can be stopped at this point as we know that $E^*(\mathbf{s}_i, 1)$ will result in infinity for all $i > 8$.

Next, we consider the range of the variable j . To evaluate $E^*(s_i, c')$, we loop over $j = 1, \dots, i-1$ and compute $E^*(s_i \setminus s_j, 1)$. When j is smaller than the index of the rightmost non-adjacent pair of tuples in \mathbf{s}_i , the error evaluates to infinity. Therefore, we use that index as the lower-bound for j . More formally, let g be the index of the latest pair of non-adjacent tuples in \mathbf{s}_i , or 1 if all pairs are adjacent, i.e., $g = \max\{1, \mathbf{G}[k] : \mathbf{G}[k] < i, k = 1, \dots, |\mathbf{G}|\}$. Then the range of j is limited to $j = g, \dots, i-1$. The split of \mathbf{s}_i is done between the index, g , of the latest gap and the end of the set. When $\mathbf{G}[c-1] = g$, we have only one choice to split \mathbf{s}_i , namely at g . The index g can efficiently be determined using binary search over \mathbf{G} .

Example 10. To compute the value of cell $(11, 2)$ in Table 1 the dynamic programming algorithm computed the error $E^*(s_{11} \setminus s_j, 1)$ for all $j = 1, \dots, 10$. However, the result is not infinite only for $j = 8$, since the tuples s_8 and s_9 are non-adjacent and cannot be merged.

In addition, it has been shown in [5] that j should be iterated in decreasing order, i.e., from $i-1$ towards g . The minimum error will be achieved before $E^*(s_i \setminus s_j, 1) > E^*(s_j, c'-1)$.

4.4 The oPTA Algorithm

Figure 3 shows the algorithm oPTA which implements the dynamic programming scheme and the optimizations discussed above. The first step is to run ITA over the input relation, r . The result is stored in \mathbf{s} which is sorted along the grouping attributes and the timestamp start. The array \mathbf{G} is initialized to the positions of the pairs of non-adjacent tuples in \mathbf{s} , which provide boundaries for the merging process. Similar, the arrays $\mathbf{S}, \mathbf{SS}, \mathbf{L}$ for computing the error are initialized. The error $E^*(s_i, c')$ is stored in a hash-map structure, E^* . Another hash-map, J , is used to store the value of the variable j which yields the minimal error for a given i and c' . In other words, the value $J(|\mathbf{s}|, c)$ indicates where the relation \mathbf{s} has to be split to obtain the right-most PTA result tuple. Lines 4 to 16 compute the optimal compression of the relation \mathbf{s} to size c , using the dynamic programming scheme and the optimizations discussed above. Finally, lines 17–23 produce the result relation \mathbf{z} by merging subsets of tuples according to the splitting points stored in J .

```

1 Algorithm:  $oPTA(r, \mathbf{A}, \mathbf{F}, c)$ 
2  $\mathbf{s} \leftarrow \mathcal{G}^{ITA}[\mathbf{F}][\mathbf{A}][T]r;$ 
3 Initialize  $\mathbf{G}, \mathbf{S}, \mathbf{SS}, \mathbf{L}$  as well as  $E^*$  and  $J$ ;
4 for  $c' = 1, \dots, c$  do
5   for  $i = c', \dots, \min\{\mathbf{G}[c'], |\mathbf{s}|\}$  do
6      $g = \max\{1, \mathbf{G}[k] : \mathbf{G}[k] < i, k = 1, \dots, |\mathbf{G}|\};$ 
7      $E^*(s_i, c') \leftarrow \infty;$ 
8     if  $\mathbf{G}[c-1] = g$  then
9        $E^*(s_i, c') \leftarrow E^*(s_g, c' - 1) + E^*(s_i \setminus s_g, 1);$ 
10       $J(i, c) \leftarrow g;$ 
11     else
12       for  $j = i - 1, \dots, g$  do
13          $E^*(s_i, c') \leftarrow \min\{E^*(s_i, c'),$ 
14            $E^*(s_j, c - 1) + E^*(s_i \setminus s_j, 1)\};$ 
15         if  $E^*(s_i \setminus s_j, 1) > E^*(s_j, c' - 1)$  then
16           break;
17        $J(i, c) \leftarrow j;$ 
18  $\mathbf{z} \leftarrow \emptyset; i \leftarrow |\mathbf{s}|;$ 
19 while  $c > 0$  do
20    $j \leftarrow J(i, c);$ 
21    $\mathbf{z} \leftarrow \mathbf{z} \cup \{s_{j+1} \oplus \dots \oplus s_n\};$ 
22    $i \leftarrow j;$ 
23    $c \leftarrow c - 1;$ 
24 return  $\mathbf{z};$ 

```

Figure 3: Algorithm oPTA.

Example 11. Consider the evaluation of oPTA over the PROJECTS relation. First, the ITA result is computed and enumerated as $\mathbf{s} = \{s_1, \dots, s_{14}\}$. Next, $E^*(s_i, 1)$ is computed for $i = 1, \dots, 8$. Similarly, we compute $E^*(s_i, 2)$ for $i = 2, \dots, 13$. When i is between 2 and 8, the value of g is 1 and j ranges between 1 and i . However, when i is between 9 and 13, the value of j equals $g = 8$. In a similar fashion we compute the remaining errors until $c' = 6$. The final result is shown in Figure 2(c).

The worst case complexity of the merging step in the oPTA algorithm is $O(n^2pc)$, since in each of c iterations

the error is computed n^2 times. p is the number of aggregate functions in the query and is usually very small.

5. APPROXIMATE PTA EVALUATION

In this section we introduce a greedy evaluation algorithm (gPTA) that provides an approximate solution to PTA. Intuitively, gPTA tries to reduce an ITA result relation to size c by choosing at each merging step the most similar pair of adjacent tuples. This might result in a compression which does not necessarily correspond to the compression with the minimal total error as used in PTA. However, the experiments show that gPTA provides results which are very close to the results of oPTA.

5.1 Computing the Error

The greedy evaluation algorithm reduces the ITA result, \mathbf{s} , by choosing at each merging step the pair of the most similar tuples. Hence, the reduction function, ρ , is deterministic, and the tuples $s_i, s_j \in \mathbf{s}$ that are chosen for merging are determined as follows:

$$(s_i, s_j) = \arg \min_{\substack{s_i, s_j \in \mathbf{s} \\ s_i < s_j}} E(\{s_i, s_j\}, \{s_i \oplus s_j\})$$

Example 12. In our running example the result of gPTA coincides with the result of oPTA, which is shown in Figure 2(c). As it will be seen in the empirical evaluation, this is often the case. Figure 4 shows a dendrogram which illustrates the first six merging steps. The most similar tuples s_1 and s_2 are merged first, followed by s_6 and s_7 , and so on.

The restriction of the error measure to the two tuples being merged yields the same result as if we would compute on each iteration the error between all the tuples of the relation and its compression. Intuitively, at any step of the gPTA reduction process we deal with an intermediate relation \mathbf{s}' , whose most similar adjacent tuples, $s_i < s_j$, are chosen for merging. Using the merged tuple we will reduce the relation again to obtain a new intermediate result \mathbf{s}'' and proceed recursively. Therefore, the tuples s_i, s_j are the most similar ones when the error $E(\mathbf{s}, \mathbf{s}'')$ is minimized. With the following proposition we show that due to the properties of the error function in Definition 6 only s_i and s_j have to be considered to calculate the error $E(\mathbf{s}, \mathbf{s}'')$ that merging them would induce.

PROPOSITION 1. *Let \mathbf{s} be an ITA result relation, $\mathbf{s}' = \hat{\rho}(\mathbf{s}, c+1)$ be its compression to size $c+1$ and $\mathbf{z} = \hat{\rho}(\mathbf{s}, c)$ is compression to size c , where in the last iteration the tuples $s_i, s_j \in \mathbf{s}'$ are merged to $z \in \mathbf{z}$. Then the following holds:*

- (1) $E(\mathbf{s}', \mathbf{z}) = E(\{s_i, s_j\}, \{z\})$
- (2) $E(\mathbf{s}, \mathbf{z}) = E(\mathbf{s}, \mathbf{s}') + E(\mathbf{s}', \mathbf{z})$

PROOF. To prove the first statement it is enough to observe that the sets $\mathbf{s} \setminus \{s_i, s_j\}$ and $\mathbf{z} \setminus \{z\}$ are identical, hence only s_i and s_j contribute to the total error.

For the second statement, recall that the error is calculated as a the total sum over all tuples in \mathbf{s} . Therefore, $E(\mathbf{s}, \mathbf{z})$ can be expanded as $E(\mathbf{s}, \mathbf{s}') - E(\mathbf{s}, \{s_i, s_j\}) + E(\mathbf{s}, \{z\})$. Inserting the latter into (2) and simplifying we get

$$E(\mathbf{s}, \{z\}) - E(\mathbf{s}, \{s_i, s_j\}) = E(\mathbf{s}', \mathbf{z}).$$

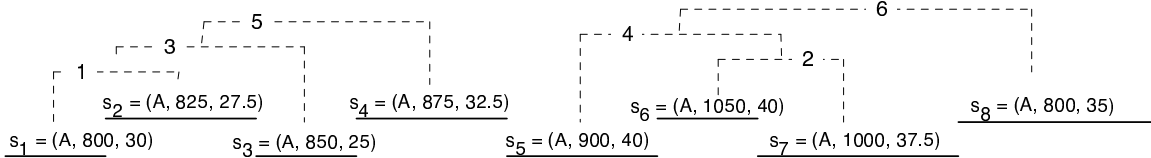


Figure 4: Dendrogram of Merging Steps.

Furthermore, let $\mathbf{s}_i^* \subset \mathbf{s}$ be the tuples in \mathbf{s} that make up \mathbf{s}_i , i.e., for all $s \in \mathbf{s}_i^*$, $s[\mathbf{A}] = s_i[\mathbf{A}] \wedge s.T \subseteq s_i.T$ holds. Let \mathbf{s}_j^* be defined for s_j in a similar fashion. Then for s_i , and similarly for s_j , we have

$$E(\mathbf{s}_i^*, \{z\}) - E(\mathbf{s}_i^*, \{s_i\}) = E(\{s_i\}, \{z\}).$$

Expanding the errors, simplifying and recalling that s_i is obtained by merging tuples in \mathbf{s}_i^* , i.e. $\sum_{s \in \mathbf{s}_i^*} \frac{|s.T|}{|s_i.T|} s.B = s_i.B$, we see that the statement holds. \square

In the following we will first describe a baseline version of the greedy evaluation algorithm for PTA, followed by a more efficient greedy algorithm which additionally integrates the computation of ITA and the merging step. We will show that both approaches provide equivalent results.

5.2 A Baseline Algorithm

The baseline version of the greedy evaluation algorithm, gPTABASIC, implements the greedy merging strategy by employing a heap to efficiently determine the pair of most similar tuples in each merging step. The heap supports the standard operations PUSH, POP, and HEAPIFY in $O(\log n)$ time [3]. Each tuple in the ITA result relation is represented by a heap node N , which is comprised of the four fields. $N.tuple$ refers to the tuple itself, $N.prev$ and $N.next$ point to the two adjacent neighbors of $N.tuple$. The key value of N is the error of merging the node's tuple with the tuple of the preceding neighbor, i.e., $N.key = E(\mathbf{s}, \{N.tuple, N.prev.tuple\})$. If $N.tuple$ and $N.prev.tuple$ are non-adjacent, the key is set to infinity.

The procedure INSERT constructs a new node for each tuple, sets the links to the two neighbors, and pushes the node into the heap. The procedure MERGE takes the top node off the heap and merges its tuple into the preceding neighbor. Then it recomputes the key values and ensures that the heap property is maintained.

Figure 5 shows the baseline version of the gPTA algorithm. After running ITA, the obtained result tuples are inserted into the heap. Then the algorithm iteratively merges pairs of the most similar tuples until the size constraint c is satisfied.

Example 13. Figure 6(a) depicts the heap nodes after inserting the first six tuples of the ITA result in our running example. The boxed node represents the top of the heap, thus s_2 and s_1 are the most similar tuples. The dashed lines show the predecessor-successor relationships between the tuples/nodes. Figure 6(b) shows the heap after merging the top element, i.e., the tuples s_1 and s_2 . The old node is shaded gray, whereas the updated nodes are marked in bold.

The ITA result over an input relation with n tuples can be at most of size $2n - 1$. Thus, the merging step of the baseline

```

1 Algorithm: gPTABASIC( $r, R, \mathbf{A}, \mathbf{F}, c$ )
2  $H \leftarrow$  new empty heap;
3  $H.last \leftarrow$  NULL;
4  $\mathbf{s} \leftarrow \mathcal{G}^{ITA}[\mathbf{F}][\mathbf{A}][T]r$ ;
5 for each tuple  $s$  in  $\mathbf{s}$  ordered by  $\mathbf{A}$  and  $T$  do
6   INSERT( $H, s$ );
7 while Heap size  $> c$  do
8   MERGE( $H$ );
9 return  $H$ ;

```

Figure 5: Algorithm gPTABASIC.

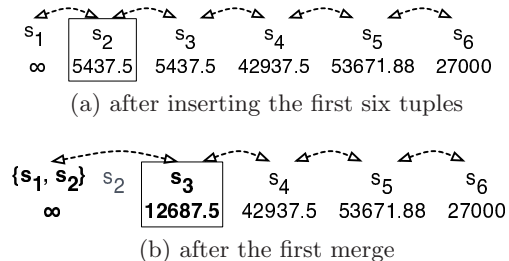


Figure 6: Heap of the gPTABASIC Algorithm.

algorithm, gPTABASIC, takes in the worst case $O(np \log n)$ time, where p is the number of aggregation functions in the query. The main drawback of this algorithm is the huge memory consumption, since the entire ITA result is kept in memory.

In the next section we tackle the problem of the huge memory consumption and provide a space efficient evaluation algorithm which tightly integrates the calculation of ITA and the merging step.

5.3 The gPTA Algorithm

Temporal aggregation is likely to be used over huge argument relations storing historical data. The PTA operator offers a way to obtain a compressed aggregation result of c tuples. However, the baseline version of the greedy algorithm, gPTABASIC, and the exact algorithm, oPTA, rely on the entire intermediate ITA result being stored in memory, which can be up to twice as large as the input relation.

Here we propose an improvement to space and time complexity of the baseline gPTA algorithm that does not penalize its quality. The main idea of the new approach is to tightly integrate the computation of ITA and the merging step. When ITA result tuples arrive (in sorted order along the aggregation groups and time line), they are added immediately to the heap. Whenever the heap size exceeds c , we try to merge the pair of most similar tuples to reduce it to

size c . However, a merge might not be possible immediately, but only after reading some more ITA result tuples. More specifically, the merge is not possible if the node at the top of the heap represents the most recently arrived ITA tuple. This tuple cannot be deemed most similar with its preceding neighbor when the succeeding neighbor is not yet known; thus we have to wait for the next tuple. The size of the heap structure is limited to $c + \delta$, where δ is a small buffer space for “look ahead”. The reduction of the heap size also leads to a better performance. In the worst, yet unlikely, case the algorithm reads the whole ITA result relation before any merging can take place, i.e. $c + \delta = |\mathbf{s}|$.

Figure 7 shows the integrated gPTA algorithm. We adapted the ITA algorithm in [2] to produce the ITA result tuples one by one, which does not change its performance and space requirements.

```

1 Algorithm: gPTA( $\mathbf{r}, \mathbf{A}, \mathbf{F}, c$ )
2  $H \leftarrow$  new empty heap ;
3 Initialize ITA operator with  $\mathbf{F}, \mathbf{A}$ , and  $\mathbf{r}$ ;
4  $s \leftarrow$  new ITA result tuple;
5 while  $s \neq \text{NULL}$  do
6   INSERT( $H, s$ );
7   while  $\text{size}(H) > c \wedge \text{top}(H) \neq s$  do
8     MERGE( $H$ );
9    $s \leftarrow$  new ITA result tuple;
10 while  $\text{size}(H) > c$  do
11   MERGE( $H$ );
12 return  $H$ ;

```

Figure 7: Algorithm gPTA.

Example 14. We run gPTA over the PROJECTS relation with $c = 5$. Figure 8 depicts the contents of the heap each time a new ITA result tuple is processed. When s_6 is read and inserted into the heap, the heap size exceeds $c = 5$, and the algorithm executes one merge step. Inserting s_7 into the heap puts the pair s_6, s_7 as the most similar pair of tuples on top of the heap. Even though the heap size exceeds 5, merging cannot take place until the next insertion. When s_8 comes to the heap, two merging steps are necessary to reduce it to 5. The algorithm proceeds in this way until the entire ITA relation is read.

	s_1	s_1
	s_2	s_2

	$\overline{s_1}, \overline{s_2}, s_3, s_4, s_5$	s_6
	$s_1 \oplus s_2, s_3, s_4, s_5, \overline{s_6}$	$\overline{s_7}$
	$\overline{s_1 \oplus s_2}, \overline{s_3}, s_4, s_5, \overline{s_6}, \overline{s_7}$	s_8
	$s_1 \oplus s_2 \oplus s_3, s_4, \overline{s_5}, \overline{s_6 \oplus s_7}, s_8$	s_9

Figure 8: gPTA Heap while Inserting s_1, \dots, s_9 .

The complexity of the gPTA algorithm depends on the ITA algorithm. Assume that the latter takes T time and S space. Then, gPTA requires $O(T + np \log(c + \delta))$ time and $O(S + c + \delta)$ space.

5.4 Correctness of the gPTA Algorithm

We prove that gPTABASIC and gPTA provide equivalent results. We begin by introducing the concept of a core pair.

Definition 8. (Core pair) A pair of adjacent tuples, $s_i \prec s_j$, in a sequential relation, \mathbf{s} , is a *core pair* iff the following two conditions hold:

- (1) $E(\{s_i, s_j\}, \{s_i \oplus s_j\}) < E(\{s_{i-1}, s_i\}, \{s_{i-1} \oplus s_i\})$
- (2) $E(\{s_i, s_j\}, \{s_i \oplus s_j\}) \leq E(\{s_j, s_{j+1}\}, \{s_j \oplus s_{j+1}\})$

Intuitively, if a set of tuples is merged to a single tuple, the core pair of that set will be the one that is merged first, that is, which is the most similar one. Clearly, each PTA result tuple, z , that is produced by merging a subset of ITA tuples has a core pair. Core pairs have the following properties:

1. there is at least one core pair in an ITA result relation;
2. all core pairs are disjoint;
3. the most similar of the core pairs is always at the top of the heap in the gPTA algorithm.

Example 15. The core pairs are easy to spot in the dendrogram of gPTA merging steps in Figure 4. The core pairs are the tuples s_1, s_2 and s_6, s_7 . The gPTA merges first the most similar of the core pairs, namely s_1, s_2 .

The following lemma shows that the final result does not change if not the most similar core pair would have been merged first.

LEMMA 1. *Let \mathbf{s} be a sequential relation of size n that has to be reduced to c tuples using $\hat{\rho}(\mathbf{s}, c)$. Further, let a pair of tuples $s_i \prec s_j \in \mathbf{s}$ be one of the c most similar core pairs. Then, $\hat{\rho}(\mathbf{s}, c)$ is equivalent to*

$$\hat{\rho}((\mathbf{s} \setminus \{s_i, s_j\}) \cup \{s_i \oplus s_j\}, c).$$

PROOF. The merging order changes only if the tuple $s_i \oplus s_j$ becomes more similar to s_{i-1} or s_{j+1} than s_i or s_j had been. However, that is not possible due to monotonicity of the error measure. \square

THEOREM 1. *The gPTA algorithm produces exactly the same result as the gPTABASIC algorithm.*

PROOF. The gPTABASIC operates on a subset, $\mathbf{s}_{c+\delta} \subset \mathbf{s}$ of the ITA result relation, i.e., $\mathbf{s}_{c+\delta} = \{s_1, s_2, \dots, s_{c+\delta}\}$, where $\delta \geq 1$. At least one pair of tuples in this subset has to be merged to reduce \mathbf{s} to c tuples. If the most similar pair of tuples in $\mathbf{s}_{c+\delta}$ is also a core pair, it can be merged immediately according to the Lemma 1. However, it is not possible to check if the pair $s_{c+\delta-1}, s_{c+\delta}$ is a core pair without knowing $s_{c+\delta+1}$. That is why the last pair is never merged by the algorithm. \square

6. EXPERIMENTAL EVALUATION

In this section we present the results of experimental evaluations of the two PTA evaluation algorithms, oPTA and gPTA, and the approximate temporal coalescing (ATC) algorithm presented in [1].

6.1 Data

We used two different temporal relations. First, the real world “Incumbents” data from the University of Arizona, which store employee salary records. With each record

a project ID, department ID, salary, and time interval in months are associated. The relation has 83 857 records in total. The second relation, called employee temporal dataset (ETDS), has been generated by F. Wang [15] and depicts the evolution of employees in a company. There are 2 875 697 records in total. Each record stores employee number, sex, department, title, salary, and the valid time interval of the record in months.

Every ITA aggregation with different grouping attributes and aggregation functions produces a new dataset where the attribute value distribution, the number of aggregation groups, and the temporal gaps differ. Table 2 summarizes the features of the ITA results with various groupings. With each grouping we used various combinations of aggregation functions, eventually obtaining 19 datasets from the Incumbents and ETDS relations.

Table 2: Different Groupings of Datasets.

Source	Grouping	ITA tuples	c_{min}
Incumbents	None	2 658	0
Incumbents	Dep.	5 552	14
Incumbents	Project	11 643	40
Incumbents	Proj., Dep.	16 144	131
ETDS	None	6 394	0
ETDS	Title	38 338	7
ETDS	Dep.	57 424	0
ETDS	Sex	12 787	2
ETDS	Sex, Title	76 245	14
ETDS	Dep., Title	188 138	0
ETDS	Sex, Dep., Title	344 188	0
ETDS	Emp. No	5 450 737	30024

6.2 Quality Experiments

For each dataset we compute the error of reducing the data to every possible c , using the PTA and ATC algorithms. Reducing a relation to $c = c_{min}$ results in the maximum possible error, which is the same for PTA and ATC. We use the maximum error as a normalization factor corresponding to 100% of the error.

The ATC approach takes as input the upper limit ϵ of the local error that the algorithm is allowed to make when merging the tuples. We map ϵ to c by simulating a set of ϵ values in the range between 0 and 1. For each such value the ATC result and the total error are computed. Whenever two different errors for two ATC result relations of the same size are obtained, the smaller one prevails.

Figure 9 shows the error induced by running oPTA, gPTA, and ATC. The experiments revealed that a significant reduction of the ITA result is possible without introducing large errors. In Figure 9(a) the Incumbents data is grouped by Project and the AVG function. The reduction of 11 643 tuples to 150 (2%) yielded approximately a 10% error. The results are even better for the same data grouped by Department in (b). In (c) the ITA aggregation over the ETDS dataset was reduced from 344 188 tuples to 250 before the error reached 10%.

The graphs in Figure 10 confirm the same observation and summarize the gPTA result over all datasets derived from Incumbents and ETDS data, using AVG as an aggregation function. More specifically, the Incumbents datasets in Figure 10(a) can be reduced to less than 2% before the error exceeds 10%. The ETDS datasets in Figure 10(b) do not exceed the 10% threshold, even when compressed to 0.1% of

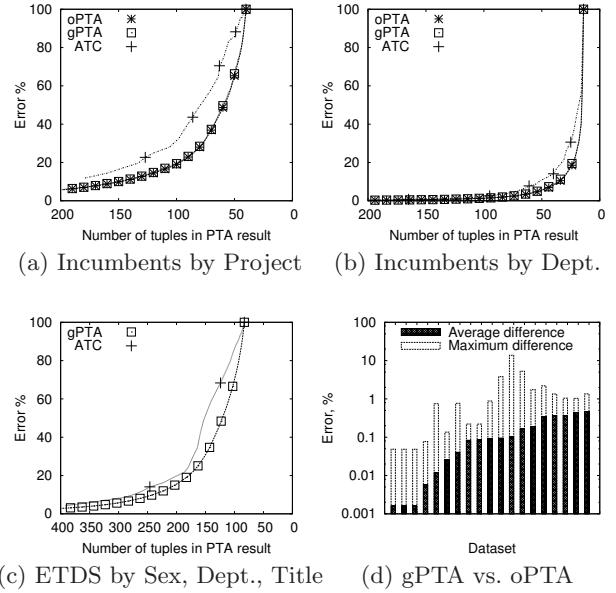


Figure 9: Comparison of Errors introduced by oPTA, gPTA and ATC.

the ITA result relation. Figure 10(c) depicts the two largest datasets showing that a reduction to 15% is still below 10% of the error.

In addition, we observe that the gPTA algorithm approximates the optimal solution very well. Among the datasets originating from Incumbents data, the gPTA induced on average an error 0.1% bigger than the exact PTA algorithm. Figure 9(d) illustrates this observation in more detail. For each dataset we computed the average and maximum difference between the optimal and the gPTA errors over all values of c . Note, that the vertical axis scales logarithmically and the biggest difference between the two algorithms is 13% of the maximal error.

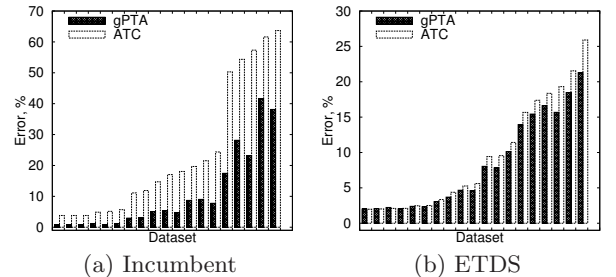


Figure 11: Average Error of gPTA and ATC per Dataset.

Finally we note that the average error of the gPTA algorithm is consistently smaller than that of ATC. Figure 11 depicts the average error of the two algorithms per dataset, confirming the observation. The figure also shows that for some datasets the average errors of ATC, gPTA, and PTA are very close. We have observed, that such situations happen for datasets that can be compressed extremely well, e.g., Figure 9(b). When the error of exact solution grows rapidly with decreasing c , the error of ATC deviates away as for

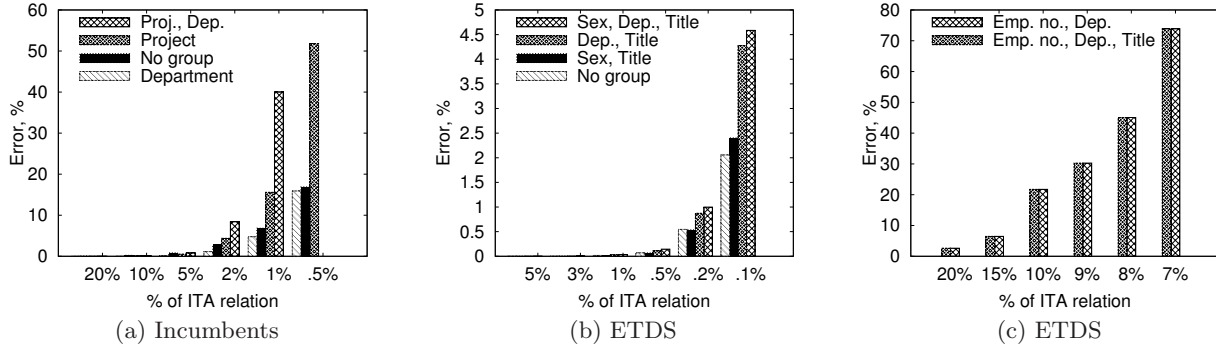


Figure 10: gPTA Achieved Good Compression Introducing Only Small Errors.

example in Figure 9(a) and (c).

6.3 Performance Evaluation

All the algorithms have been implemented using the Java programming language. The machine running the experiments has 16GB of random access memory and 4 AMD Opteron processors running at 2600MHz each.

In the following performance graphs the time taken to produce ITA results is excluded as it is out of the scope of this paper. We also exclude the time needed to fetch the data and write the final result back to the database.

6.3.1 Performance of oPTA

We compare three different implementations of the exact PTA algorithm. First, the plain dynamic programming (DP) approach. Second DP that uses binary search (DP+BS) to limit value range of variable j as described in [5]. Third, the *oPTA* algorithm that implements the optimizations developed in this paper.

Two datasets are used. The “No Gaps” dataset has no aggregation groups nor temporal gaps and was obtained by running ITA on the Incumbents data. On the contrary, the “Gaps” dataset has multiple aggregation groups and gaps and was obtained by running ITA on the Incumbents data grouped by project and department.

We measure the time to compute the PTA result over subsets of varying sizes. Figure 12(a) shows the execution time of the algorithms on the “No Gaps” data with $c = 80$. The DP algorithm takes a lot more time than DP+BS and *oPTA*. As expected, the latter two perform very similarly, since the heuristics of *oPTA* are not useful for data with no gaps or aggregation groups. With c increased by a factor of two in Figure 12(b), the algorithms take twice as much time. That leads to the conclusion that the approaches scale linearly with respect to c .

Next, we evaluate the performance of the algorithms on subsets of the “Gaps” dataset. For the best results the value of c_{min} should remain constant for each subset. We have observed that by choosing subsets between 700 and 2000 tuples the value of c_{min} varied between 75 and 85. Such a variation is not significant to the overall outcome.

As Figure 12(c) depicts, all of the algorithms perform faster with the second dataset, as the presence of gaps reduces the amount of computations. In addition, the *oPTA* algorithm significantly outperforms the other two.

To see how the change of c influences the performance, we

fixed the size of the “Gaps” subset to 1200 tuples ($c_{min} = 85$) and ran the experiment by varying c from 0 to 400. The result is plotted in Figure 12(d). All the algorithms take constant time to complete for all $c \leq c_{min}$. From that point on, DP scales linearly as expected and the timings of the other two approaches remain almost constant.

To conclude, the *oPTA* algorithm introduced in this paper performs well for data with and without temporal gaps or aggregation groups. Nevertheless, when the input dataset is larger than a few thousands of tuples and c is big, the required time is unacceptable.

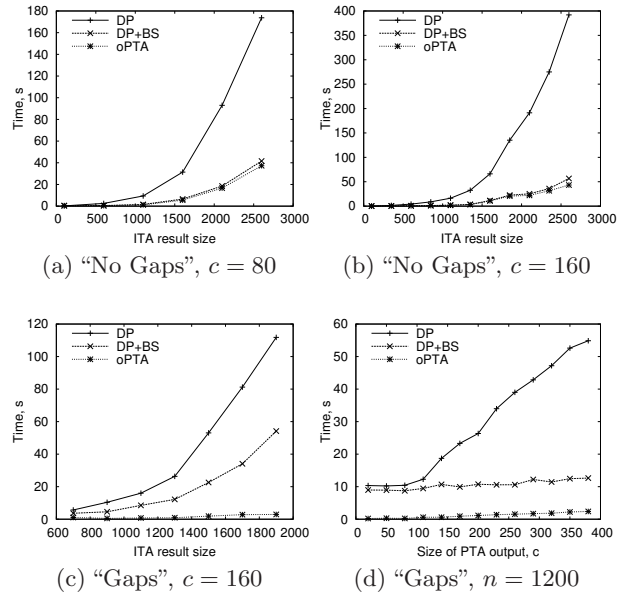


Figure 12: Performance of Exact Algorithms.

6.3.2 Performance of gPTA

We used three different groupings of the ETDS dataset to compare the running times of gPTA and ATC. All groupings yielded relations with more than 5 million tuples. Setting $c = 1$ and $\epsilon = \infty$ we ran the algorithms on various subsets of the data. Figure 13(a) depicts the average running time of each algorithm with respect to the size of the input. We conclude that the ATC algorithm outperforms gPTA only by a small margin.

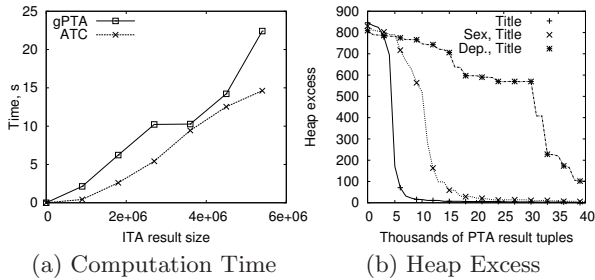


Figure 13: Performance of gPTA Algorithm.

In addition, we evaluate the memory requirements of the *gPTA* algorithm. Recall, that the algorithm operates with a heap of size $c + \delta$ where δ varies throughout the aggregation, depending on the distribution of the data and the value of c . Intuitively, the worst case happens when c is small and the tuples in the dataset are very similar. In such situations no core pair can be found within c tuples and long look-aheads are necessary.

Indeed we observed the highest values of δ when aggregating datasets that can be reduced significantly without big error, i.e., the ones of Figure 10(b). Figure 13(b) depicts the relation between c and δ using these datasets. The horizontal axis ranges over the values of c and the vertical over δ . Even the highest values of δ are very small compared to the datasets in the figure, which vary from 38 to 76 thousands of tuples. In addition, δ tends to decrease with higher values of c .

7. CONCLUSIONS

In this paper we introduced PTA, which is a new temporal aggregation operator that combines the best features of ITA and STA. The operator (i) computes the ITA result over the input relation and (ii) compresses this intermediate result to a user-specified size c , by merging adjacent tuples and keeping the total error minimal.

We presented two evaluation algorithms for PTA queries. First, the oPTA algorithm computes the exact PTA result. It is based on dynamic programming and explores all possibilities to reduce the intermediate ITA result to size c . We developed a number of improvements to further reduce the search space. The algorithm runs in $O(n^2)$ space and $O(n^2pc)$ time, where n is the size of the input relation and p is the number of aggregate functions used. Second, the gPTA algorithm computes an approximate solution of PTA by greedily merging the pairs of most similar tuples. Moreover, it tightly integrates the computation of ITA result and the merging step. While not guaranteeing an optimal solution, the algorithm significantly reduces the time and space consumption. It runs in $O(c + \delta)$ space and $O(np \log(c + \delta))$ time, where δ is a small buffer for "look ahead".

An experimental evaluation of the algorithms revealed the following results: considerable reductions of the result size introduce only small errors; gPTA is scalable for large data sets; the error of gPTA is very close to the optimal compression offered by oPTA, and it is consistently better than the approximate temporal coalescing approach described in [1].

Future work will include the following two aspects. First, the merging process shall be extended to allow merges across

different aggregation groups as well as bridging small temporal gaps between tuples. Second, a careful investigation of different similarity measures is worthwhile. Especially, when merging across aggregation groups we have to compare categorical and numerical attribute values.

8. REFERENCES

- [1] K. Berberich, S. J. Bedathur, T. Neumann, and G. Weikum. A time machine for text search. In *Proc. of SIGIR*, pages 519–526, 2007.
- [2] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *Proc. of EDBT*, volume 3896 of *LNCS*, pages 257–275. Springer, 2006.
- [3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, Inc., 1990.
- [4] J. Gordevicius, J. Gamper, and M. Böhlen. A greedy approach towards parsimonious temporal aggregation. In *Proc. of the 15th International Symposium on Temporal Representation and Reasoning (TIME-08)*, pages 88–92, June 2008.
- [5] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. of VLDB'98*, pages 275–286, 1998.
- [6] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *Proc. of ICDM'01*, pages 289–296, 2001.
- [7] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proc. of ICDE'95*, pages 222–231, Taipei, Taiwan, March 1995.
- [8] B. Moon, I. F. Vega Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):pp. 744–759, May/June 2003.
- [9] S. B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1-3):147–175, 1989.
- [10] Y. Qu, C. Wang, and S. Wang. Supporting fast search in time series for movement patterns in multiple scales. In *Proc. of the 7th International Conference on Information and Knowledge Management*, 1998.
- [11] R. T. Snodgrass, S. Gomez, and L. E. McKenzie. Aggregates in the temporal query language TQuel. *IEEE Transaction on Knowledge and Data Engineering*, 5(5):826–842, 1993.
- [12] P. Tuma. Implementing historical aggregates in TempIS. Master's thesis, Wayne State University, Detroit, Michigan, 1992.
- [13] I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):pp. 271–286, 2005.
- [14] C. Wang and S. Wang. Supporting content-based searches in time series via approximation. In *Proc. of the 12th International Conference on Scientific and Statistical Database Management*, 2000.
- [15] F. Wang. Employee temporal data set. <http://timecenter.cs.aau.dk/>.
- [16] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal*, 2003.