Department of Informatics, University of Zürich

BSc Thesis

Lineage Aggregation on Temporal-Probabilistic Relations

Mirko Richter

Matrikelnummer: 12-917-175

Email: mirko.richter@uzh.ch

July, 2017

supervised by Prof. Dr. Michael Böhlen and Katerina Papaioannou

University of Zurich[UZH]

Department of Informatics

Acknowledgements

I need to thank Katerina Papaioannou, my supervisor, for supporting, guiding and for her helpful advise. I also want to thank Prof. Michael Böhlen for the opportunity to conduct my bachelor thesis with the Database Technology Group.

Abstract

In this thesis, we investigate the challenges of lineage aggregation in temporal-probabilistic relations. In contrast to existing types of aggregation that have been specially treated, e.g. cummulative or selective, data lineage for a single result interval does not correspond to a single numerical value but can become as large as the number of input tuples. This leads to runtime and space efficiency problems due to the high number of tuples that might be valid over an interval. In order to overcome these overheads, we exploit the semantics of lineage aggregation and introduce the *Lineage-Update* algorithm. The *Lineage-Update* algorithm is a sweeping algorithm that uses a data structure tailored for the needs of lineage. By exploiting the semantics of lineage aggregation, our *Lineage-Update* algorithm improves in both time and space complexity to existing approaches since it (a) updates data lineage instead of recomputing it for every result tuple and (b) avoids restoring subexpressions that occur in the lineages of consecutive tuples. We perform an extensive experimental analysis, using both synthetic and real datasets, that shows that *Lineage-Update* outperforms established aggregation algorithms when lineage-aggregation is performed.

Zusammenfassung

In dieser Bachelorarbeit untersuchen wir die Herausforderungen von der Aggregation des Lineage Attributes mit zeitlich-probabilistische Relationen. Im Gegensatz zu den existierenden Aggregations-Typen, wie cumulative oder selective Aggregation, kann die aggregierte Lineage die gleiche Anzahl Elemente wie Tupels in der Input-Relation haben. Das Resultat der existierenden Aggregations-Typen ist immer eine einzelne Zahl. Das führt zu Effizienzproblemen in der Laufzeit und dem Speicher aufgrund der grossen Anzahl an Tupels, die zur gleichen Zeit aktiv sein können. Um diese Probleme zu lösen, nutzen wir die Semantik der Aggregation der Lineage und stellen den Lineage-Update Algorithmus vor. Der Lineage-Update Algorithmus ist ein sweeping basierender Algorithmus, der eine Daten Struktur verwendet, die speziell für die Aggregation der Lineage entwickelt wurde. Weil wir die Semantik der Aggregation der Lineage kennen und ausnutzen, hat unser Lineage-Update Algorithmus eine verbesserte Laufzeit und Speicher Verwendung, da er (a) die aggregierte Lineage aktualisiert anstatt sie ganz neu zu berechnen und (b) Teil-Ausdrücke der Resultate, die sich aufeinanderfolgenden Resultate teilen, nicht erneut speichert. Wir führen eine ausführliche empirische Analyse mit synthetischen und echten Datensets durch, die zeigt, dass der Lineage-Update Algorithmus besser als die existierenden Algorithmen funktioniert, wenn die Aggregation der Lineage ausgeführt wird.

Contents

List of Figures

1 Introduction

Combining any number of non-temporal attributes with an interval attribute gives temporal data. The tuple with its non-temporal attributes is then considered valid within its interval. We refer to a relation as a temporal relation, if it consists of temporal data. Temporal aggregation is describing the use of an aggregation operation on temporal data. Lineage is an attribute that identifies a tuple uniquely. We use the term data lineage to specify the lineage aggregation result, which is a concatenation of lineages that contribute to a particular result. An area data lineage is especially useful are probabilistic relations. Tuples in a probabilistic relations have an attribute called probability with a value between 0 and 1. This value represents the probability that the data in this tuple is true or correct. Probabilistic data is also called uncertain data.

We discuss temporal-probabilistic relations which have the lineage attribute and we expect the database system to aggregate the data lineage for the result tuples in all queries.

Temporal aggregations queries have especially high computation costs. They expect an aggregation result for every time point at which an input tuple is valid. Established temporal aggregation approaches use almost uniformly the same result semantics to reduce computational cost and the size of the result. They exploit the fact that the temporal aggregation result remains constant for a certain interval. The temporal aggregation result for a specific time point is computed from the tuples that are valid at this time point, thus the tuples in the data lineage. Consequently, if two time points have the same data lineage, then they also have the same aggregation result. Hence, instead of aggregating the result for every time point, they concentrate on start and end points of the input tuples, ie. at that time points the data lineage changes. This splits the result tuples into intervals for which the data lineage remains constant. Splitting the result intervals on start and end points of the input tuples simultaneously maximizes the length of intervals with constant data lineage. Maximizing the length of result intervals minimizes the number of result tuples needed and therefore reduces the amount of aggregations needed to compute the result as well as the size of the result.

The lineage aggregation result is generally a concatenation of the lineages that contribute to that result tuple. The aggregation functions to concatenate the lineages include the logic *AND* and logic *OR* functions. Which function to use is determined by the type of input relation and type of query. The data lineage of an aggregation with uncertain data differs based on the variant that is used. Murthy et al. [5] introduces three variants to determine a distinct aggregation result in probabilistic databases. Namely, low bound, high bound and the expected value. For sake of simplicity and without loss of generality, we use the variant high bound throughout the thesis for aggregations in probabilistic databases. Therefore we concatenate the lineages with the logic *AND* operator. This simplification allows us to disregard the probability attribute. In this thesis, we focus on the challenges of lineage aggregation. Computing probabilities based on data lineages, is not part of this thesis.

R

tid	$Name$	$Team$	T	λ	p
r_1	Xabi Alonso	Liverpool	[2002, 2005)	r_1	0.5
r_2	Niall Quinn	Sunderland	[1998, 2006)	r_2	0.8
r_3	Julio Arca	Sunderland	[2000, 2006)	r_3	0.9
r_4	Peter Reid	Sunderland	[1998, 2003)	r_4	0.5
r_5	David Bellion	Liverpool	[2005, 2007)	r_5	0.9

(i) Temporal-Probabilistic relation R with Lineage attribute.

Team	Interval	Count	Lineage
Sunderland	[1998,2000)	2	$r_2 \wedge r_4$
Sunderland	[2000,2003)	3	$r_2 \wedge r_3 \wedge r_4$
Sunderland	[2003,2006)	2	$r_2 \wedge r_3$
Liverpool	[2002,2005)	1	r_1
Liverpool	[2005,2007)	1	r_5

(ii) Result Table of Temporal and Lineage Aggregation.

Figure 1.1: Input Relation **R** and Result Table.

We use the following example to introduce the challenges and solutions of lineage aggregation over lineage-enhanced probabilistic-temporal relations.

Example. *Consider the following query **How many players played for each team at every point in time** over the input relation in figure 1.1i. This query requires counting the valid players for each team at each point in time. Figure 1.1ii shows the result table for such a query. Since we want to know the number of players within a team, we differentiate the result tuples by team. Then, within a team, the result tuples are split by intervals to cover all years from the first start point to the last end point of the input tuples. These intervals are split by the start and end points of the input tuples. Considering figure 1.2, the vertical lines show the time points at which the data lineage changes. So either a new tuple becomes valid or a currently valid tuple stops being valid. These lines split the result intervals into maximum length intervals for constant data lineages. Since we use the high bound variant, the lineage aggregation result is concatenated by the AND operator. In the first result tuple, we have a count of two players for team Sunderland and the interval [1998,2000) and the data lineage reports that the tuple* r_2 ***and*** r_4 *contributed to the result.*

*For team Liverpool the aggregation result is **1** for the entire time [2002,2007). Despite the constant result, we break the interval [2002,2007) at 2005 since the data lineage changes at this time point. Therefore, we get two result intervals for team Liverpool: for [2002,2005) the count is 1 and the valid tuple is* r_1 *and for [2005,2007) the count is also 1 but the valid tuple is* r_5.

This result semantics are well known and used in algorithms for temporal aggregation. Dividing the result into intervals with constant data lineages makes it a fitting approach when

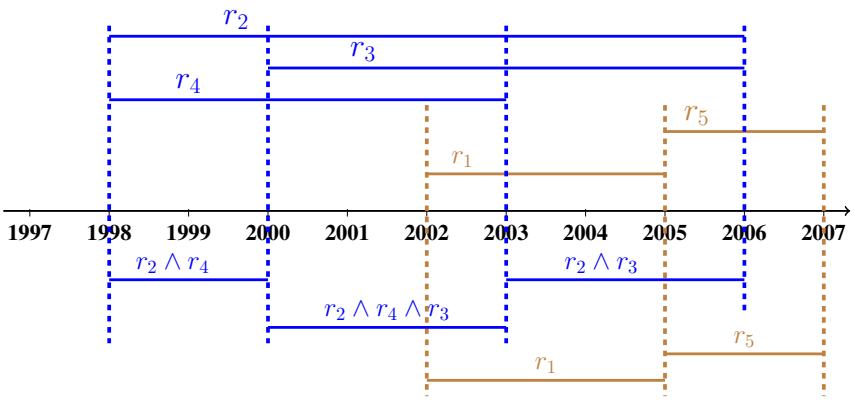


Figure 1.2: Input and Result in graph form.

used over lineage-enhanced temporal relations. Nevertheless, no established temporal aggregation algorithm has implemented data lineage computation in their result.

The second result tuple from the result table in figure 1.1ii has an aggregation result *count* of 3 and therefore 3 lineages in the data lineage. This shows that the size of the data lineage is determined by the number of overlapping tuples for that interval and only bounded by the number of input tuples. This number could be as high as the number of input tuples given they all belong to the same aggregation group and overlap in a time point. Therefore the main challenge that proves lineage aggregation on temporal data as non-trivial, is efficiency. The result semantics from the temporal aggregation require aggregating and storing the data lineage once for every result tuple. We can divide the challenge into computational and space efficiency. On one hand, we want to be able to compute the data lineage efficiently. But we are also interested in reducing the required space for the results of the lineage aggregation.

A second observation from the result table in figure 1.1ii is that the data lineage for team Sunderland only ever changes in one lineage in adjacent result tuples. From the first to the second tuple, the lineage r_3 gets added and from second to third result tuple, the lineage r_4 is removed. Hence, instead of aggregating the data lineage from scratch for every result tuple, we update it. Similarly, we want to be able to reuse the instances of lineages from the previous result tuple, so that we can prevent storing the same lineages as often as they contribute to a result.

The trivial approach of fetching all input tuples to compute the data lineage for every result tuple can be improved significantly by using a data structure to maintain and update the data lineages. Simultaneously, the required space for the result can be reduced by reusing the lineage instances from previous results.

Established algorithms that solve the issues of temporal aggregation have not tried to enhance the algorithm to handle lineage-enhanced relations. Trying to extend the algorithms with lineage aggregation generally leads to one of these two problems. Firstly, extending the algorithm to do lineage aggregation can be straight forward, since the established algorithm

structure allows to add lineage computation easily. But since its temporal aggregation concept strongly favors a separate lineage computation for each result interval, the algorithm will lack efficiency. Secondly, the established approach might already exploit the idea of keeping and updating intermediate results in temporal aggregation. But the approach does not provide a suitable data structure to maintain, update and store the data lineage efficiently.

Our contributions are as follows:

- We define lineage aggregation as a new type of aggregation and declare its properties.
- We suggest a data structure that specifically exploits the properties of lineage aggregation and therefore reduces the computational cost and required result space significantly.
- We introduce the Lineage-Update algorithm that uses the proposed data structure to compute lineage aggregation over lineage-enhanced temporal relations while exploiting the properties of the new lineage aggregation type.
- We report empirical results with synthetic and real world data that confirm our intuition and show our approach performing lineage aggregation over temporal data efficiently.

In Section 2, we discuss related work followed by preliminaries in Section 3. In Section 4, we introduce lineage aggregation and determine its properties. Section 5 presents our proposed data structure with the add and remove operations and we introduce our algorithm that can be used in lineage-enhanced temporal environment and computes temporal aggregation and data lineages. Section 6 presents the results of our empirical experiments. Section 7 contains the conclusion.

2 Related Work

All of the related work we discuss performs temporal aggregation by determining the result intervals from all start and end points of the input tuples and computing the aggregation result once for these intervals. Neither of the established algorithms we discuss work unmodified on lineage-enhanced temporal relations.

Böhlen et al. [1] introduced the TMDA-CI algorithm using the proposed *group table* structure that stores current information to be able to compute the result intervals and the aggregation result for multiple groups while scanning the input relation. The algorithm processes input tuples and directly computes the result for intervals up to the start point of the input tuple. Simultaneously, it uses end point trees. There is one end point tree for every aggregation group. This tree stores the data of the input tuples sorted by their end points to be able to produce the remaining result tuples after all input tuples are scanned. At every point in the algorithm the end point tree represents all valid tuples within an aggregation group. The algorithm aggregates by traversing the end point tree for every new result tuple. This makes it easy to extend the algorithm with lineage computation. The lineage can be stored alongside other attributes in the end point tree and the data lineage can also be computed by traversing the end point tree. But this method recomputes and restores the lineage from scratch for every result tuple, hence it is not sufficiently efficient.

A different approach is to prepare the temporal data in a way that non-temporal aggregation operators can be used. Dignös et al.[3] adjust the initial intervals so that the subintervals produced are either equal or disjoint. Thus, they reduce temporal aggregation to non-temporal aggregation. They introduce two interval adjustment functions, called normalize and alignment. In both methods they use a left outer join to produce the subintervals. The left outer join is computed by a nested-loop-join or a merge-join and is difficult to improve. They already require $\mathcal{O}(n * m)$ of comparisons, with n being the number of input tuples and m the number of distinct start and end points, to adjust the intervals.

The idea of the Timeline Index proposed by Kaufmann et al. [4] introduced an idea to perform temporal joins and aggregations by using a reduced representation of the relation, called the Timeline Index. The Timeline Index consists of two tables which combined, contain all relevant information to recall for every moment the valid data lineages, thus supports temporal aggregation. They differentiate between cumulative aggregation, e.g. sum,avg,count, and selective aggregation, e.g. min,max. The temporal aggregation algorithms for both, cumulative and selective aggregation, use the concept that the result can be updated from the previous interval, instead of recomputing from scratch every time. We will later see that lineage aggregation has properties that are different from cumulative or selective aggregation. Therefore the

methods used for cumulative and selective aggregation by the Timeline Index cannot handle lineage computation. Nevertheless, the Timeline Index allows to compute the result intervals and access the valid tuples for these intervals. By using this custom aggregation method mentioned by Kaufmann et al. [4], the algorithm falls back to recomputing and restoring the data lineage for every new result tuple.

Cafagna and Böhlen [2] propose a solution for deteriorating sort-merge algorithms in temporal environments. Sort-merge based algorithms deteriorate with temporal data since an interval with start and end point does not have a total order. In their approach, first, they introduce an algorithm to partition the input relation in such a way that within a single partition no two tuples are overlapping in their intervals. They prove that performing a temporal join, temporal anti-join or full outer join on these partitions instead of the entire relation eliminates backtracking and thus reduces and caps unproductive comparisons. Cafagna and Böhlen [2] introduce an algorithm, DIPMerge, that performs these joins. They show that the result intervals for a temporal aggregation can be computed by performing full outer joins of the DIP partitions of the input relation. Then the temporal aggregation function needs to be adapted to work on columns instead of rows to compute the aggregation results. Joining the partitions generates a table containing all information that is necessary to compute the aggregation. Then the aggregation function still needs to fetch all result tuples to compute the aggregation result. Adding the lineage as an attribute allows lineage computation but does require $i * j$ more elements in the result table, i being the number of result tuples and j the number of partitions. A DBMS has usually a maximum number of columns per table. PostgreSQL, for example, has 250-1600 as a maximum number of columns depending on the column types. Since $i * j > 1600$ is possible and likely for big relations where a lot of tuples are overlapping, the DIP approach is not implementable in PostgreSQL. Similar holds for other DBMS. Also the empirical results in section 6 show that this approach is heavily dependent on the number of partitions it produces, ie. the maximum number of intervals overlapping in a single time point in the input relation. One additional partition leads to one additional run of the DIPMerge algorithm. This means an additional intermediate result table is produced. The next DIPMerge always uses this intermediate result table as the outer relation of the next DIPMerge. With a lot of partitions, the intermediate results grow and not only we have to run more DIPMerge algorithms but also the DIPMerge algorithm becomes slower.

3 Preliminaries

3.1 Definition

We denote a **temporal-probabilistic schema** by $R^{Tp}(\mathbf{F}, \lambda, T, p)$, where $\mathbf{F} = (A_1, A_2, \ldots, A_m)$ is an ordered set of attributes, and each attribute A_i is assigned to a fixed domain Ω_i (for each $i \in \{1, \ldots, m\}$). T is a dedicated temporal attribute with domain $\Omega^T \times \Omega^T$, where Ω^T is a finite and ordered set of time points. Conversely, p is a dedicated probabilistic attribute with domain $\Omega^p = (0, 1] \subset IR$. A **temporal-probabilistic relation r** over R^{Tp} then is a finite set of tuples. Each tuple $r \in \mathbf{r}$ is an ordered set of values in the appropriate domains. The value of attribute A_i of r, is denoted by $r.A_i$ (and analogously for the other attributes).

Moreover, we consider a **lineage expression** λ to be a Boolean formula, consisting of tuple identifiers and the three Boolean connectives $\neg$ (“not"), $\wedge$ (“and") and $\vee$ (“or"). Tuple identifiers implicitly represent Boolean random variables, among which we assume independence. For a base tuple r, $r.\lambda$ is an atomic expression consisting of just r itself. For a result tuple $\tilde{r}$, which is derived from one or more tuples, $\tilde{r}.\lambda$ is a Boolean expression as defined above. For this thesis we will use the high bound variant from Murthy et al. [5] to determine an exact data lineage for the result. Therefore, we generally use the Boolean connection $\wedge$ (“and”) to concatenate the base lineages of the tuples contributing to the result.

3.2 Existing Types of Aggregation

In this subsection, we want to capture the properties of certain types of aggregation functions so that an algorithm can exploit these properties to become more efficient. Kaufmann et al. [4] introduced two types of aggregation, namely cumulative and selective aggregation. The algorithm that uses the Timeline Index to perform aggregation exploits the properties of both these types. This enables the algorithm to update their aggregation results efficiently instead of recomputing them for every new result tuple. Based on the semantics of the result in temporal aggregation, we are especially interested in three properties of types of aggregations: a) how to compute the aggregation result, b) how to update the aggregation result whenever a new tuple becomes valid and c) how to update it when a tuple stops being valid.

3.2.1 Cumulative Aggregation

Cumulative Aggregations covers functions like SUM, COUNT or AVG. The aggregation result is computed by adding up some value from all tuples that are relevant to the result. For example the sum adds up all values from the tuples while count adds up 1 for every tuple.

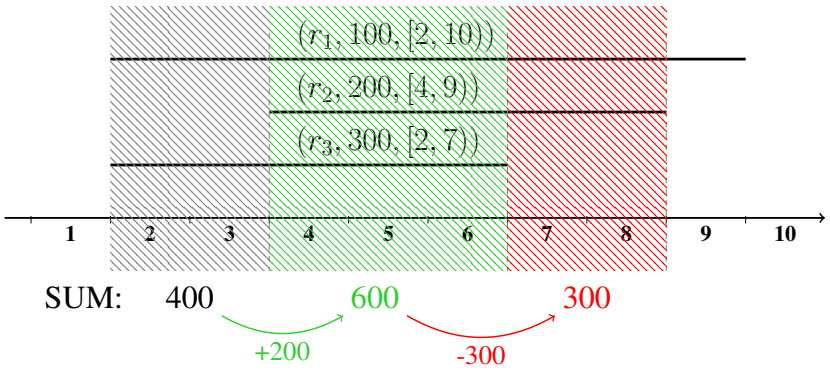


Figure 3.1: Cumulative Aggregation (SUM).

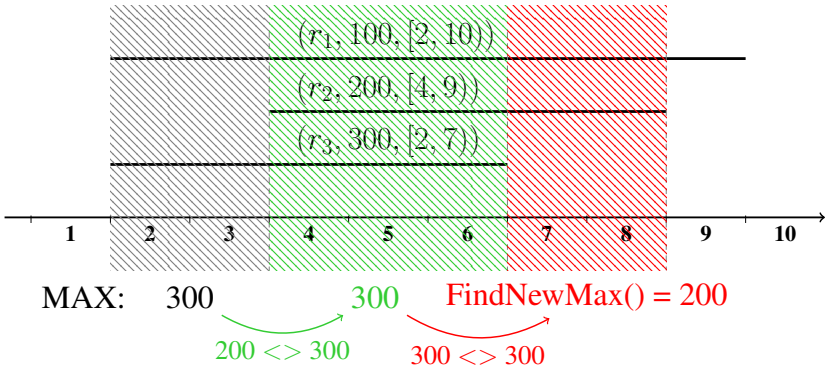


Figure 3.2: Selective Aggregation (MAX).

Consider figure 3.1, in the first interval [2,4) two tuples (r_1, r_3) are valid and the sum of their values is $100 + 300 = 400$. At 4 tuple r_2 becomes valid and we can update the result by adding the value of r_2 to the previous result and we get 600. Similar for time point 7, when r_3 stops being valid. Instead of computing the sum from scratch we can update the result by subtracting the value of r_3, which is 300, from the previous result. We get the current aggregation result of 300.

All aggregation functions which belong to the cumulative aggregation are capable to update their results in that fashion. The temporal aggregation as presented in the work of Kaufmann et al.[4] is the only established work that uses this property and introduced an algorithm that simply stores a current cumulative aggregation result and updates it on every start or end point.

3.2.2 Selective Aggregation

Selective Aggregations covers functions like MIN or MAX. The aggregation result is a value of a single tuple that has been selected on its characteristic. For example the aggregation function MAX computes the result by selecting the maximum value of all valid values.

Consider figure 3.2 to see how the selective aggregation result gets updated. We have the

aggregation function MAX and in the first interval [2,4) the maximum is 300. At 4, when r_2 becomes valid, we compare the new value with the current result. Since the new value 200 is smaller than our current result, we keep the maximum at 300. If the new value were bigger, we would update the result with the new value. Similarly, when a value which is smaller than the current result stops being valid, we can keep the result the same. But in figure 3.2 at time point 7, r_3 with value 300 stops being valid. Since this is the current maximum, we are forced to find the new maximum from the remaining valid values. In this case, the new maximum at time point 7 is 200 from tuple r_2.

To efficiently handle the last case, a sorted Top-K data structure has been proposed[4]. Whenever the current result stops being valid, the new top value in the Top-K data structure is the new result. The data structure is called Top-K, since only the top k values are stored kept sorted. This is more efficient than keeping all valid tuples sorted. The new worst case is if simultaneously all top k values stop being valid. There are no sorted elements and the new result must be found from the unsorted list holding the remaining values.

4 Lineage Aggregation

In this section, we introduce the lineage aggregation and explain its semantics as well as its differences to existing aggregation types. We show how the result of lineage aggregation can be represented, efficiently updated and efficiently stored.

4.1 Aggregation Properties

The difference of lineage aggregation in comparison to cumulative or selective aggregation is that instead of aggregating integer values, we aggregate the lineage attribute. The function we use to aggregate the lineages is the logic and '$\wedge$' operator. Hence, the lineage aggregation result is not a single integer value, as in case of cumulative and selective aggregation, but a concatenation of the lineages of the valid input tuples. Maintaining a current representation of this result, requires access to the lineages of all valid tuples at the current time. This form of the result makes it impossible to affiliate the lineage aggregation in the selective aggregation. In lineage aggregation, we are not interested in a selection of a single lineage but in the concatenation of all of them.

Reaching a start point of a tuple demands to add the lineage of this tuple to the current representation of the lineage aggregation result. This add operation depends on the structure that is used to store the current result. In the next section, it becomes clear why we use a list to store the current result. Consequently, the add operation can be implemented as pre- or appending the new lineage to the existing list of lineages. In figure **??** at time point 4, a tuple starts being valid, hence we need to add its lineage to the current result. The current result is $r_1 \wedge r_3$ and we add r_2 to get $r_1 \wedge r_3 \wedge r_2$.

When a tuple stops being valid, the lineage of this tuple essentially needs to be removed from the current lineage aggregation result. This remove operation needs to find and then remove the lineage of the tuple that stops being valid in the current representation. The requirements of the remove operation prevent affiliating lineage aggregation in the cumulative aggregation. In cumulative aggregation, it is not needed and impossible to find a specific value within the current result. Considering figure **??** at time point 7, the tuple r_3 stops being valid. We need to find and remove the lineage r_3 from the current result. After removing r_3, we get $r_1 \wedge r_2$ for the succeeding result interval.

4.2 Result Space

In this section we investigate the space needed to store the data lineages in the result tuples and how elements of the data lineages can be reused to reduce the required space. Look back

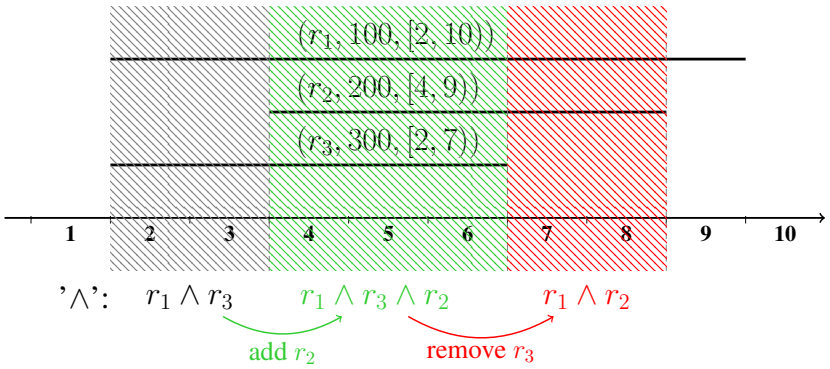


Figure 4.1: Update Lineage Aggregation.

Team	Interval	Count	Lineage
Sunderland	$[1998, 2000)$	2	p_1

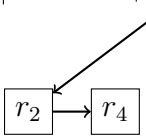


Figure 4.2: Lineage Aggregation Result (Pointer).

at the results in figure **??**. The only difference between the first and the second lineage result is the added lineage r_2. In other words, $r_1 \wedge r_3$ appear in both results. In fact, r_1 appears in all three results. In this subsection, we use those common partial results to reduce the space required to store the result of lineage aggregation.

We use a linked list to store the lineages of the currently valid tuples. Whenever we create a result tuple, we store a pointer to a node in this list to represent the data lineage. In fact, the data lineage for this result tuple is the concatenation of all elements in the list from the node the result points to and onwards. Figure 4.2 shows a result tuple and the current lineage list. We store in the result tuple in the lineage column the pointer p_1. p_1 points to the list node with lineage r_2. Concatenating the elements in the list from r_2 an onwards with the '$\wedge$'-concatenation function gives us $r_2 \wedge r_4$. This is the correct lineage aggregation result for the interval [1998,2000) and team Sunderland from our introduction example. At this point, we have created a first result tuple and a lineage list of two elements. The goal is to reuse this lineage list for the succeeding results.

Before we show how we reuse the list, we need to highlight that the elements in the list in figure 4.2 must not be changed. Removing or altering one of these elements or appending a new element changes the result of p_1. Therefore, to preserve the correct results already created, we cannot alter the list in any way from the node a result points to and onwards. Also, the lineage aggregation for a result tuple is the concatenation of *all* valid lineages for that interval.

Team	Interval	Count	Lineage
Sunderland	$[1998, 2000)$	2	p_1
Sunderland	$[2000, 2003)$	3	p_2

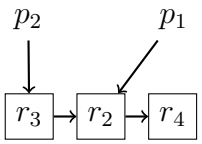


Figure 4.3: Reusing previous Lineage List.

And the lineage list keeps all of these valid lineages. Thus whenever we produce a result tuple, the pointer to the lineage list must point to the front element of the list to ensure that **all** valid lineages are included in the result.

However, we are able to prepend new lineages to the existing lineage list without changing the already produced results. We established that the lineage list must not be altered *after* a node a result points to, but we can prepend a new element and preserve the produced result. In figure 4.3, we produced the second result tuple. The lineage of the second result tuple is $r_3 \wedge r_2 \wedge r_4$, so the only difference to the lineage of the first result tuple is the added r_3. By prepending r_3 to the existing lineage list and creating the pointer p_2, which points from the second result tuple to the new front element in the list, we successfully preserved the previous result and reused the previous result to produce the current result. Figure 4.3 highlights that following p_1 still provides the result $r_2 \wedge r_4$ and p_2 provides the new result $r_3 \wedge r_2 \wedge r_4$ while reusing the elements r_2 and r_3 from the previous result.

Updating the lineage also includes the remove operation. When a tuple stops being valid, we need to remove its lineage from the list. We already determined and in figures 4.2 and 4.3 we can observe, that the pointer from the latest result always points to the front element of the lineage list. Otherwise the result would not include *all* currently valid lineages. This makes it unattainable to update the list by removing an element and preserving the previous result. Hence, when we perform a remove operation to update the lineage list we must first create a copy of the current list before we can remove an element. Figure 4.4 highlights the third result tuple that has been produced after tuple r_4 stopped being valid. p_3 from this result tuple points to a new list created by copying the existing list and removing r_4.

Team	Interval	Count	Lineage
Sunderland	$[1998, 2000)$	2	p_1
Sunderland	$[2000, 2003)$	3	p_2
Sunderland	$[2003, 2006)$	2	p_3

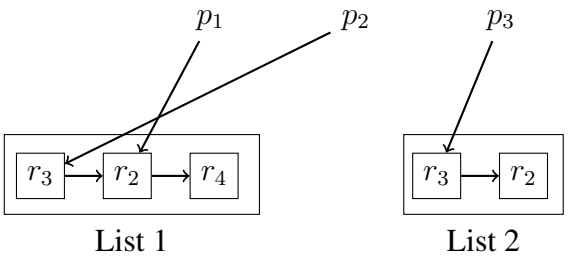


Figure 4.4: Update Lineage List in Remove Operation.

5 Algorithm

In this section we introduce our Lineage-Update algorithm alongside its lineage specifically designed data structure. The algorithm performs lineage aggregation on temporal relations and exploits the semantics of Lineage Aggregation from section 4.

The idea of the algorithm is to scan the input tuples sequentially by start point within their aggregation group. By first grouping the input tuples, the algorithm can process the input tuples without the need of finding the corresponding group during the computation. Then the lineage aggregation result and the result intervals are computed on the fly. Using the semantics from section 4, the algorithm is capable of updating the data lineage from result to result instead of recomputing it. To do that it needs to process and store the information of the tuples in a way, so that it can keeps the current data lineage result, can update it and knows the next start or end point to split the result interval. First we propose a data structure that enables the algorithm to store and to update the information of the input tuples as needed.

5.1 Data Structure

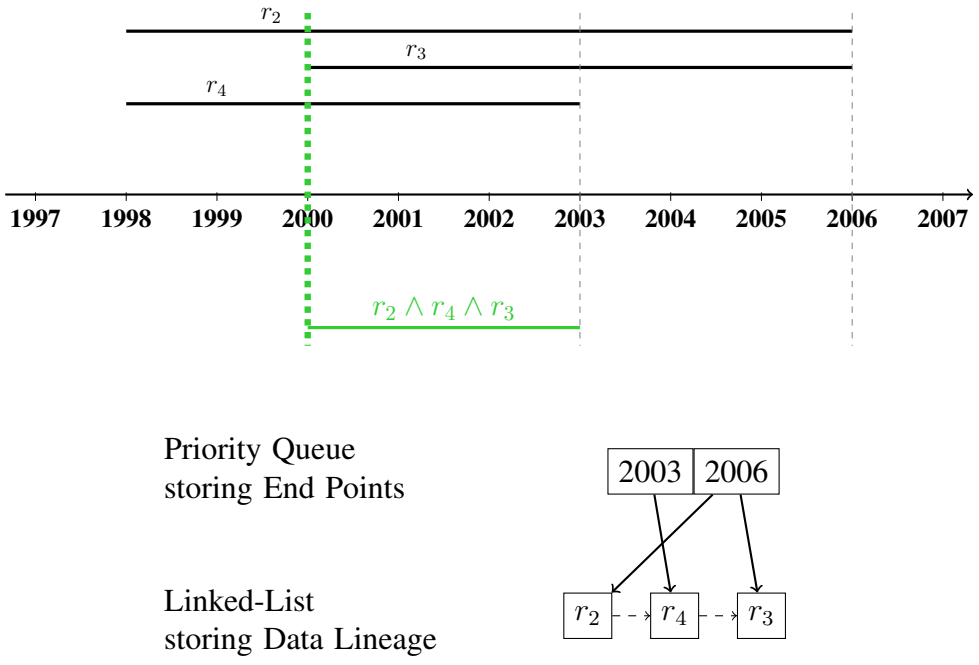


Figure 5.1: Data Structure evaluation at t=2000.

In section 4, we already propose a linked list to maintain the current lineage aggregation result, based on the properties of the lineage aggregation result and our goal to reuse partial previous results to reduce required space of the result tuples. We also established that we need to perform an add and a remove operation on the list and the goal was to do this efficiently. Within the remove operation we need to find the lineage to remove. As of now we have an unsorted linked list, hence searching the list for a specific element takes linear time to the number of elements in the list. To be able to find a lineage in the list more efficiently, we add a second data structure. We add a priority queue sorted by and holding all end points of currently valid tuples. By connecting the end points from the priority queue to the lineages in the linked list that stop being valid at this end point, we prevent searching the unsorted linked list. The priority queue and the connecting pointers allow to find the lineages that need to be removed. Although, the lineage list is still an unsorted linked list, the access on the list occurs in a sorted manner.

The priority queue also serves as a dynamic schedule of the end points. The algorithm scans the tuples sequentially, hence when a tuple is processed its start point is the next start point in line. Since the end points also split the result intervals, the algorithm needs to know the end points to be able to split the result intervals correctly. The remove operation needs to be performed whenever the algorithm detects an end point that is between start points of input tuples or when it is processing the remaining end points after all input tuples are processed. Because the algorithm scans the input tuples sequentially, it is generally true, that the next point the remove operation needs to be performed is the lowest end point of all valid tuples. By holding all valid end points in ascending order in the priority queue, we ensure that the next end point is accessible in constant time.

As a consequence, the data structure consists of a priority queue, a linked list and the connecting pointers between the queue and the list elements. The priority queue is implemented by a Min-Heap. This ensures that the front element is the lowest end point of all end points from valid tuples. Each node in the priority queue is connected to at least one node in the linked list.

Figure 5.1 shows our data structure at time point 2000 after r_3 has been added. In the queue we have the end points 2003 and 2006 with links to the lineages that stop being valid at this end points. In this example we see that we prevent duplicates in the priority queue by the ability to have multiple connections from one priority queue node to several lineage list nodes. So instead of creating multiple nodes in the priority queue for tuples with the same end point, we create multiple connections from their end point to the lineages.

Add Operation

Whenever the algorithm reaches a start point of a tuple, the tuple becomes valid and we need to add its lineage to our data structure. In order to add a valid tuple and its lineage to our data structure, we need to complete three actions: a) prepend the new lineage to the existing lineage list, b) insert the end point of the new tuple in the priority queue, provided this end point does not already exist and c) connect the created list element with the created or existing corresponding end point.

In our example at time point 2000, we need to add the tuple r_3 to our data structure. Figure

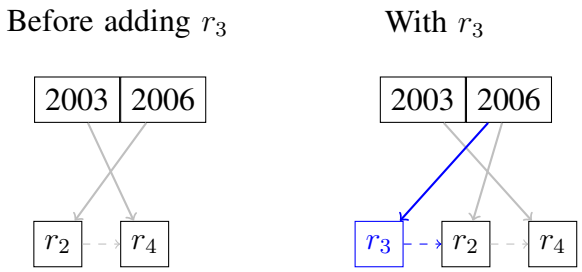


Figure 5.2: Updating Data Structure at t=2000.

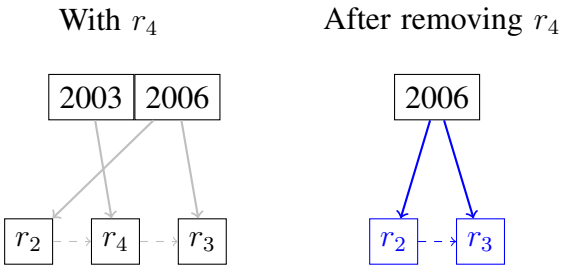


Figure 5.3: Updating Data Structure at t=2003.

5.2 shows us our data structure just before and just after adding r_3. We perform a) by creating a new list node and prepend it to the existing lineage list. The tuple r_3 has the end point 2006 which already exists in the queue for tuple r_2. Therefore it is not necessary to insert a new node in the queue. Lastly we connect the end point 2006 to the new list element r_3. Figure 5.2 shows the newly created nodes and pointers in blue. Notice that the lineage list nodes r_2 and r_4 are not blue. As discussed in the previous section, by prepending new lineages to the list, we can reuse the existing list and prevent creating a copy.

Remove Operation

When the algorithm reaches an end point, tuples with that end point stop being valid and need to be removed from our data structure. In the previous section we established the impossibility of removing a lineage from the existing list without altering the lineage aggregation result for the previous result tuple. Hence, when we perform the remove operation, we need to create a new list. Instead of copying the existing list and then remove the lineages that stop being valid, we only create copies of the list elements that remain valid. This can be achieved since we know that the lineages connected to the front end point in the queue are the lineages that need to be removed. Consider figure 5.3, the algorithm is at time point 2003 and needs to remove all tuples that have end point 2003. The tuple r_4 has the end point 2003, thus the lineage r_4 is connected to the end point 2003 in the queue. If there were multiple tuples with end point 2003, all of them would be connected to the end point 2003 in the queue.

By traversing the priority queue from the **second** element and onwards and following all connections to the lineages, we precisely access the lineages that remain valid. In the example in figure 5.3 the remaining lineages are r_2 and r_3. We traverse the priority queue starting from

ag_1

tid	$Name$	$Team$	T
r_1	Xabi Alonso	Liverpool	[2002, 2005)
r_5	David Bellion	Liverpool	[2005, 2007)

ag_3

tid	$Name$	$Team$	T
r_2	Niall Quinn	Sunderland	[1998, 2006)
r_4	Peter Reid	Sunderland	[1998, 2003)
r_3	Julio Arca	Sunderland	[2000, 2006)

Figure 5.4: Aggregation Groups sorted by start point.

the second element, which is the node 2006. Following the connections of the node 2006, we receive the remaining lineages r_2 and r_3. We create a new lineage list with all elements we access in this manner. Simultaneously, we connect the remaining end points to the newly created lineage list. Removing the front element in the queue, leaves the result that can be observed in figure 5.3. The newly created list elements and connections are highlighted.

5.2 Lineage-Update Algorithm

Figure 5.5 shows the Lineage-Update algorithm in pseudo code. The algorithm expects 3 parameters: the input relation r, the aggregate function F and a set of non-temporal attributes to group by, S. In a first step, the algorithm scans the input relation and creates aggregation groups sorted by start point. An aggregation group gathers all input tuples that have the same value(s) in the attribute(s) in S. For example in figure 5.4, we built the aggregation groups for our example relation in section 1, grouped by team and sorted by start point. The function findAG(input tuple r) in our algorithm returns the aggregation group that r belongs to. Preparing the input in this way leads to fewer comparisons while processing the input tuples.

Lineage-Update Algorithm (**r**,F, S)

```
1:  for each r ∈ r do
2:      if !(a ← findAG(r)) then
3:          Initialize new aggregation group a;
4:          ag ← ag ∪ a;
5:      a ← a ∪ r sorted by t_s;
6:  for each a ∈ ag do
7:      Initialize (Q, L);
8:      for each r ∈ a do
9:          if r.t_s > current.t & current.t ≠ −∞ then
10:             node ← Peek(Q);
11:             while node ≠ null & node.t_e ≤ r.t_s do
12:                 result ← WriteResult(node, (Q,L));
13:                 current.t ← node.t_e;
14:                 CopyRemove((Q,L), node);
15:                 node ← Peek(Q);
16:             if current.t < r.t_s then
17:                 result ← WriteResult(r.t_s, (Q,L));
18:                 current.t ← r.t_s;
19:         PrependLineage((Q,L), r.lineage);
20:         Insert new node with key = r.t_e in Q;
21:         Connect new node from Q with head of L;
22:         current.t ← r.t_s
23:         UpdateAggregation(current, r, F)
24:     node ← Peek(Q);
25:     while node ≠ null do
26:         result ← WriteResult(node, (Q,L));
27:         current.t ← node.t_e;
28:         CopyRemove((Q,L), node);
29:         node ← Peek(Q);
    return result
```

Figure 5.5: Pseudo Code of Lineage-Update Algorithm.

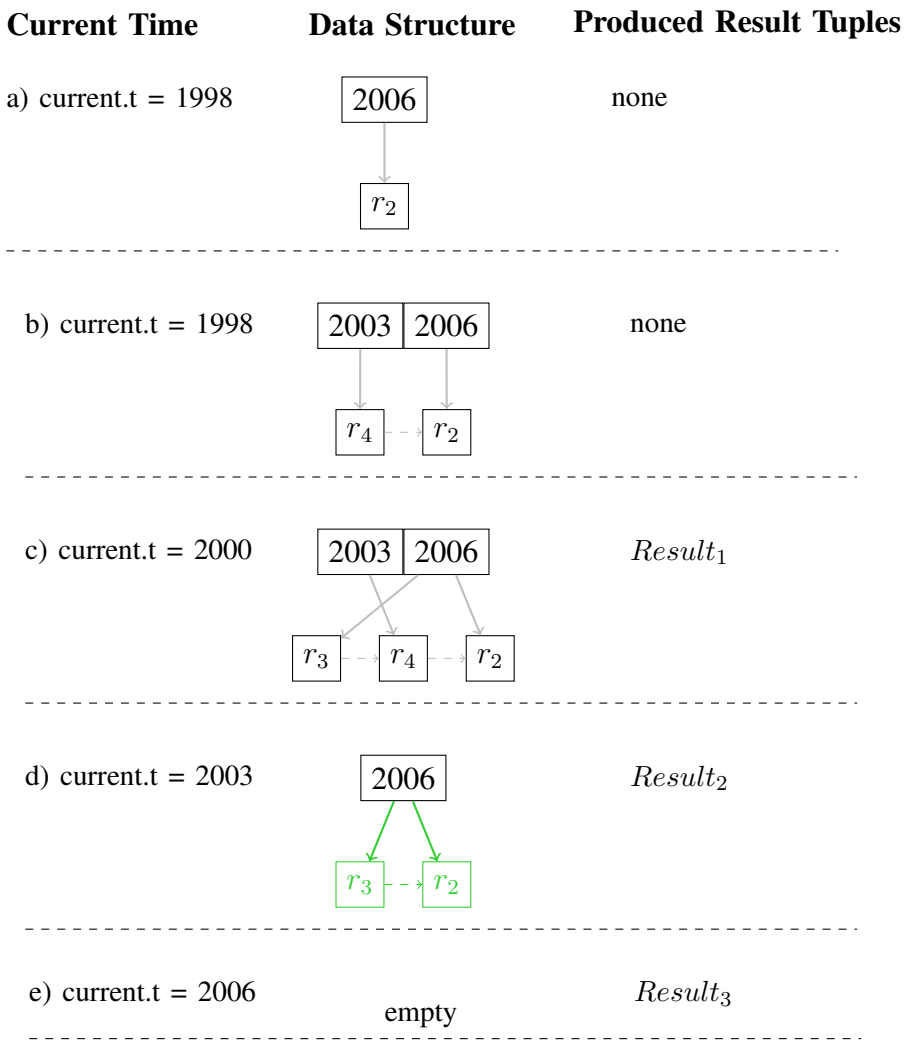
In the next step, the algorithm takes an aggregation group and goes through its tuples in chronological order. For every aggregation group, we first initialize the queue (Q) and the data lineage list (L) of our data structure (Q,L). For every new tuple, r, we compare in line 10 the start point of the tuple to the current time point (current.t) in the sweepline status. If $r.t_s$ is bigger, then we can produce the result intervals up to that start point. Line 10 to 15 then allows for the possibility that there are end points of previous processed input tuples in the interval [$current.t$, $r.t_s$). This insures that intervals are split by continuity of data lineage. After the while loop from line 11, the algorithm produced all result tuples up the biggest end point in the priority queue that is smaller than the start point of the current tuple. The if statement in line 16 then produces the results between this end point and the start point of the current tuple, if necessary. In line 19, we observe the copy and remove functions of the data lineage list we discussed in the previous sections. The algorithm continues by inserting the lineage of r in (L) and a new node (Q) with its key being the end point of the tuple that is being processed. In line 21, we connect the new node in (Q) with the new lineage node in (L). Line 22 updates the current time point up to which the results are produced and in Line 23 we update the intermediate result of the temporal aggregation. We repeat this procedure until all tuples in the aggregation group have been processed. At this point, the algorithm has computed the results for the current aggregation group up until the latest start point of the tuples. It continues to process the end points from (Q) to produce the remaining result tuples. When all end points have been processed, the algorithm continues with the next aggregation group. After processing all aggregation groups in this manner, the algorithm returns the final result.

We use the following example to highlight how the algorithm uses the data structure to compute the data lineages and stores them in the result tuple.

Example. *Consider the example query and relation from the Introduction section:* ***How many players played for each team at every point in time****. First the algorithm creates the aggregation groups based on the team attribute. For this example we focus on ag_2 from figure 5.4.*

The algorithm starts by processing the first tuple r_2. Since it is the first tuple, we add it to the data structure. The second tuple r_4 has the same start point. Hence there is no result interval to produce and we add r_4 to the data structure as seen in figure 5.6i)b. The algorithm proceeds with the tuple r_3. $r_3.t_s$ is bigger than current time point which is 1998 (the start point of the previously processed tuple). There are no end points smaller than $r_3.t_s$ in the data structure. The algorithm jumps to the if statement on line 16 and writes the first result tuple with interval [1998,2000) as seen in figure 5.6ii. Then it adds r_3 to the data structure. At this point all input tuples are scanned and the algorithm proceeds by processing the end points.

First it takes the lowest end point 2003 and produces the result tuple with interval [2000,2003) and lineage $r_3 \wedge r_2 \wedge r_4$. Notice in figure 5.6ii that the first two result tuples use the same lineage list for their result. They only point to a different node in the list to receive the correct result. Then the algorithm updates the data structure by removing 2003. As explained, when we remove lineage from the list, we create a new lineage list to preserve the previous results. This is highlight by the new color of the list in figure 5.6i)d. Processing the end point 2006 in the same way produces the last result tuple.



(i) Evaluation of Lineage-Update algorithm after processing r_2, r_4, r_3.

	Team	Interval	Count	Lineage
$Result_1$:	Sunderland	$[1998, 2000)$	2	
$Result_2$:	Sunderland	$[2000, 2003)$	3	
$Result_3$:	Sunderland	$[2003, 2006)$	2	

(ii) Result Tuples with Lineage Lists.

Figure 5.6: Lineage-Update algorithm example.

5.3 Complexity

In comparison to the cumulative and the selective aggregation, the lineage aggregation poses the biggest challenge performance wise. Therefore, we focus on lineage aggregation in the complexity analysis of the Lineage-Update Algorithm. Within an aggregation group we have three recurring steps, a) removing operation of our data structure, b) add operation of our data structure and c) writing lineage aggregation result to result tuple.

As we have discussed, the remove operation requires to fetch all elements in the list that stay valid. This leads to a linear complexity n_v, where n_v is the number of valid tuples. Removing the front element from the queue is essentially a pop function on a queue and only has a constant complexity. The add operation on our data structure can be divided into searching the priority queue, inserting a new node in the priority queue and prepending a node to the lineage list. Prepending a node to the lineage list takes constant time. We implemented a heap as underlying structure of the priority queue, hence insertion has a complexity of $\log n_v$ and searching has linear complexity to n_v. As we use a pointer to the head of the lineage list to store the lineage aggregation result in the result tuple, the cost of connecting the result tuple with the lineage list is constant. At this point and for this result interval, the heavy-lifting has already been done by the add and remove operations. Combined, lineage aggregation with our Lineage-Update Algorithm has a complexity of $\mathcal{O}(n_r * \max(n_v, n_v))$ which is $\mathcal{O}(n_r * n_v)$, where n_r is the number of tuples in the input relation. The worst case is $\mathcal{O}(n_r^2)$ for input where start and end points of the input tuples are distinct and there is some time point at which all input tuples are valid.

6 Experiments

We carried out several experiments comparing our new Lineage-Update algorithm to the TMDA-CI [1], the Temporal Alignment [3], the Disjoint Interval Partitioning [2] and the Timeline Index [4]. We use synthetic data to specifically investigate certain special cases and n the end, we compare the algorithms on real world data.

6.1 Experimental Setup

For the experiments, we used a 2 x Intel(R) Xeon(R) CPU E5-24400 @2.40GHz machine, with 64GB main memory, running CentOS 6.7. All experiments were conducted using only main memory. We enhanced the TMDA-CI Algorithm [1], the Timeline Index [4] and the Disjoint Interval Partitioning [2] with lineage aggregation, to be able to compare them with our Lineage-Update Algorithm. These algorithms have been implemented by the authors in C. We also compare these algorithms to the Temporal Normalization [3], which is integrated in postgres. In the legends of the figures, we use the following abbreviations. Lineage Update: LU, Timeline Index: TI, TMDA-CI: TMDA, Disjoint Interval Partitioning: DIP and the Temporal Normalization: TN.

TMDA: The TMDA-CI algorithm stores the currently valid tuples in a balanced search tree sorted by the end point. If the input tuples need to be grouped by an attribute, the TMDA will maintain a separate tree for each group. Since the TMDA can store the tuples with its lineage in the tree, we can traverse this tree to compute the data lineage.

Timeline Index: The Timeline Index does not store the currently valid tuples. For selective and cumulative aggregation the Timeline Index stores an current result, but none of these data structures is suited to keep the lineages of the currently valid tuples. Nevertheless, Kaufmann et al. [4] suggest for other types of aggregation to use the Timeline Index as a window to seek the currently valid tuples. Consequently, we need to traverse the Event List of the Timeline Index from the top to the the current point to see what tuples are currently valid. The Version Map of the Timeline Index provides the time points at which the result intervals are split, thus the lineage needs to be computed. This leads to re-traversing the Event List for every new result tuple. While traversing the Event List a double linked list is used to add and remove the lineages from the tuples. This list represents the data lineage after the Event List is traversed to the current time point.

Disjoint Interval Partitioning: We use the suggested temporal aggregation algorithm from Cafagna and Böhlen [2]. This means we create the disjoint partitions from our input relation

and then perform full outer joins between all partitions with the DIPMerge algorithm introduced by Cafagna and Böhlen [2]. The lineage is represented as a list. When performing DIPMerge the lineage lists of two tuples that are joined are concatenated. This ensures that after all partitions are joined the resulting lineage list is the data lineage for that result interval.

6.2 Synthetic Data

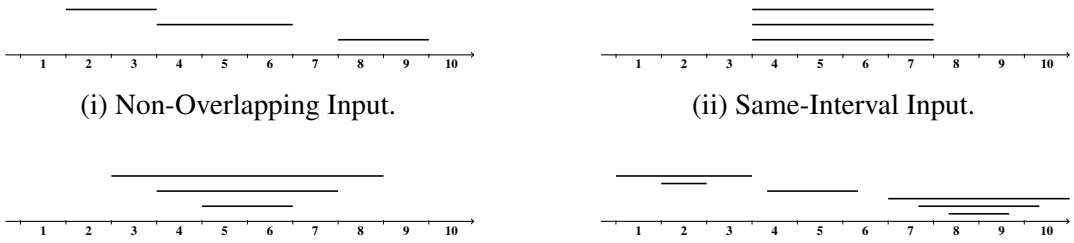
We generated synthetic data to examine the algorithms in specific input cases. We characterize the data by considering two factors: a) *max overlap*, maximum number of input tuples overlapping in a single time point and b) *distinct time*, number of distinct start or end points of these overlapping tuples. The number from factor a) defines the maximum size of a data lineage. The bigger the number of overlapping tuples in one time point, the bigger the correct data lineage in that time point. A bigger data lineage leads to a more expensive lineage computation. The number of distinct start and end points directly determines the number of result intervals. The number of result intervals influences the runtime, since the data lineages need to be computed for every result interval. Hence, we are interested in factor b), that is the number of distinct start and end points of the tuples that are overlapping. So the number of lineage computations that need to be done with large lineages. When the query requires to group the input tuples in more than one group, then we are interested in the max overlap and the distinct time factors of the input tuples within such a group.

	max overlap	*distinct time*
Non-Overlap	low	low
Same Interval	high	low
Worst Case	high	high
Random	low	high
Multiple Groups	high within group	high

Figure 6.1: Characterization of Input Cases.

We generated the five different input cases from figure 6.1. In the first case, Non-Overlap, the tuples are arranged as shown in figure 6.2i. The expected result is exactly the input relation, so there is nothing to aggregate. Same Interval is case where all input tuples overlap, thus the *max overlap* number is high. But at the same time all the input tuples have the same interval and generate only 2 distinct start and end points. The expected result is only one tuple with the interval from the input tuples and the data lineage representing all input tuples. Then we generated data that induces the worst case runtime for all approaches we tested. The structure in 6.2iii represents this worst case. The *max number* is equal to the number of input tuples and these input tuples have distinct start and end points. Next, we examined the algorithms by using data which randomly determined whether the new input tuple overlaps with the previous input tuple. The probability that the new tuple overlaps with the previous is 0.5. The input relation generated in this way has a low *max overlap* number around 15, but these overlapping tuples have distinct start and end points. Lastly, we generated data with $26^2 = 676$ different groups by adding two attributes with each on character to the input. We assign the input tuples

randomly to one of these groups. The algorithms then have to group by these attributes. Within a group, the tuples are arranged as in the worst case. The size of a group fluctuates around the number of input tuples divided by the number of groups.



(i) Non-Overlapping Input.

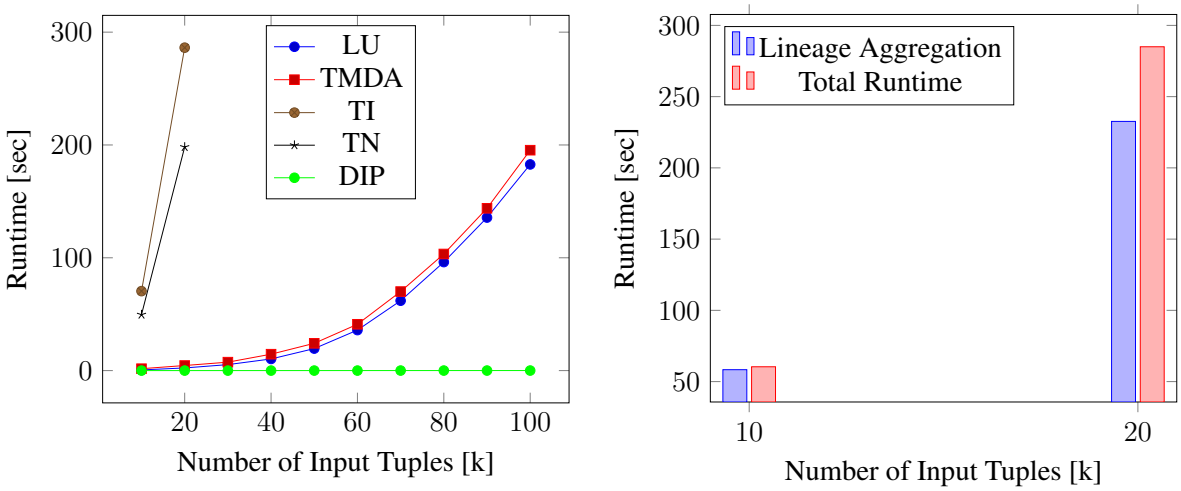
(ii) Same-Interval Input.

(iii) Worst Case Input.

(iv) Random Case Input.

Figure 6.2: Interval characteristics of synthetic input data.

Non-Overlap: The case of non-overlapping tuples is a simple one, since there is nothing to aggregate and the result is exactly the same as the input relation.



(i) Non-Overlapping Input Run-Time Result.

(ii) Runtime distribution for TI.

Figure 6.3: Experiment Result for Non-Overlapping Input.

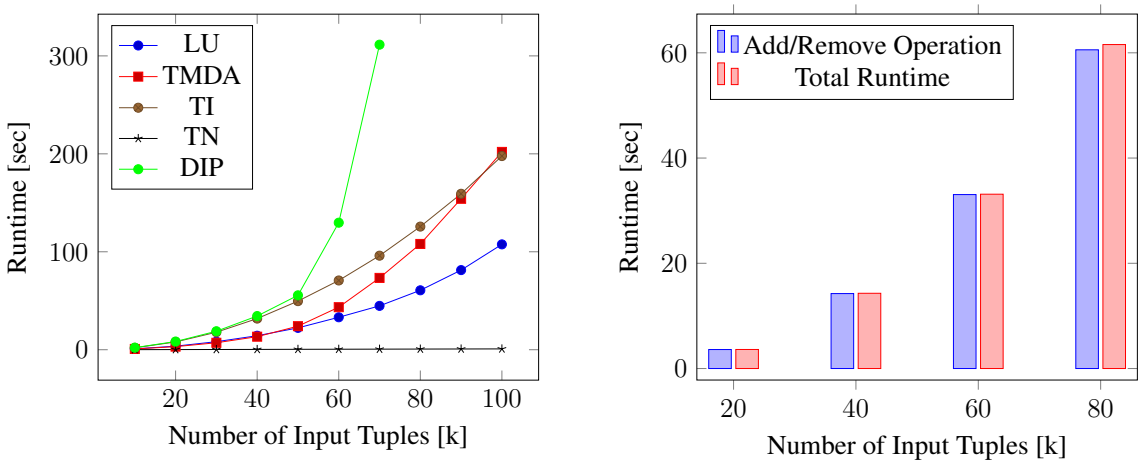
Figure 6.3i shows the runtime results for all approaches in the Non-Overlap input case. The Timeline Index and the Temporal Alignment do not perform well in this case. The runtime of the Temporal Alignment is determined by the time of the outer join of the input relation with the union of the start and end points of the input relation. Since the input tuples have distinct start and end points, the union of the projection of these is big. Consequently the join is expensive.

In the Experimental Setup section we explained how we compute the data lineage with the Timeline Index. This lineage computation primarily led to the bad runtime result for the

Timeline Index. Figure 6.3ii shows the proportion of the lineage computation to the total runtime. Despite the easy input case, the Timeline Index still traverses the Event List for every new result tuple to compute the data lineage.

The TMDA and the Lineage-Update perform practically the same. Both need to maintain their data structures with the current valid tuples to compute the result. The DIP approach is the only one that exploits the fact that the input relation is the result. Since there are no overlapping tuples, DIP creates only one partition and does not need to perform DIPMerge.

Next, we investigate the Same Interval input case. We reduce the distinct start and end points to two but all input tuples are overlapping. The only result interval of lineage aggregation is the interval of the input tuples. Therefore the data lineage only needs to be computed once. The size of the data lineage is n, with n being the number of input tuples.



(i) Run-Time Result for Same-Interval Input.

(ii) Runtime distribution for the LU.

Figure 6.4: Experiment Result for Same-Interval Input.

The results in figure 6.4i show that the Temporal Alignment approach outperforms all other approaches. In contrast to the Non-Overlap input case, we now have only one distinct start and one distinct end point. This leads to a union of start and end points with size 2. Hence, the outer join of the input relation with this union is now considerably faster than with many distinct start and end points.

The runtime of Lineage-Update is in this case dominated by the data structure operations. Figure 6.4ii reveals that the add and remove operation on the Lineage-Update data structure combined, represents almost the complete runtime. Similar is true for the TMDA algorithm and the operations on the tree that holds the currently valid tuples.

The DIP algorithm has to create n different partitions since all n input tuples are overlapping. With n partitions, the DIP algorithm has to perform DIPMerge n-1 times. We can see that the DIP is especially sensitive to the *max overlap* number. The more tuples overlap, the more partitions are required and DIPMerge needs to be computed more often.

The worst case of all approaches is to maximize the *max overlap* and the *distinct time* number. This maximizes the size of the data lineages, thus making the lineage computations more expensive. Simultaneously, the lineage must computed multiple times, since the input tuples have distinct start and end points.

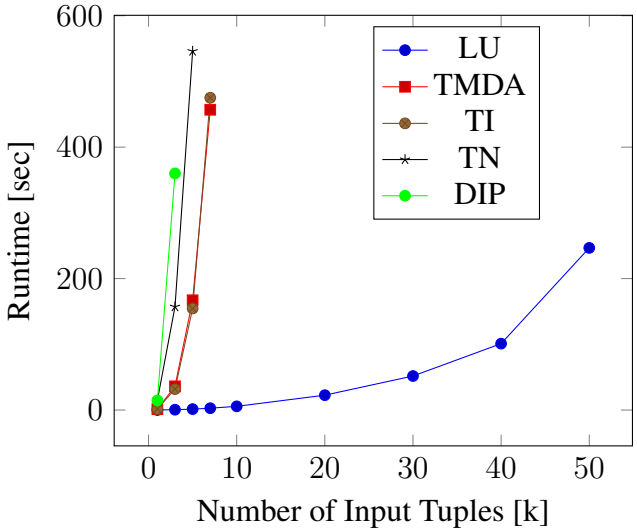


Figure 6.5: Worst Case Input Run-Time Result.

In figure 6.5 we can see the result of the algorithms with the worst case input. The results show that the Lineage-Update algorithm outperforms all existing approaches. As in the last experiment, the DIP approach requires n partitions with n being the number of input tuples. The Temporal Alignment needs to perform the outer join on big relation, since the input has distinct start and end points. The Timeline Index still needs to traverse the Event List from top to compute the lineage. The TMDA is slow, because the tree that holds the currently valid tuples has to hold up to n elements and the lineage is computed by traversing the tree.

This is the input where computing the lineage is most expensive and the Lineage-Update algorithm is the only approach that exploits the semantics of lineage aggregation to improve efficiency when computing data lineages. Therefore it is no surprise that the Lineage-Update algorithm is the only approach that stays fairly robust on worst case input. The data structure that holds the current lineage allows the Lineage-Update algorithm to update the lineage aggregation result instead of recomputing it.

The next experiment uses random input case. This case has a fairly low *max overlap* number. It fluctuates around 15. If the tuple overlap, then they have distinct start and end points. Hence, within these overlapping tuples, the *distinct time* number is high.

Figure 6.6 presents the runtime results with randomized input. Since the *max overlap* number is relatively low compared to the size of the relations, the DIP approach performs the best. The Temporal Alignment still performs badly for input with distinct start and end points and the Timeline Index requires too much time to compute the lineage. The TMDA and the

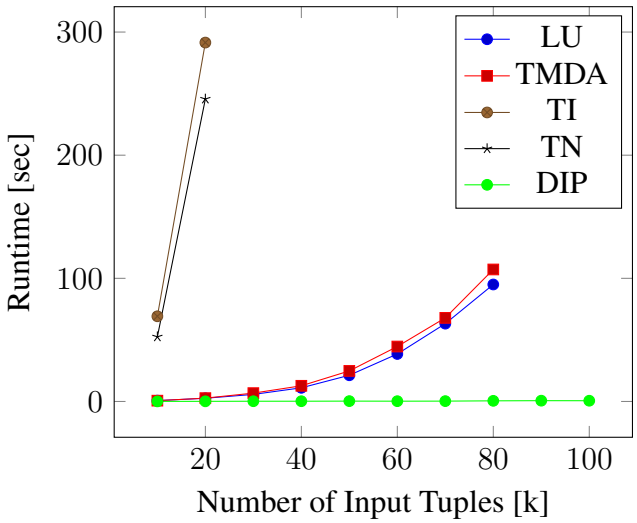


Figure 6.6: Run-Time Result of Randomized Input.

Lineage-Update perform similarly. Both process the input tuples sequentially and use a data structure to keep the currently valid tuples or lineage. These data structures remain small, because of the low *max overlap*.

Lastly, we wanted to investigate the algorithms when they have to group the input by two attributes as well as perform lineage aggregation on these groups. The Lineage-Update, Time-line Index and DIP algorithm all create the aggregation groups when sorting the input. Then the algorithms are performed on each group one another. The TMDA still processes the input tuples sequentially and assigns the input tuples to the corresponding aggregation groups on the fly.

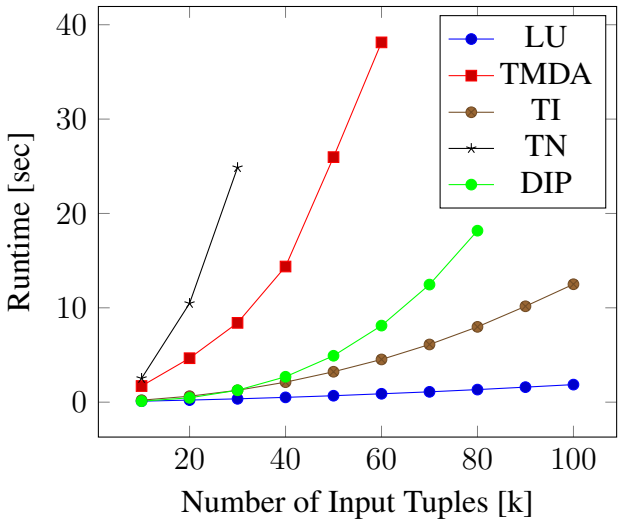


Figure 6.7: Run-Time Result for Multiple Groups Input.

The result in figure 6.7 shows that the Lineage-Update algorithm still performs better than the other approaches when the query requires to group the result tuples. Within a group, the input is arranged similar to the worst case input. Therefore, it was expected that the Lineage-Update algorithm outperforms the other approaches.

The Timeline Index performs with multiple groups better than in any other input case. This is due to the size of the relation the index is created on. In this case, a new Index is created for each group, hence the Event List becomes considerably smaller. Consequently, computing the data lineage becomes less expensive and the Timeline Index performs better.

Since *max overlap* is not big but grows with the number of input tuples, the DIP performs well for small input relations and gets worse with increasing the number of input tuples.

Similar to the results before, the Temporal Alignment is slow whenever there are a lot of distinct start and end points. And the TMDA is the only approach that handles the grouping on the fly. This in conjunction with the worst case input structure within a group, lead to the high runtime of the TMDA for this input case.

6.3 Real World Data

The real world data comes from the WebKit Open Source Project[6]. The real world data represents git commits on single files. This leads to a lot of tuples with the same identifier, thus only few different aggregation groups. Within an aggregation group there are blocks of overlapping tuples with only few distinct start or end points. This does result in data lineages with respectable size, but fewer data lineage computations.

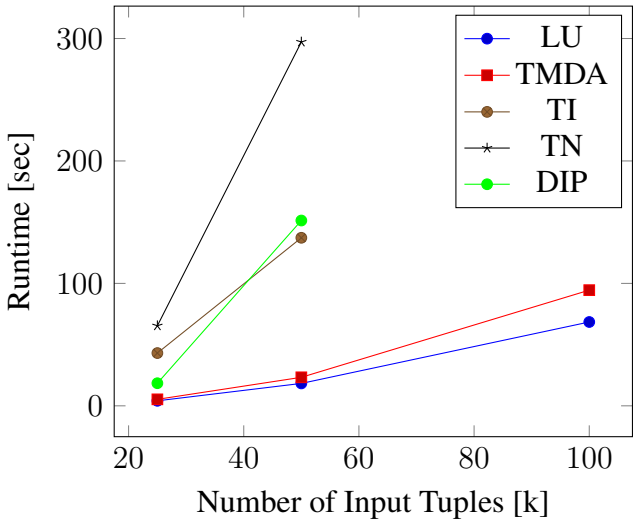


Figure 6.8: Runtime results with Real World Data.

Figure 6.8 gives us the result of the algorithm runtime for real world data. We see that the sweeping based approach of the TMDA and Lineage-Update seem to suit the characteristics of the input data. As in all our experiments, the Timeline Index is dominated by the time used for the lineage computation. The performance of the Timeline Index is directly linked to the size

of the relation the index is created on. With large groups in this dataset, the Timeline Index performs rather badly. While *max overlap* is not as big as in the worst case, it is sufficiently large that the DIP algorithm has to create many partitions. Therefore requires DIPMerge several times, which slows the DIP algorithm down. As in all cases with synthetic input except for the Same Interval case, the Temporal Alignment approach can not keep up with the other approaches.

Due to the more efficient lineage computation of the Lineage-Update, it is able to just outperform the TMDA. Nevertheless, the TMDA outperforms any other approach. It benefits in comparison to the other approaches from the low number of groups, low number of lineage computation with a respectable size of data lineages. Since the dataset contains blocks and an element in the tree of current valid tuples can hold multiple tuples, the tree stays small. So, although the data lineages are of respectable size, the tree stays small and traversing it to compute the lineage is fairly fast.

7 Conclusion

In this thesis we introduced lineage aggregation as a new type of aggregation. Determining its semantics allowed us to develop a data structure and the Lineage-Update algorithm that uses this data structure to improve efficiency when computing data lineages on temporal relations. Instead of recomputing the data lineage for every new result tuple, the Lineage-Update algorithm maintains and updates the current data lineage based on the semantics of the lineage aggregation. The algorithm is capable to include data lineages in result tuples of temporal aggregation queries. Empirical Experiments have shown that, especially in the worst case input scenario of temporal aggregation algorithms, the Lineage-Update algorithm outperforms established temporal aggregation algorithms. Additionally, the Lineage-Update algorithm requires less space to store the data lineages in the result tuples due to a clever way of reusing previous data lineages.

Bibliography

[1] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. "Multi-dimensional Aggregation for Temporal Data". In: *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*. Ed. by Yannis E. Ioannidis et al. Vol. 3896. Lecture Notes in Computer Science. Springer, 2006, pp. 257–275. ISBN: 3-540-32960-9. DOI: 10.1007/11687238_18. URL: http://dx.doi.org/10.1007/11687238_18.

[2] Francesco Cafagna and Michael H. Böhlen. "Disjoint Interval Partitioning". In: *The VLDB Journal* 26.3 (June 2017), pp. 447–466. ISSN: 1066-8888. DOI: 10.1007/s00778-017-0456-7. URL: https://doi.org/10.1007/s00778-017-0456-7.

[3] Anton Dignös et al. "Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries". In: *ACM Trans. Database Syst.* 41.4 (2016), 26:1–26:46. DOI: 10.1145/2967608. URL: http://doi.acm.org/10.1145/2967608.

[4] Martin Kaufmann et al. "Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 1173–1184. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465293. URL: http://doi.acm.org/10.1145/2463676.2465293.

[5] Raghotham Murthy, Robert Ikeda, and Jennifer Widom. "Making Aggregation Work in Uncertain and Probabilistic Databases". In: *IEEE Transactions on Knowledge & Data Engineering* 23 (2010), pp. 1261–1273. ISSN: 1041-4347. DOI: doi.ieeecomputersociety.org/10.1109/TKDE.2010.166.

[6] *The WebKit Open Source Project*. http://www.webkit.org. 2012.