Freie Universität Bozen

Libera Università di Bolzano

Free University of Bozen · Bolzano

Fakultät für Informatik | Facoltà di Scienze e Tecnologie informatiche | Faculty of Computer Science

Large-Scale Temporal Aggregation Using MapReduce

Philipp Hofer

supervised by

Prof. Johann Gamper

Acknowledgments

I would like to thank my supervisor Prof. Johann Gamper for the valuable support during the development of the algorithms and the writing of this thesis. His availability for discussions and his knowledge about temporal databases and temporal aggregation were fundamental for the refinement of the single approaches and their effectiveness.
Further I would like to thank my fellow students for the great time at the university. Especially, Andreas Heinisch, Manfred Malleier, Simon Carraro, Johannes Erschbamer and Michael Gurschler became good friends who also supported me in many occasions during the lectures and the preparations for the exams. I had a great time during the funny and adventurous trips of our little group of computer scientists.
Last but not least I would like to thank my family who supported me during all these years of studies. Especially, my mother, without her moral encouragement I would never been able to finish my studies. Further, I would like to thank my brother Armin for the relaxing hikes during the summer to recharge the batteries and so start the fall semesters with new energy and focus.

Thank you all! Philipp

Abstract

Temporal aggregation is a method to compute a compact summary of an input dataset that contains time varying information. Many different application domains, such as finance and environment, record huge amounts of data from which compact summaries have to be extracted. Due to the massive growth of the amount of such data, fast query processing becomes a challenge. In particular, computing instant temporal aggregation (ITA), a special form of temporal aggregation, is an expensive and time consuming process. To tackle these problems of processing massive datasets, various techniques and infrastructures have been proposed, including the use of clusters or networks of commodity machines. One of the most popular programming models recently proposed is MapReduce, which provides a framework for automatic parallel data processing in a cluster.

In this thesis, we propose three different algorithms that implement the abstract programming model MapReduce in order to compute instant temporal aggregation for a given input relation. All three approaches are based on different strategies to compute temporal aggregation, hence provide distinct characteristics and strengths. The first approach is a straightforward extension of the traditional way to compute temporal aggregation. The second algorithm processes small chunks of the input file and iteratively merges the partial results to obtain the final relation. The third approach divides the timeline into independent partitions which can be processed in parallel. Furthermore, we report the results of a detailed empirical evaluation of the algorithms using differently structured datasets. All algorithms were implemented in Java and executed on a small cluster on top of the MapReduce framework Apache Hadoop. Additionally, we performed a comparison of the three MapReduce solutions with the existing Bucket algorithm for large-scale temporal aggregation. The experiments show that our algorithms outperform the Bucket algorithm by a factor of 5 to 20.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Nowadays, dealing with large amounts of raw data is become a common task encountered in many situations, e.g. social networks or company management. Numerous application have to handle gigabytes or even terabytes of data and the user expects that tasks are completed in a reasonable amount of time. For this reasons computation done using clusters, large networks of many commodity machines connected through a network, becomes increasingly important. In order to be able to spread the computation over many single workers, Google introduced the new abstract programming model MapReduce [5]. Another important advantage of a cluster is that the single machines and so the hardware are easily replaceable and far cheaper than high-end components used in state-of-the-art servers. For this thesis the free open-source version of MapReduce Hadoop is used to run the developed approaches that do parallel computations on the cluster.

A technique of extracting useful information out of time varying data is temporal aggregation. The increasing importance of this kind of aggregation comes from the fact that computers have invaded nearly every aspects of live, constantly monitoring time relevant data, e.g., in the environmental, medical or financial domain. Temporal aggregation, in contrast to standard aggregation in a relational database, does not return a single scalar value but always an aggregation relation. The reason for this behavior is that temporal aggregation is strongly related to the included temporal dimension. Every result value is bound by a so called constant interval [2], a set of time instants in which the relation remained fix, i.e. no new tuple starts or ends. In the past three different types of temporal aggregation have been defined, *Instant Temporal Aggregation* (ITA), *Moving Window Temporal Aggregation* (MWTA) or *Cumulative Temporal Aggregation* and *Span Temporal Aggregation* (STA) [2].

In instant temporal aggregation, the most important of the three types, the time domain is partitioned into the finest possible time unit, namely time instants t. Afterwards, the aggregation groups are formed, by associating tuples from the input relation with the time instant t. To each aggregation group the aggregation operator is applied producing an aggregation value for each specific time instant. At the end, value equivalent, consecutive tuples are coalesced leading to maximum time intervals. Due to their nature of involving large amounts of tuples during the computation and partitioning the timeline in the finest possible units, calculating the temporal aggregation result is a resource and time intensive procedure decreasing the availability of updated results in frontend applications.

The concept of moving window temporal aggregates was first presented by Navathe et al. [15] and in later works also referred to as cumulative temporal aggregation [20] [26]. In MWTA the aggregation groups are not only influenced by tuples of the argument relation holding at a single time instant t, but by tuples valid at a time interval $[t - w, t]$ with offset $w > 0$. After applying the aggregation function to each group, value-equivalent tuples are coalesced leading to result tuples over maximum time intervals.

The last form of temporal aggregation is span temporal aggregation. In this form of temporal aggregation, the constant intervals are predefined and therefore

independent of the argument relation. The aggregation groups are formed by grouping tuples overlapping the predefined time intervals. As a last step, the aggregation operator is applied to each aggregation group producing the result relation. Note that for this type of temporal aggregation no coalescing is needed, because the constant intervals of the result relation are totally independent from the time intervals of the argument relation.

1.2 Running Example

Consider the temporal relation shown in Figure 1. It displays six datatuples of three different employees, having a name, a salary warring through time and the related timestamp in which the salary is valid. Assume that the result of the following query has to be retrieved: *Q1: What is the total amount of employees working for the company at any time since the foundation ?* The resulting relation of the aggregation query $Q1$ is shown at the bottom of the diagram in Figure 1. In Table 1a the initial relation, whereas in Table 1b the final numerical result is displayed together with the input tuples that participate to the result tuples. For instance, the tuples r_1 and r_2 have the same aggregation value, but due to the preservation of the lineage information (i.e. different tuples produce different result tuples) they generate two separate result records. Further the time interval $[13, 15)$ has a aggregation result of 0, which could be omitted without any information loss, in order to save computation time and space. As a last point, the previously mentioned case applies also to the intervals $[-\infty, T_1)$ and $[T_n, \infty)$ where T_1 is the first and T_n is the last timepoint of the input relation.

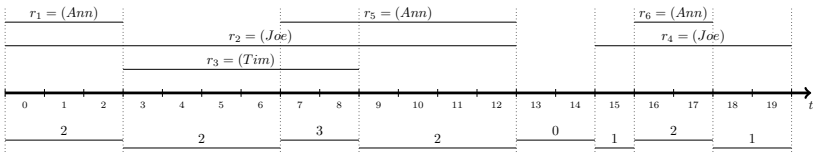


Figure 1: Temporal Relations Example

ID	Name	Salary	T
r_1	Ann	120	$[0, 3)$
r_2	Joe	110	$[0, 13)$
r_3	Tim	80	$[3, 9)$
r_4	Joe	150	$[15, 20)$
r_5	Ann	170	$[7, 13)$
r_6	Ann	180	$[16, 18)$

(a) Example Tuples

T	Count	Tuples
$[0, 3)$	2	$\{r_1, r_2\}$
$[3, 7)$	2	$\{r_2, r_3\}$
$[7, 9)$	3	$\{r_2, r_3, r_5\}$
$[9, 13)$	2	$\{r_2, r_5\}$
$[13, 15)$	0	$\{\}$
$[15, 16)$	1	$\{r_4\}$
$[16, 18)$	2	$\{r_4, r_6\}$
$[18, 20)$	1	$\{r_4\}$

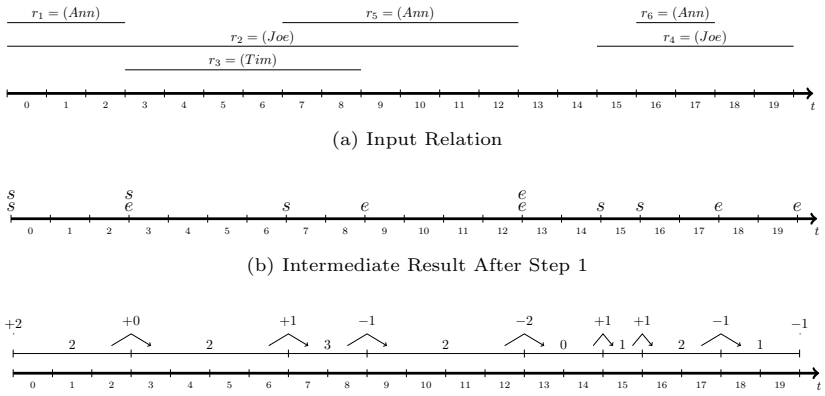
(b) Result Relation using Count

Table 1: Initial Relation and Result

1.3 Basic Concepts of Temporal Aggregation

The basic concept used by the developed approaches consists of three steps and was first presented by Bongki Moon et al. [14]. As a first step (Figure 2b), the

timestamp of the read tuples is extracted and each single timepoint is associated with a tag indicating weather it is a starting or an ending point. As a second step, the timepoints and tags are sorted in increasing order of the values. In order to compute the count aggregate, the entire timeline is scanned from beginning to end. Further a counter is needed, which is incremented by one each time start tag is encountered and decremented by one each time a end tag is processed (Figure 2c).



(a) Input Relation

(b) Intermediate Result After Step 1

(c) Final Result of ITA

Figure 2: Basic Idea of Temporal Aggregation

In Table 2 we provide some descriptions of the common notations used throughout the subsequent chapter:

Notation	**Description**
$\mathcal{P} = \{P_1, \ldots, P_p\}$	Set of Temporal Partitions
$\lvert\mathcal{P}\rvert = p$	Total Number of Partitions
$P[T_s, T_e]$	Partition with Valid Time Interval
$P.T_s, P.T_e$	The Starting and Ending Point of the Partition
$P1, P2, \ldots$	Specified Partition Number, e.g. Partition 1, Partition 2, ...
P	Unspecified Partition Number
n	Total Number of Tuples in Input File
R	Total Number of Available Reducers

Table 2: Notations

1.4 Contribution

Different algorithms have been proposed in order to efficiently calculate the ITA result of a given input relation. Some of the approaches introduce the parallel computation of the temporal aggregation. However, none of them involves the abstract programming model MapReduce and not do consider the advantages provided by that model. The benefits of executing the algorithm on a cluster, consisting of hundreds of commonly machines connected through a network, are superior computational power and the use of low-cost hardware. Further, due to the use of a distributed file system, input relation of arbitrary length, bigger in size than any single hard disk accessible by the cluster, can be handled.

In this thesis we are going to present three approaches that enable computation of temporal aggregates on a Hadoop cluster. The algorithms are based on different assumptions, but all implement the MapReduce programming model. The main contributions can be summarized as follows:

- The first developed approach, the *HTA-Standard* algorithm, is based on the basic technique presented in the work of Bongki Moon et al. [14] of computing temporal aggregation results. It uses three MapReducer iterations to produce the final result relation. The first MapReducer run tags the starting and ending points of the single valid time intervals. The second iteration divides the timeline into partitions and applies the aggregation function to the previously marked timepoints. The final iteration changes the aggregation results according to the values transferred between the previously created temporal partitions.

- The next approach, *HTA-Merge*, starts by computing the aggregation relation of small chunks of the input file. These partial results are then merged using a variable amount of MapReduce iterations. The exact number of iterations depends on the produced amount of partial results. At the end a single output is produced containing the final aggregation relation.

- The third algorithm, *HTA-Partition*, uses only a single MapReduce iteration to complete the final result. The timeline is again partitioned into several parts and tuples crossing the bounds of those partitions are split into single units. This technique allows an autonomous parallel processing of the various partitions, keeping the number of iteration to one.

- All developed algorithms are implemented in Java and executed on a cluster running the MapReduce implementation Hadoop. Further differently structured input file with increasing amount of tuples were used as input relation, recording the execution times of the approaches.

- The developed algorithms are further compared to the *Bucket & Balance Tree* approach developed by Moon et al. [14]. The combination of both approaches is needed, because the *Bucket* algorithm prepares the input files for the main memory based *Balance Tree* approach, which computes the final temporal aggregation relation.

1.5 Organization of the Thesis

The thesis is structured in the following way: in Section 2 we discuss the relevant related work, together with some technical details about the MapReduce framework Hadoop. At the beginning of Section 3 we shortly introduce the basic calculation concept of temporal aggregation together with an example and explain some general terminology and functions used throughout the subsequent chapters. The rest of the section describes the first algorithm *HTA-Standard* and the exact step by step processing of the example relation (subsection 1.2). In Section 4 we describe the MapReduce iterations of the second algorithm, *HTA-Merge*, together with some implementation details. Section 5 illustrates the MapReduce run of the third developed algorithm, *HTA-Partition*. Section 6 shows the results of the experimental evaluation using two different real-world

scenarios with increasing amount of tuples as input relation. Further the advantages and disadvantages of the single approaches are highlighted referring to the measured execution times. In addition we provide the promising results of the comparison with another temporal aggregation algorithm. The thesis concludes with some final remarks in Section 7 and an outlook regarding future work, including possible improvements and extensions.

2 Related Work

2.1 Temporal Aggregation

The work of Tuma [23] extends the approach of computing aggregates in a relational database of Epstein [7] and suggests an evaluation process for temporal aggregates. First, the constant intervals are computed needing a scan of the argument relation. Next, the aggregation groups are formed to which later on the aggregation operator can be applied. Tuples overlapping a constant interval are selected and partitioned by the constant time interval they cross into the aggregation groups. At the end, the result relation is formed by associating the resulting aggregation values with the argument relation. This approach needs two scans of the input relation, one in order to compute the constant intervals and the second to compute the aggregate function, leading to a worst case running time of $O(mn)$, where m is the amount of result tuples and n is the magnitude of the argument relation.

Another approach which reuses and refines the evaluation method of Tuma is the work of Kline et al. [11]. They use a tree like structure, called aggregation tree, to store partial aggregation results in main memory. The aggregation tree is incrementally constructed while scanning the argument relation once. The leaf nodes of the tree represent the constant intervals of the resulting temporal aggregation. The final aggregation relation is formed, by traversing the tree from the root ($[0, \infty]$) to the leaf-nodes in depth-first order, summing up all encountered partial aggregation values. The run time of the algorithm depends on the sort order of the input relation. Since the tree is not balanced, a ordered argument relation leads to a linked list and a worst time complexity of $O(n^2)$. Another drawback leading to an drastic increment in the run-time of the algorithm is the dependency on the available main memory. In the case, the whole tree can not be held entirely in main memory, the computation causes page swapping and therefore a lot of disk-seeks.

Moon et al. [14] addressed the balancing problem of the aggregation tree of Kline. They split the five aggregation operators into two groups, namely *count*, *sum* and *avg* on one side and *min* and *max* on the other side. The distinction is done according to the computational effort needed to keep track of the aggregation values. For each group of aggregation operator a different algorithm was proposed, for the first group a balanced tree based method and for the later one a merge-sort variant. The tree based algorithm stores the start and end timestamps together with the two partial aggregation value starting and ending at that timestamp. To allow a dynamic run-time balancing of the tree, each node contains an additional color tag as used by the red-black tree insertion algorithm. The balancing handles the problem of the aggregation tree according the creation of linked list if the input relation is sorted by the timestamps leading

to a run-time of $O(n^2)$. The result relation is then computed by an in-order traversal of the tree aggregating the values of all nodes encountered. Storing information of the constant intervals not only in the leave nodes, but also in the internal nodes, reduces the amount of nodes hold in main memory by about half the nodes required by the aggregation tree algorithm. The merge-sort based algorithm uses a bottom-up approach to calculate the aggregation values. It uses a divide-and-conquer strategy to recursively merge two intermediate results ending up with the final aggregation result. Both proposed algorithms have a time complexity of $O(n \log n)$ for a argument relation of n tuples. Moon et al. proposed also a solution to address the main memory limitation of the aggregation tree. The time-line of the input relation is partitioned into buckets and the aggregation operator is applied to each bucket independently. The partial aggregation results of long-lived tuples, tuples spanning more than one bucket, are kept in a separate meta-array. As a last step the partial aggregation results of the buckets and the meta-array have to be combined leading to the final aggregation relation.

The so called SB-tree or segmented B-tree proposed by Yang et al. [26] is a balanced, disk based index structure that allows lookups of temporal aggregate values by time-instants in $O(h)$ time, where h is the height of the SB-tree. Additionally, the tree supports efficiently incremental insertions and deletions making it more attractive in comparison to materialized database views maintaining temporal aggregates. Together with the algorithms computing instant temporal aggregation and cumulative temporal aggregation Yang et al. [26] developed also modified versions of their SB-tree. The JSB-tree or joint SB-tree uses two SB-trees to further increment the effectiveness of lookups, even though the time complexity does not change. For the min and max aggregation operator they implemented another version of the SB-tree, the so called MSB-tree. The mayor tradeoff of this approach is that the size of the increases and therefore incrementing the operation time by a constant factor in comparison to the SB-tree.

Böhlen et al. [3] proposed a completely different attempt to efficiently compute temporal aggregates. Their model named GTA (general temporal aggregation), extends the framework of Klug [12], which provides a two parameter based method to calculate non-temporal aggregation, in order to allow computations of temporal aggregates. The GTA framework clearly separates the partial result relation g that contains a partial result tuple for each data record included in the result relation, from the aggregation groups r_g over which the aggregation functions $F = \{f_1/C_1, \ldots, f_k/C_k\}$ are computed. In contrast, in the traditional approach both aspects are determined by the grouping attributes. A mapping function $\theta : r \to g$ is then used to associate a set of argument tuples, called aggregation group r_g, with a partial result tuple $g_1 \in g$. The extensive flexibility of the GTA model originates from the strict division of the partial result relation g, which only purpose is the grouping of the result relation and the argument relation r. This allows an expression of different forms of aggregation, like ITA, MWTA and STA using a unified general model.

Due to the computational complexity of computing temporal aggregates and the vast increment in size of the amount of tuples to process, in the past years the idea of parallel processing came up. Ye et al. [27] presented in their work two algorithms based on the aggregation tree of Kline and Snodgrass [11]. The two proposed procedures use the divide-and-conquer strategy and are designed for a

shared-memory architecture [21]. The first approach simply partitions the tuples according to their group-by attributes and distributes the resulting partitions among the available cores of the multicore machine. Each thread maintains its own aggregation tree allowing concurrent application of the aggregate operations on data, but forcing each processor to check all tuples in the argument relation. The second algorithm uses only one aggregation tree for each attribute value, shared by all processors. The input relation is partitioned into several sections, subsequently distributed and checked by the single threads of the multicore machine. Due to the fact that all processors use only a shared aggregation tree the update operations have to be executed sequentially.

Gendrano et al. [9] [8] presented in their work another aggregation tree based parallel strategy. They presented five new algorithms based on the master-slave principle and computing temporal aggregates on a shared-nothing architecture. The first approach simply extends the aggregation tree algorithm of Kline and Snodgrass [11] by parallelizing disk I/O. The argument relation is partitioned among the worker nodes, which compute the constant intervals and send them to the central coordinator responsible for constructing the aggregation tree and returning the final result. The second algorithm is a refinement of the first by introducing computational parallelism. Each worker node constructs their own aggregation tree reading part of the input relation and forwards the partial tree to the coordinator which merges them into one result tree. The third solution constructs again partial aggregation trees for each worker node. Then the aggregation values are pushed to the leave nodes of the tree and all intermediate nodes are discarded leading to a local representation of the constant intervals. After evenly partitioning the local tree ('local partition set'), the coordinator receives all leave nodes and equally subdivides the partitions creating a so called 'global partition set'. A fixed partition is assigned to each worker-node, which can now use the broadcasted global information together with the local leave nodes to exchange missing leave-nodes. Afterwards, each worker nodes merges the received leave nodes with the local leave nodes. When all workers are done, the coordinator can produce the result relation by collecting the partial results in sequential order. Another approach presented allows a pairwise merge of the leave nodes of two by the coordinator randomly picked worker-nodes. The temporal aggregation is finished, when all involved worker-nodes have merged their result leading to the result relation. The last variant of the algorithm presented discharges the final collection and merging process returning a distributed set of partial results.

2.2 MapReduce

Using the functional programming construct MapReduce as a programming paradigm to process large raw data in parallel is first presented in 2004 by Dean et al. [5]. Programs implementing this highly scalable model run on large shared-nothing [21] clusters distributing the workload on hundreds or even thousands of connected nodes [25]. Further they are provided with fault tolerance and efficient use of disk space and optimized network transfers [6]. The MapReduce programming model, introduced in [18], is basically defined by the two main methods, map and reduce. The map tasks read the partitioned input file in parallel and completely independent, process the data and generate a set of intermediate key-value pairs. Next, in the so called shuffle-phase, the key-value

pairs are assigned to different reduce-tasks and sorted by the key attributes. The reduce functions receives a subset of the intermediate key-value pairs, having the same key and processes the associated values. The resulting key-value pairs are stored on the disk depending on the user defined code. More detailed information can be found in subsection 2.3.

Another way of benefiting from the MapReduce programming model was explored by Chu et al. [4]. In contradiction to the work of Dean et al. [5] where the MapReduce principle is used to allow a processing of data by a cluster Chu et al. [4] focused on single machine computation working with multicore processors. In addition to the basic methods, they have also in common the idea of splitting the input data and declaring one master, which coordinates map and reduce tasks and multiple workers, which process the assigned work. One of the main differences in this two architectures is the reliability of data exchange. In a cluster, the exchange of data over a network is unreliable and has to be considered in the architectural structure of the framework, whereas in a multicore environment this drawback can be omitted.

Some work was done on comparing the MapReduce approach to parallel DBMSs (e.g. Vertica, DBMS-X) [22] [16], the goal being to understand the differences between the two approaches on performing large-scale data analysis. Both systems provide a high-level programming environment to express the desired tasks. In addition, they state that almost any parallel processing task can be expressed as either a set of database queries or a set of MapReduce jobs. The results of the performance tests clearly stated that both parallel database systems have a significant performance advantage over the MapReduce implementation Hadoop. This substantial gap in the execution times is due to the architectural differences. A parallel DBMS is designed to effectively query large data sets, whereas the main purpose of MapReduce styled systems are complex analytic and ETL tasks.

A combination of a MapReduce based system and a DBMS was presented by Abouzeid et al. [1]. The resulting framework, HadoopDB, replaces the standard storage layer HDFS (Hadoop Distributed File System) used by Hadoop with a DBMS-style optimized storage for structured data. Basically, Hadoop is used to connect multiple single node database systems using the MapReduce framework as a task coordinator and network communication layer. The system preservers the fault tolerance, scalability and flexibility of Hadoop, while taking advantage of the performance and efficiency of parallel databases by pushing most of the query processing inside the database engine.

A successful attempt to outsource the computation to the GPU is presented in the work of [10]. They implemented a modified MapReduce framework called *Mars*, which hides the programming complexity of the GPU behind a MapReduce interface.

2.3 Hadoop

In this subsection we are going to describe in detail the popular MapReduce implementation Hadoop [24] [19]. We provide a general overview of the features and the single stages the framework uses during the processing of the data. The different general techniques together with helpful hints about the customization of the single MapReduce methods in Hadoop and so to trigger the desired behavior of the framework were taken from the books of Donald Miner et al. [13]

and Perera Srinath et al. [17].

HDFS: The Hadoop Distributed File System (HDFS) is a custom file system design to store very large files, having many similarities with other existing distributed file systems. An important difference comparing HDFS to a local filesystem is that the default block size instead of being around 512 B, is by default 64 MB. This decreases the seek time of transferring a large file over the network compared to the transfer time. Further it uses streaming data access patterns, i.e. writing the data once and reading multiple times. Another benefit is that HDFS allows storing a single file over more than one physical media, i.e. a file can be larger than any single disk in the cluster. Further the system provides fault tolerance and increases availability by replicating the blocks to physically separated machines in the cluster.

Nodes: Hadoop uses two basic types of nodes in the cluster, a *Namenode* as master and several *Datanodes* as workers. The filesystem is managed by the Namenode which also maintains the filesystem tree and the metadata for all the files and folders in the tree. Further, the Namenode also knows the Datanodes that store the data blocks of a given file. Due to the fact, that the filesystem HDFS can not be used without a Namenode, an obliteration of the machine running that node would lead to a total information loss. For this reason, it is possible to start a so called *Secondary Namenode* on a physically different machine as a backup that takes over in case of failure.

The Datanodes are the workers in the cluster, which receive assigned data and maintain blocks from the *Namenode* or other *Datanodes*. Further, they periodically notify the Namenode about the locally stored blocks.

In addition, the Namenode runs a *Jobtracker*, which is responsible for coordinating all job executed on the cluster. This is done by scheduling single task to run on the so called *Tasktrackers* executed on every Datanode in the cluster, which continuously report the progress to the Jobtracker. Further, if a task on a Datanode fails, the Jobtracker is responsible for rescheduling the assigned work using a different Tasktrackers and so a different Datanode.

Input File: A Hadoop input file is typically stored in HDFS providing the advantages previously described. Hadoop gives the user the possibility to choose different input-formats, which mainly influence the creation of the *InputSplits* described in the next Subsection. Further the format determines the reading method the so called *RecordReader* subsequently used by the mapper to read the data records from the assigned InputSplit, i.e. reading the file line by line or reading a whole chunk at once. The *InputFormat* can be a few large files, many small files or even an accessible database.

InputSplit: As mentioned before the InputFormat influences the InputSplit, which is a logical (not physical) chunk of the input file. The InputSplit has a defined byte length (default 64 MB, same as the HDFS block size) and can be further divided into single records. Due to the logical split, a InputSplit does not contain the actual data of the input file, but only references to storage locations. The division of the file in smaller parts allows several map-tasks to operate in parallel on the same file. The previously defined InputFormat is also responsible

for guaranteeing the integrity of the data contained in each InputSplit, i.e. a text based InputForamt divides the records in such a way that the lines are not broken.

Mapper: The custom *mapper* can consists of three basic phases, the setup, the mapping and the cleanup. The mapper starts with the customizable setup method, only executed once for each InputSplit processed by the task. As a next step, the selected RecordReader is used to initialize the input parameters of the mapping-function. The mapper uses these parameters to construct and emit intermediate key-value pairs.Once the InputSplit is entirely processed, the cleanup method can for example be used to delete temporal data.

Shuffle: Immediately after the first mapping tasks have completed, the generated intermediate key-value pairs are exchanged and forwarded to the assigned reducers. In this so called shuffle phase the customizable *partitioner* and *comparator* influence the final destination of the intermediate keys. The purpose of the user defined partitioner is to assign to each key-value pair a valid reduce-task taking into consideration information retrieved from the key part. As a sideeffect, key-value pairs having the same key are assigned to the same reducer regardless which mapper is the origin. Further there are two possible application for the comparator, first as a sort-comparator influencing the order of the key-value pair and second as a group-comparator grouping key-value pairs according to a portion of a composite key.

Reduce: The customizable *reducer* receives a subset of intermediate key-value pairs all having either the same key or in case of a composite key having a common key-part. The sorted values are then processed by the reduce-function producing final key-value pairs written to the output file. A reducer-task, similar to the mapper, uses three phases possibly customized by the user, namely the setup, reduce and cleanup-phase.

Output File: The specifiable *OutputFormats* correspond to the InputFormats previously described. Each reducer produces exact one output file stored on the HDFS. The selected format influences the method used to write the file, for example single text lines. If the implemented solution needs more than one MapReduce-run the selected OutputFormat determines the structure of the mapper for the subsequent iteration.

3 Hadoop Temporal Aggregation - Standard

The first approach, the *HTA-Standard* algorithm, needs 3 MapReduce iterations to compute the temporal aggregates. The appendix 'standard' references to to the calculation technique used to determine the result of the aggregation operator. The algorithm considers the fact, that the aggregation value can only change when a new tuple starts or ends. The time instants are the main attributes used during the first iteration in order to group the key-value pairs and assign a reduce-task. As a first step, the amount of starting and ending tuples at each timepoint is determined.

The second iteration partitions the timeline into various segments, called partitions and sorts the time instants within each partition. Then the algorithm goes through all the pairs overwriting the current counter value with the result got by adding the value of the previous pair to the current counter value. At the end of each partition the last counter value is transferred to all consecutive partitions by creating a special so called 'transfer' key-value pair.

The third iteration sums up all values contained in the 'transfer' key-value pairs and adds the result to all timepoints in the partition generating the final result relation.

3.1 Iteration 1: Extract Starting and Ending Timepoints

A detailed graphical illustration of the first iteration, including the data of the running example, can be found in Figure 5.

3.1.1 Map

As a first step, the input file is split into so called InputSplits. These fixed-sized chunks of the input file are distributed across the cluster nodes and processed in parallel by the Hadoop framework.

Definition 1. (Mapper) *Let r be the currently processed tuple read from the assigned InputSplit having scheme $(A_1, \ldots, A_n, T_s, T_e)$, with $r.T_s$ as starting and $r.T_e$ as ending timepoint of the valid time interval $[T_s, T_e)$ with $T_s < T_e$. Then the map-function is defined in the following way:*

$$Mapper(r) \rightarrow \{(r.T_s, \text{'s'})\} \cup \{(r.T_e, \text{'e'})\}$$

The mapper (Figure 3) reads a tuple r, having scheme $[A_1, \ldots, A_n, T_s, T_e]$, form the assigned InputSplit. Afterwards, the timepoint $r.T_s$ together with the character 's', indicating that the included timepoint is a starting point, form a intermediate key-value pair. The ending point of the tuple $r.T_e$ is also used to create a new key-value pair, that the char 'e' used to tag the pair to contain an ending timepoint. Notice, the amount of pairs is two times the total amount of tuples in the input file. Assume the file contains n data records, than the amount of intermediate key-value pairs is $2n$.

Function: `Mapper`$(r[A_1, \ldots, A_n, T_s, T_e])$

Input: The tuple r read from the InputSplit
Output: The intermediate key-value pairs produced from the input tuple r

begin
  `Emit`$(r.T_s$, `'s'`$)$;
  `Emit`$(r.T_e$, `'e'`$)$;

Figure 3: *HTA-Standard*: Mapper (Iteration 1)

Example 1. In the example at Figure 5 a InputSplit consists of the tuples r_1, r_2 and r_3 which are sent to the map-task of node 1. The time interval of tuple r_1 is split into the values 0 and 3 which are then associated with the corresponding char leading to two intermediate key-value pairs $(0, s)$ and $(3, e)$. The tuples r_2 and r_3 are processed using the same methodology and generate the pairs $(0, s), (13, e)$ and $(3, s), (9, e)$ respectively.

3.1.2 Shuffle

As a next step, the default partition function divides the set of intermediate keys into subsets by computing the default hash of the key T, modulo the total numbers of available reducers. Due to the hash function, pairs with the same key end up in the same subset and due to the modulo function the amount of reducers is flexible, while guaranteeing the validity of the assignments. Before processed by the reducer, each subset of intermediate keys is sorted using the default method provided by the framework.

Example 2. Since in the example only 2 nodes and therefore 2 reducers are available, the modulo function used by the default partitioner leads to a separation of even and odd timepoints T, i.e. a even number generates the result 0 and a odd number the result 1.

3.1.3 Reduce

Definition 2. (Reducer) *Let T be the timepoint used as a key during the map phase and $V = \{v_1, \ldots, v_m\}$ be the set of associated values all having T as associated key with the cardinality m. Then the reduce-function is defined as follows:*

$$Reducer(T, V) \rightarrow (T, f_1(V))$$

where the time instant T remains unchanged and the joint value is the result of the following function f_1:

$$f_1(V) = \sum_{\substack{v \in V \wedge \\ v = 's'}} 1 + \sum_{\substack{v \in V \wedge \\ v = 'e'}} (-1)$$

As a next step, the custom reducer function (Figure 4) receives a subset of key-value pairs (T, V), with $V = \{v_1, \ldots, v_m\}$, all having the same time value T. The reducer now goes through all the values in V, updating the counter c according to the char contained in $v_i \in V$. After all values have been processed, an output tuple is created using the unchanged timepoint T and the final counter c. The cardinality of V depends directly on the data in the input file and is within the interval $(1 \leq |V| \leq n)$, where n is the amount of tuples in the input file, i.e. from one tuple starting or ending at the timepoint T to all tuples starting and/or ending at T.

Example 3. The reducer in node 1 now receives all intermediate key-value pairs having a even timepoints sorted by their values. The two pairs $(0, s)$ and $(0, s)$ have the same timepoint value which leads to a single output tuple containing the timeinstant 0 and the determined result 2. This value means that at this timepoint two tuples are starting. The same is done with the other pairs leading to the result that at timepoint 16 another tuple starts and at each timepoint 18 and 20 one tuple ends. Therefore, the following key-value pairs are generated: $(16, 1)$, $(18, -1)$, $(20, -1)$.

Function: `Reducer`$(T, \{v_1, \ldots, v_m\})$

Input: A subset of intermediate key-value pairs with the same timepoint value T
Output: Writes output tuple to the output file

```
begin
    c ← 0;
    foreach v ∈ {v_1, ..., v_m} do
        if v = 's' then
            c ← c + 1
        else
            if v = 'e' then
                c ← c − 1
    Output(T, c);
```

Figure 4: *HTA-Standard*: Reducer (Iteration 1)

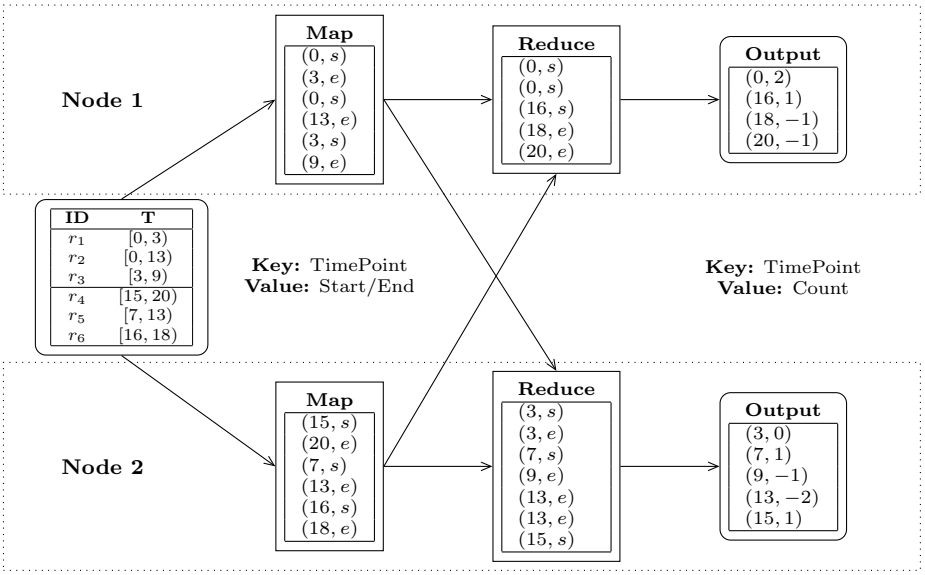


Figure 5: Graph of *HTA-Standard* Algorithm (Iteration 1)

3.2 Iteration 2: Partition Timepoints & Generate Transfer Values

3.2.1 Map

Before the second iteration, graphically illustrated in Figure 10, the partitions file (subsubsection 6.1.1) is produced. Assume that the file is successfully created and is available to all Datanodes in the cluster.

Definition 3. (Mappe) *Let T be the timepoint and c be the counter value of the previously iteration. Further assume, that $P[T_s, T_e]$ is the partition to which the currently processed time instant T belongs, having $[T_s, T_e]$ as associated time interval and that the total amount of partitions is equal to p. Then the map-function of the second iteration is defined as follows:*

$$Mapper(T, c) \rightarrow \{([P, T, p], c\}$$

where the current partition $P \in \mathcal{P}$ and $P[T_s, T_e]$ encloses the current timepoint T such that $P.T_s \leq T < P.T_e$.

The iteration starts with the map-function shown in Figure 6. The key-value pairs (T, c) generated by the first iteration are read from the file separating again the timestamp T from the counter c. As a next step the partitions file is used to retrieve the partition $P[T_s, T_e]$ to which the timestamp T belongs, fulfilling the condition $P.T_s \leq T < P.T_e$. Then a new pair is formed using the unique identification number of the partition P, the current timestamp T and the total amount of partitions p as key and the unchanged counter c as value. Notice, the amount of key-value pairs remains the same, only the amount of data stored in the key is augmented.

Function: `Mapper`(T, c)
Input: The tuples read from the output file of the 1. iteration
Output: The intermediate key-value pairs produced from the input tuple (T, c)
Let $\mathcal{P}$ be the set of partitions stored in the partitions file;
Let p be the total number of partitions;
begin
    $P \in \mathcal{P}$ with $P_r.T_s \leq r.T < P_r.T_e$;
    `Emit`$([P, T, p], c)$;

Figure 6: *HTA-Standard*: Mapper (Iteration 2)

Example 4. For the example assume that the output file generated by node 1 during the first iteration, containing tuples of form (T, c), is again been processed by the same node. The partition, shown in Figure 7, of the single timeinstants is looked up leading to the result that timepoint 0 belongs to partition $P1$ and the points $16, 18, 20$ to partition $P4$. Further we can see that the total number of partitions p is equal to 4. This information is used to form the following key: $[1, 0, 4], [4, 16, 4], [4, 18, 4], [4, 20, 4]$. At the end the intermediate key-value pairs are formed combining the key with the respective counter values c: $([1, 0, 4], 2), ([4, 16, 4], 1), ([4, 18, 4], -1), ([4, 20, 4], -1)$.

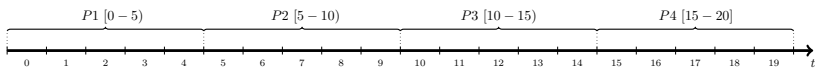


Figure 7: Temporal Partitions of Example

3.2.2 Shuffle

The second iteration uses a custom partition function (Figure 8), because while forming the subsets of the intermediate keys the function cannot consider the whole key but only the assigned partition number P, i.e. if they have the same partition number they end up in the same subset regardless the different timestamp T. This means that all key-value pairs having the same partition number are assigned to the same reduce-task. Another important fact that has to be considered by the partition function is that the reducer processing a certain partition can not be arbitrarily selected by the framework. In order to divide the final aggregation relation into consecutive output files, successive partitions

are assigned either to the same reducer or if available to the chronological next reducer. Notice, the first partition number is 1, but Hadoop numerates the available reducers starting from 0, which forces the decrement by 1 during the calculations of the partition function. The partitioner calculates the amount of temporal partitions that have to be assigned to each of the reduce tasks in order to evenly distribute the workload without introducing the error. This $offset$ is calculated by dividing the total amount of partitions p, stored in the key part of each intermediate key, by the total amount of available reduce-tasks R. Now the unique partition number of the currently processed key-value pair is divided by the determined $offset$, getting a valid reducer-task number num. In the case that $p = R$ meaning that $\frac{p}{R} = 1$, each reducer receives exactly one partition. In the case that $\frac{p}{R} > 1$ the single partitions are distributed among the available reducer-tasks.

The default behavior of Hadoop is to group the key-value pairs according to the key attribute. Since this algorithm uses a composite key, a custom group-comparator is used in order to ensure that the pairs are grouped solely by the partition number P. The effect of this method is that all keys with different timestamps T, but the same partition number P are gathered together leading to a subset of the following form: $[P, \{T_1, \ldots, T_m\}, p], \{v_1, \ldots, v_m\}$. An additional sideeffect of the comparator is that the timepoints T are sorted from earlier to lather.

Function: `Partitioner`$([P, T, p], R)$
Input: The key of the current key-value pair and the total number of reducers
Output: The assigned reducer number
begin
  $offset \leftarrow \frac{p}{R}$;
  $num \leftarrow \lfloor \frac{(P-1)}{offset} \rfloor$;
  return num;

Figure 8: *HTA-Standard*: Partitioner (Iteration 2)

Example 5. Due to the fact that in the example we use 2 reducers and have a total amount of partitions $p = 4$, the calculated offset is as follows: $\frac{p}{R} = \frac{4}{2} = 2$. Next, the partition number of the first pair is divided by the offset $\lfloor \frac{(P1-1)}{2} \rfloor = 0$ assigning it to the reducer of node 1. The same is done for the next key-value pair with the following result $\lfloor \frac{(P-1)}{offset} \rfloor = \lfloor \frac{(P4-1)}{2} \rfloor = 1$ sending it to node 2. The partitioner continuous until all pairs have been processed. At the end, this leads to a partitioning of the intermediate key-value pairs where node 1 receives $P1$ and $P2$ and node 2 gets $P3$ and $P4$. Further the group-comparator groups the timepoints according to the assigned partition number creating the sets $[(1, \{0, 3\}, 4], \{2, 0\})]$ and $[(2, \{7, 9\}, 4], \{1, -1\})]$ for node 1.

3.2.3 Reduce

Definition 4. (Reducer) *Let P be the identification number of the currently processed partition and the set $\{T_1, \ldots, T_m\}$ the set of timestamps all having the same P. Further assume that p is the total number of partitions and $V = \{v_1, \ldots, v_m\}$ is the set of values related to the key. Then the reduce-function is*

defined as follows:

$$Reducer([P, \{T_1, \ldots, T_m\}, p], V) \rightarrow$$
$$\bigcup_{i=1}^{m}([P, T_i, p], c_i) \cup \bigcup_{l=P+1}^{p}([l, -1, p], c_m)$$

where $T_i \in \{T_1, \ldots, T_m\}$ *and the counter* $c_i = c_{i-1} + v_i \in V$ *with* $c_0 = 0$.

The reduce-task (Figure 9) receives the sorted subset of the intermediate keys and starts processing the contained key-value pairs. Since all keys in the subset are grouped by the partition number $P.N$ and the total amount of partition p is globally the same, both variables remain constant within each subset. As a first step the function initializes the counter c_{prev} to 0 and adds the variable c_{prev} to the counter c_i, not modifying its value. The counter c_i, together with the key-part consisting of the partition number $P.N$, the timestamp T_i and the total amount of partitions p, form the key-value pair written to the output file. Then the counter c_i is maintained by overwriting $c_{prev} \leftarrow c_i$. In the following iterations of the loop the counter value c_{prev} is added to the current value v_i and a new output tuple is written to the file. Once the tuple was written the current counter c becomes c_{prev} and the next key-value pair is processed until the end of the subset is reached i.e. $i = m$, where $m = |T| = |V|$ is the cardinality of the subset of timestamps.

After all the timestamps of one partition are processed, the reducer generates some additional so called 'transfer' key-value pairs. An additional pair has to be constructed for every subsequent partition $P = P+1$, until the upper bound p is reached. So the partition number $P + 1 = l$ together with the timestamp -1, the total amount of partitions p and the last counter value c_m form a new transfer key-value pair. The algorithm continuous until the last transfer tuple containing p as partition number is written to the output file. The amount of additional generated tuples depends directly from the total number of partitions used by the algorithm. Since the amount of partitions p is known the upper bound of the total number of transfer keys generated by all available reducers is equal to $p(p-1) = \mathcal{O}(p^2)$.

Function: `Reducer`$([P, \{T_1, \ldots, T_m\}, p_{tot}], \{v_1, \ldots, v_m\})$
Input: A subset of intermediate key-value pairs with the same partition number P
Output: Writes output tuple to the output file

```
begin
    c_prev ← 0;
    for i = 1 to m do
        c ← v_i + c_prev;
        Output([P, T_i, p], c);
        c_prev ← c;
    if c ≠ 0 then
        l ← P + 1;
        while l ≤ p do
            Output([l, −1, p], c);
            l ← l + 1;
```

Figure 9: *HTA-Standard*: Reducer (Iteration 2)

Example 6. In the example Node 1 starts by processing the first subset $[(P1, \{0, 3\}, 4], \{2, 0\})]$. The reducer uses the partition number $P1$, the first timestamp $T_1 = 0$, the total amount of partitions $p = 4$ and the unchanged counter $c_i = 2$ to form the first output tuple: $([P1, 0, 4], 2)$. Next, the counter $c_i = 2$ is maintained and the next timepoint and associated value is processed. This leads to the following output tuple: $([P1, 3, 4], 2)$, where 3 is the new timepoint T_i and $c_i = v_i + c_{prev} = 0 + 2 = 2$ is the result of the previous state of the counter $c_{prev} = 2$ and the current counter value $v_i = 0$. The algorithm continuous until all timeinstants contained in the subset are processed and due to the non-zero value of the last counter some transfer keys have to be generated. The first transfer key consists of the consecutive partition of $P1$ which is $P1 + 1 = P2$, the timestamp -1, the total amount of partitions p and as value the last status of the counter $c_m = 2$. The following transfer pairs contain the same values except the partition number which is continuously increased by 1 until it reaches the total amount of partitions $P4 = p$. Then the next subset is processed until all assigned work is done.

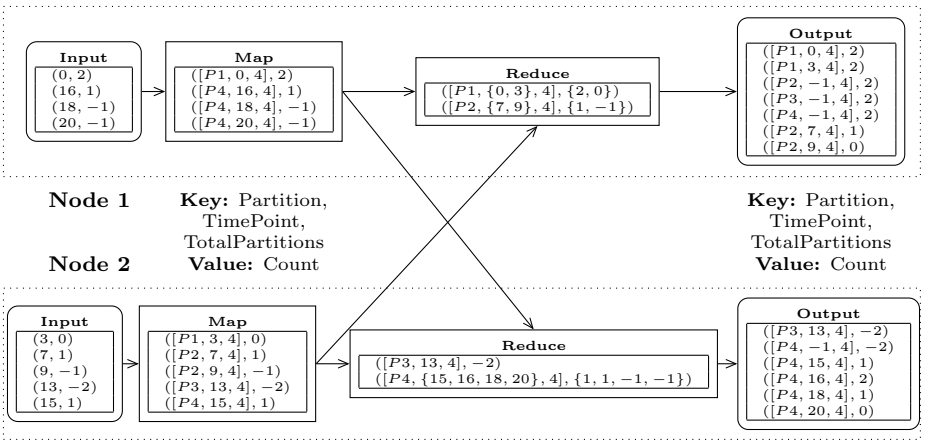


Figure 10: Graph of *HTA-Standard* Algorithm (Iteration 2)

3.3 Iteration 3: Integrate Transfer Values

The graphical illustration including the example relation of the final iteration is shown in Figure 13. As a first step, the output file of the previous iteration is split into the single InputSplits and read by the map-function.

3.3.1 Map & Shuffle

Definition 5. (Mapper) *Let P be the identification number of the partition, T be the timestamp, p be the total amount of partitions and c be counter determined by the previous MapReduce iteration. Then the map-function is defined as follows:*

$$Mapper([P, T, p], c) \rightarrow \{([P, T, p], c\}$$

The mapper (Figure 11) does not change any values, it only reads and emits the key-value pairs and forwards the data to the partition-function. Due to the unchanged structure of the key-value pairs, the same principles for the partitioner (Figure 8) and the group comparator as in the previous iteration are reused. This means, that the set of key-value pair is again split into subsets of equal partition number P and sorted by timestamp T. Since all transfer key contain a negative timestamp the sorting phase causes them to be on the top of the list of keys having the same partition number P.

Function: `Mapper`$([P, T, p], c)$

Input: The tuple read from the output file of the 2. iteration
Output: The intermediate key-value pairs produced from the input tuple

begin
  `Emit`$([P, T, p], c)$;

Figure 11: *HTA-Standard*: Mapper (Iteration 3)

Example 7. The Mapper reads the tuples one by one and restores the key-value pair structure. The partition function is the same as in the previous iteration, which due to the same partition numbers calculates the same offset and assigns the same partition number to the same reducers. This means, that $P1$ and $P2$ are processed by node 1, whereas $P3$ and $P4$ by node 2. During the shuffle phase the group comparator groups the pairs according to the partition number creating the subset $[(P1, \{0, 3\}, 4], \{2, 2\})]$ for $P1$ and $[(P2, \{-1, 7, 9\}, 4], \{2, 1, 0\})]$ for $P2$. We can see that the transfer key-value pair was moved at the beginning of the subset because of the negative timepoint $T_i = -1$.

3.3.2 Reduce

Definition 6. (Reducer) *Let $T = \{T_1, \ldots, T_k, T_{k+1}, \ldots, T_m\}$ be the set of timestamps all having the same partition number P, p be the total number of partitions and $V = \{v_1, \ldots, v_k, v_{k+1}, \ldots, v_m\}$ be the set of values associated with the timestamps in T. The subset $T' = \{T_1, \ldots, T_k\}$ contains k negative timestamps and the subset $V' = \{v_1, \ldots, v_k\}$ is composed of the k values associated with the the k negative timestamps. Then the reduce-function is defined as follows:*

$$Reducer([P, T, p], V) \rightarrow$$
$$\{[P.T_s, T_{k+1}], inc\} \cup \bigcup_{i=k+1}^{m} ([T_i, T_{i+1}], inc + v_i) \cup \{[T_m, P.T_e], inc + v_m\}$$

where $inc + v_i$ is the result of the temporal aggregation valid for the time interval $[T_i, T_{i+1}]$ with inc being the result of f_2 defined as follows:

$$inc = f_2(V') = \sum_{i=1}^{k} v_i$$

The subset of intermediate keys is now received by the reduce-function (Figure 12), which calculates the final value of the aggregation and their valid time-intervals. As a first step, the reduce-task initializes the time-interval t consisting

of a start-timestamp T_s and an end-timestamp T_e, the counter c, the increment inc and a boolean variable used to handle the special case occurring when the first key-value pair is processed. Then, the reducer starts processing the timestamps T_i. Due to the contained negative timepoint, all additionally generated transfer keys are located at the beginning of each subset, allowing the function to sum up the related values v_i, as long as the timestamp T_i remains negative. Along with the sum stored in the variable inc the boolean variable 'first' is set to true. This boolean variable is used to recognize the first key-value pairs not being a transfer key. Once the transfer keys are processed and the timestamps T_i starts to be positive, the computation of the aggregation values and the building of the related valid time-intervals can start. As a next step, it is checked if the current timestamp T_i is larger as the starting point of the timeinterval $t.T_s$. Since this is the first iteration of the loop and the interval t still contains the initialization value, the check will be evaluated to be true. The next test involves the boolean variable 'first'. If the variable is true, meaning that there were transfer keys at the beginning of this partition, the startpoint of the interval $t.T_s$ is overwritten with the startpoint of the current partition $P.T_s$. Otherwise, if the boolean variable is false, meaning that no data tuples are entering the current partition, nothing has to be done and the part is skipped. The background of this special case is that, if there are no tuples entering the partition the first encountered timestamp is the start of the aggregation relation, otherwise the result relation starts at the beginning of the partition. Assuming that at the beginning of the reducer some transfer-keys were processed, the startpoint of the time interval $t.T_s$ becomes equal to $P.T_s$. Next, a check is done testing if the sum of the counter c and the increment inc is different than 0. If not, no output tuple has to be produced. Assuming that the sum is different than 0, the current timestamp T_i becomes the endpoint of the timeinterval t and a new output tuple is written to the file consisting of the timeinterval t and the aggregation result $(c + inc)$. Once the tuple is written, the current timestamp T_i previously used as endpoint of the interval, becomes the starting point of the next interval. Further, the value v_i associated with the timestamp previously outputted is maintained using the counter c. In the second iteration of the loop a repetition of the same timestamp value causes only a update of the counter c using the current value v_i. A timestamp bigger than the previous one causes the previously described procedure leading to a generated output tuple.

As soon as all timestamps T_i contained in the subset are processed, a final output tuples is produced in the case the last written time interval t has a smaller ending point than the endpoint of the partition $P.T_e$ and the last aggregation value $(c + inc)$ is different from 0. This additional key-value pair extends the time-interval of the last output tuple in order to reach the ending timestamp of the partition $P.T_e$. Notice, this does not corrupt the result relation, this is needed because the endpoint of the partition is not present as a key-value pair in the assigned subset. The problem could also be fixed by generating some additional key-value pairs marking the endpoint of the currently processed partition. The total amount of output tuples is equal to the number of different timestamps excluding the transfer keys, plus up to one tuple per partition generated in the last special case previously mentioned.

Example 8. The reducer of node 1 starts by processing the subset $[(P1, \{0, 3\}, 4], \{2, 2\})]$ with the partition number $P1$. We can see that there

```
Function: Reducer([P, {T_1, ..., T_m}, p], {v_1, ..., v_m})
Input: A set of IKV-Pairs with the same assigned Partition Number
Output: Writes KV-Pairs to the Output File
begin
    t[T_s, T_e] ← [0, 0];
    c ← 0;
    inc ← 0;
    first ← false;
    for i = 1 to m do
        if T_i < 0 then
            inc ← v_i;
            first ← true;
        else
            if T_i > t.T_s then
                if first then
                    t.T_s ← P.T_s;
                    first ← false;
                if (c + inc) ≠ 0 then
                    t.T_e ← T_i;
                    Output(t, (c + inc));
                t.T_s ← T_i;
                c ← v_i;
            else
                c ← v_i;
    if (c + inc) ≠ 0 ∧ t.T_e < P.T_e then
        t.T_e ← P.T_e;
        Output(t, (c + inc));
```

Figure 12: *HTA-Standard*: Reducer (Iteration 3)

are no contained transfer keys, which enables us to write the first output tuple by simply connecting the two timestamps $t = [0, 3)$ and associate them with the aggregation value $(c+inc) = (2+0) = 2$. The algorithm continuous, because the endpoint is smaller than the partition endpoint, i.e. $t.T_e < P.T_e = 3 < 5$ and $(c+inc) \neq 0$. Therefore an additional tuple is created containing the endpoint of the partition $P1$, i.e. $([3, 5), 2)$. Next the subsequent partition $P2$ is processed having $[(P2, \{-1, 7, 9\}, 4], \{2, 1, 0\})]$ as subset. At the beginning of the partition a transfer key is present leading to $inc = 2$ and 'first' $= true$. Therefore, the first output tuple contains as starting point of the interval t the partition start point $t.T_e = P.T_s = 5$, the endpoint $t.T_s = T_i = 7$ and the aggregation value $(c+inc) = (0+2) = 2$. The second tuple written to the output file is $([7, 9), 3)$, where the aggregation result is determined by $(c + inc) = (1 + 2) = 3$. As a last step, the algorithm determines that $t.T_e < P.T_e = 9 < 10$ and $(c+inc) \neq 0$ leading to a last additional output tuple $([9, 10), 2)$.

4 Hadoop Temporal Aggregation - Merge

The second developed algorithm, *HTA-Merge*, is design in a way that no initialization phase is required, omitting all parameters such as min, max, count and mode needed by the other two approaches. Moreover, the algorithm has no need for the partitions file, saving computation time and storage space. In order to be able to process the input tuples without partitioning the timeline the algorithm uses a variable amount of iterations. The exact amount depends on the number

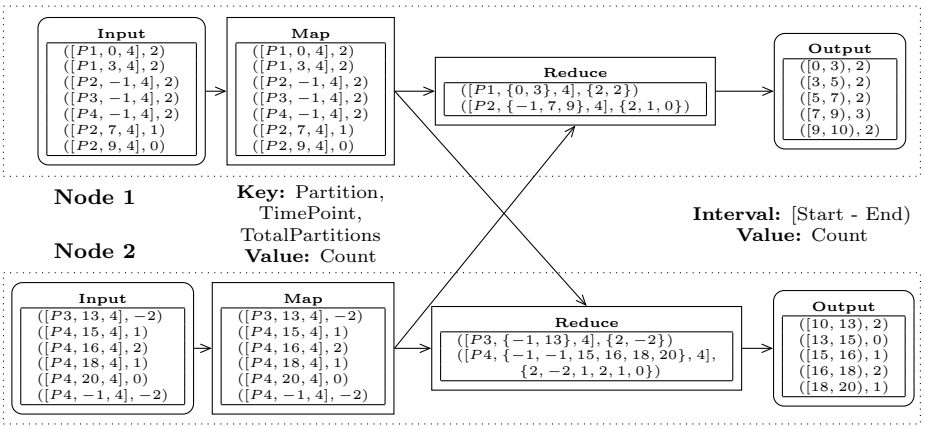


Figure 13: Graph of *HTA-Standard* Algorithm (Iteration 3)

of reduce-tasks and is within the interval $[2 \dots \log_2(O_2)]$, where O_2 is the number of output-files created by the second iteration of the algorithm. This means as the number of reducers increases, the workload of the single reduce-functions decreases, while the amount of repetitions of the third iteration increases. The first iteration uses as main reference the offset within the input file to partition the key-value pairs and distribute them among the available reduce-tasks. The result of the reduce-function is a partial aggregation relation involving only a chunk of the input file. Notice that at this point there can be multiple partial results in each produced output file. The second iteration uses the unique name of the input file in order to merge all fractions of the result relation contained in each input file to one result relation per file. The third iteration has to merge the partial results spread over the previously produced output files into a single result file. Therefore, the algorithm uses a common key for two consecutive files to force all contained tuples to be processed by the same reduce-task leading to a merged output file. The iteration is called as long as the previous run created more than one output file. This is done by forcing the Hadoop framework to reduce the number of reduce-tasks by half after every iteration leading to a logarithmic ($\log_2$) decrement in the number of partial output files. In other words, we iteratively merge two consecutive files until only one output file remains.

4.1 Iteration 1: Process Chunks of Input File

The graphical illustration of the first iteration, including the processing of the example relation, can be found in Figure 18. As a first step, the input file is split into 64 MB chunks, which are assigned to the different map-tasks. Each of those InputSplit has a certain byte-offset within the input file, which can be retrieved by the mapper.

4.1.1 Map

Definition 7. (Mapper) *Let r be the currently processed tuple read from the assigned input split having scheme $(A_1, \dots, A_n, T_s, T_e)$, where $r.T_s$ is the starting and $r.T_e$ is the ending timepoint of the valid time interval $[T_s, T_e)$ with $T_s < T_e$.*

Further assume that o is the byte-offset within the input file determined by the Hadoop-framework. Then the map-function is defined in the following way:

$$Mapper(r) \rightarrow \{([o, r.T_s], \text{'s'})\} \cup \{([o, r.T_e], \text{'e'})\}$$

The first iteration starts with a custom map-task (Figure 14) reading the tuples r from the assigned InputSplit having a certain byte offset o within the file. Each input tuple has the scheme $[A_1, \ldots, A_n, T_s, T_e]$ where A_i is an non temporal attribute and $r.T_s$ and $r.T_e$ are the starting and ending points of the valid time interval. This time-interval is split into its starting timestamp $r.T_s$ and ending timestamp $r.T_e$ and the byte-offset o is retrieved from the system. A new key-value pair is generated using the offset o and the timestamp $r.T_s$ as key and the char 's' as value, marking the key to contain a starting point. The same procedure is applied to the other timepoint $r.T_e$, except the char in the value part is a 'e'. In total the mapper, having as input n tuples, creates $2n$ intermediate key-value pairs.

Function: `Mapper`$(r[A_1, \ldots, A_n, T_s, T_e])$
Input: The tuple r read from the InputSplit
Output: The intermediate key-value pairs produced from the input tuples r
Let o be the byte-offset of the assigned InputSplit in the Input File;
begin
    `Emit`$([o, r.T_s]$, `'s'`$)$;
    `Emit`$([o, r.T_e]$, `'e'`$)$;

Figure 14: *HTA-Merge*: Mapper (Iteration 1)

Example 9. For the example assume that r_1 and r_2 have an offset of 0 and r_3 and r_4 an offset of 64 respectively. Further assume that the mapper of node 1 receives the tuples r_1 to r_4 and the remaining records r_5 and r_6, having an offset of 128 are processed by node 2. The mapper receives r_1 with the interval $[0, 3)$, separates the timepoints $T_s = 0$ and $T_e = 3$ and generates the following two key-value pairs: $([0, 0], s)$ and $([0, 3], e)$. Tuple r_2 is processed in the same. The records r_3 and r_4 have a different offset $o = 64$, leading to the following intermediate key-value pairs: $([64, 3], s)$, $([64, 9], e)$, $([64, 15], s)$ and $([64, 20], e)$.

4.1.2 Shuffle

The *HTA-Merge* algorithm uses a custom partitioner (Figure 15) very similar to the default one implemented by Hadoop. The main difference is that only the hash of the byte offset o, modulo the total amount of available partition-tasks R is used in the assignment of a valid reduce task. The partitioner assigns pairs having the same offset o to the same reducer and contemporaneously evenly distribute the intermediate keys among all reducers, adjusting the individual workload. The custom group-comparator groups the available key-value pairs, consisting of the composite key $[o, T]$, by byte-offset o and sorts the grouped timestamps T in increasing order. This leads to subsets of intermediate keys of the form $([o, \{T_1, \ldots, T_m\}], \{v_1, \ldots, v_m\})$.

Example 10. In our example we have three different offsets which are 0, 64 and 128 and two available reducers. Therefore one reducer has to process two

Function: `Partitioner`$([o, T], R)$

Input: The key of the current key-value pair and the total number of reducers
Output: The assigned reducer number

begin
    $num \leftarrow hash(o) \bmod R$;
    return num;

Figure 15: *HTA-Merge*: Partitioner (Iteration 1)

subsets of intermediate key-value pairs, whereas the other reducer receives only one subset. We assume that the partition function assigns pairs having a offset $o = 0$ and $o = 64$ to node 1 and the remaining to node 2. The now applied group-comparator groups the pairs according to their contained offset leading to the following first group: $([0, \{0, 0, 3, 13\}], \{s, s, e, e\})$. The set $\{0, 0, 3, 13\}$ contains the sorted timepoints all having $o = 0$ and the set $\{s, s, e, e\}$ contains the associated values v_i. The same is done for all intermediate key value pairs in the system.

4.1.3 Reduce

Definition 8. (Reducer) *Let $T = \{T_1, \ldots, T_m\}$ be the set of timestamps all having the same byte-offset o and $V = \{v_1, \ldots, v_m\}$ be the set of values associated with the timestamps in T. Further let $T' = \{T_j, T_{j+1} \ldots, T_{j+l}\} \subset T$ and $|T'| = l$ be a maximum sequence of consecutive identical timepoints, i.e. $T_i = T_{i+1}$ for $i = j, \ldots, (j+l)$. The subset $V' = \{v_j, v_{j+1} \ldots, v_{j+l}\}$ is the set of values associated with T' having the same cardinality l. Then the reduce-function is defined as follows:*

$$Reducer([o, T], V) \rightarrow \bigcup_{i=1}^{m-1} ([T_i, T_{i+1}], c_i))$$

where c_i is the value returned by the function f_3 with $c_0 = 0$.

$$c_i = f_3(V') = \sum_{\substack{v \in V' \wedge \\ v = \text{'s'}}} (c + 1) + \sum_{\substack{v \in V' \wedge \\ v = \text{'e'}}} (c - 1)$$

At the end of the first iteration the reducer-function, shown in Figure 17 is called. The task starts by receiving a subset of intermediate key-value pairs all having the same byte-offset o and all timestamps $T_i \in T$ sorted in increasing order. As a first step, the counter c and the time-interval t having a start-timestamp $t.T_s$ and an end-timestamp $t.T_e$ are initialized. Then a loop starts processing all timestamps $T_i \in T$ and their associated values $v_i \in V$. As a next step, a check is done testing if the current timestamp T_i is greater than the initialization value of the starting point of the interval $t.T_s$. Assuming that the timestamp is bigger than 0, the currently processed timepoint T_i becomes the ending point of the interval $t.T_e$ and a new output tuple is created containing t and the counter c as aggregation value. Once the tuple is written to the file, the previous ending point of the interval t becomes the starting point and the counter c is updated using the `UpdateCounter`$(v_i, c, 1)$ (Figure 16) function. This function unifies the counter c and the increment inc using the mathematical

method determined by v, i.e. if the value v is equal to 's' the counter c and the increment inc are added, otherwise if the value v is equal to 'e', they are subtracted. In the second iteration of the loop the new timestamp, assumed to have the same value as the previously processed, triggers only an update of the counter c without generating a new output tuple. Instead a bigger timestamp T_i causes the function to follow the previously explained steps producing a new output tuple with interval $t = [T_{i-1}, T_i)$ and value c.

```
Function: UpdateCounter(v, c, inc)
begin
    if v = 's' then
        return (c + inc)
    if v = 'e' then
        return (c − inc)
```

Figure 16: Update Counter Function

```
Function: Reducer([o, {T_1, ..., T_m}], {v_1, ..., v_m})
Input: A subset of intermediate key-value pairs with the same byte-offset o
Output: Writes output tuple to the output file
begin
    c ← 0;
    t[T_s, T_e] ← [0, 0];
    for i = 1 to m do
        if T_i > t.T_s then
            if c ≠ 0 then
                t.T_e ← T_i;
                Output(t, c);
            t.T_s ← T_i;
            c ← UpdateCounter(v_i, c, 1);
        else
            c ← UpdateCounter(v_i, c, 1);
```

Figure 17: *HTA-Merge*: Reducer (Iteration 1)

Example 11. We assume that the reducer receives the following subset of intermediate key-value pairs: $([0, \{0, 0, 3, 13\}], \{s, s, e, e\})$. Since T_1 and T_2 are both 0 the reducer only has to update the counter c without writing any output tuple, which leads to $c = 2$ because the associated values of the timestamps are 's,s'. The next timepoint $T_3 = 3$ forms the timeinterval $t = [0, 3)$ and produces a new tuple $(t, c) = ([0, 3), 2)$ written to the output file. Next, T_3 becomes the starting point of the interval and the counter c is updated using the following parameters: $v_i = e, c = 2$, leading to a decrement of the counter $c = 1$. The last timestamp $T_4 = 13$ of the first subset being bigger than $T_3 = 3$ causes the same steps as before generating the output tuple $(t, c) = ([3, 13), 1)$. The reducer of node 1 continuous with the next assigned subset if any.

4.2 Iteration 2: Merge Partial Result within Single Files

As previously mentioned, the second iteration of the algorithm reads the single output files and merges the multiple partial temporal aggregation results to

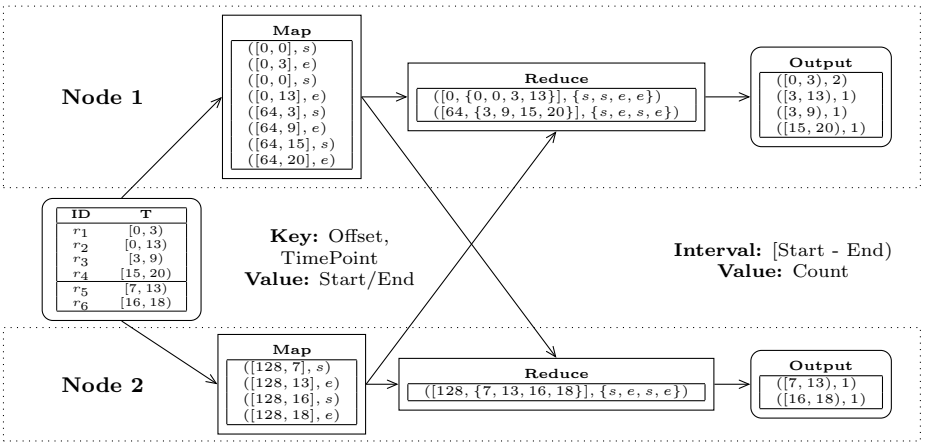


Figure 18: Graph of *HTA-Merge* Algorithm (Iteration 1)

one partial result per file. The main reference of the iteration is the unique identification number of the input file, from which the tuples are retrieved. This guarantees that key-value pairs formed from the tuples of the same input file are assigned to the same reduce-task. The detailed process of the second iteration is shown in the graph in Figure 22.

4.2.1 Map

Definition 9. (Mapper) *Let id be a unique identification number generated using the name of the input file and let* $[T_s, T_e]$ *be the time interval produced by the previous iteration with* $T_s < T_e$*. Further assume that c is the previously calculated aggregation result valid during the associated time-interval. Then the map-function is defined in the following way:*

$$Mapper([T_s, T_e], c) \rightarrow \{([id, T_s, c], \text{'s'})\} \cup \{([id, T_e, c], \text{'e'})\}$$

As a first step, the map-function (Figure 19) reads a tuple r from the assigned InpuSplit of the input file. Then, an unique identification number *id* is created, using the filename of the input file indicated as source, as main reference. This means, that map-tasks reading from different input files generate different identification numbers, whereas map-tasks processing InputSplits from the same input file generate the same *id* number. As a last step, two intermediate key-value pairs are formed using the *id*, the timestamp T_s and T_e respectively and the counter c as key. The character forming the value of the pair is determined by the type of timestamp used in the key, meaning that a starting timestamp is associated with the char 's' and an ending timestamp with 'e'.

Example 12. In our example, the mapper starts by reading the first tuple $([T_s, T_e], c) = ([0, 3), 2)$ created by the previous iteration. We assume that the identification number generated by the mapper is $id = In1$. As a next step, the time interval is split into the timepoints $T_s = 0$ and $T_e = 3$. The identification number $id = In1$, the timestamps $T_s = 0$ and $T_e = 3$ and the counter value

Function: `Mapper`$([T_s, T_e], c)$)

Input: The tuples read from the output file of the 1. iteration
Output: The intermediate key-value pairs produced from the input tuple $([T_s, T_e], c)$

begin
    Let id be the unique identifier of the Input File;
    `Emit`$([id, T_s, c]$, `'s'`$)$;
    `Emit`$([id, T_e, c]$, `'e'`$)$;

Figure 19: *HTA-Merge*: Mapper (Iteration 2)

$c = 2$ form the key and the respective chars 's' and 'e' the value of the two produced key-value pairs: $([In1, 0, 2], s)$ and $([In1, 3, 2], e)$. The other tuples contained in the InputSplit of node 1 are processed in an analogous manner.

4.2.2 Shuffle

The partitioner, shown in Figure 20, assigns valid reducers to the single intermediate key-value pairs, only considering the unique identification number id. The function does not use the hash-value of the id number, because every reduce task has to process only one subset of intermediate keys. Notice, the case that there are more different id numbers than available reduce tasks is not possible, because the amount of reducers remains constant until the end of this iteration. The partitioner calculates the reducer number num by using the formula: $id \bmod R$. This function is the key part of the second iteration, because it forces the intermediate keys from the same file to be assigned to the same reducer, merging the partial results within each file. The group-comparator groups the set of key-value pairs by identification number id and the timestamp are sorted by increasing values. Notice, additionally to the subset of timestamps T and the corresponding subset of values V a new subset is contained in each group of key-value pairs, namely the subset of associated counter values C with $|C| = |T| = |V|$. This leads to the following general structure of the subset assigned to a reduce task: $([id, \{T_1, \ldots, T_m\}, \{c_1, \ldots, c_m\}], \{v_1, \ldots, v_m\})$.

Function: `Partitioner`$([id, T, c], R)$

Input: The key of the current key-value pair and the total number of reducers
Output: The assigned reducer number

begin
    $num \leftarrow id \bmod R$;
    return num;

Figure 20: *HTA-Merge*: Partitioner (Iteration 2)

Example 13. We assume that the partition-function is evaluated as follows: $num = id \bmod R = id1 \bmod 2 = 0$, assigning all key-value pairs with a $id = id1$ to the reducer of the node 1. This subset of intermediate key-value pairs having a common id number is grouped by the group-comparator in the following way: $([id, T, C], V)$, where $id = id1$, $T = \{0, 3, 3, 3, 9, 13, 15, 20\}$, $C = \{2, 2, 1, 1, 1, 1, 1, 1\}$ and $V = \{s, e, s, s, e, e, s, e\}$. We can see, that for example the first key-value pair emitted by the mapper has been split up and distributed

over the different subsets: $([In1, 0, 2], s) \Rightarrow id = In1, T_1 = 0, c_1 = 2, v_1 = s$. The same procedure is applied to all other key-value pairs in the system.

4.2.3 Reduce

Definition 10. (Reducer) *Let id be the unique identification number of the source-input file and the time-interval $T = \{T_1, \ldots, T_m\}$ the set of timestamps all having the same id. Further assume that $C = \{c_1, \ldots, c_m\}$ is the set of previously calculated aggregation values and $V = \{v_1, \ldots, v_m\}$ is the set of chars associated with the timestamps in T. Further let $T' = \{T_j, T_{j+1} \ldots, T_{j+l}\} \subset T$ and $|T'| = l$ be a maximum sequence of consecutive identical timepoints, i.e. $T_i = T_{i+1}$ for $i = j, \ldots, (j + l)$. The subset $V' = \{v_j, v_{j+1} \ldots, v_{j+l}\}$ and $C' = \{c_j, c_{j+1} \ldots, c_{j+l}\}$ are the respective set of values and counter values associated with T' having the same cardinality l. Then the reduce function is defined as follows:*

$$Reducer([id, T, C], V) \rightarrow \bigcup_{i=1}^{m-1} ([T_i, T_{i+1}], a_i)$$

where a_i is the value returned by the function f_4 with $a_0 = 0$.

$$a_i = f_4(V', a_{i-1}, C') = \sum_{\substack{v \in V' \wedge \\ v = \text{'s'} \wedge \\ c \in C'}} (a_{i-1} + c) + \sum_{\substack{v \in V' \wedge \\ v = \text{'e'} \wedge \\ c \in C'}} (a_{i-1} - c)$$

The user defined reduce-function shown in Figure 21 receives the assigned subset of the intermediate keys all having the same identification number *id* and therefore having the same source file. The construction of the valid time interval t and the calculation of the related partial aggregation value c is done in the same way as in the previous iteration of the algorithm. However, the function `UpdateCounter`(v_i, c, c_i) used to update the counter c, while having a fixed increment of 1 in the first iteration, now has to consider a variable length of the increment parameter. The amount to increment depends on the counter value c_i, which is associated with the currently processed timestamp T_i.

Function: `Reducer`$([id, \{T_1, \ldots, T_m\}, \{c_1, \ldots, c_m\}], \{v_1, \ldots, v_m\})$

Input: A subset of intermediate key-value pairs with the same identification number *id*
Output: Writes output tuple to the output file

```
begin
    c ← 0;
    t[T_s, T_e] ← [0, 0];
    for i = 1 to m do
        if T_i > i.T_s then
            if c ≠ 0 then
                t.T_e ← T_i;
                Output(t, c);
            t.T_s ← T_i;
            c ← UpdateCounter(v_i, c, c_i);
        else
            c ← UpdateCounter(v_i, c, c_i);
```

Figure 21: *HTA-Merge*: Reducer (Iteration 2)

Example 14. In our example the first assigned subset is the following: $([id1, \{0, 3, 3, 3, 9, 13, 15, 20\}, \{2, 2, 1, 1, 1, 1, 1, 1\}, \{s, e, s, s, e, e, s, e\})$. The processing starts with $T_1 = 0$ and the related values $c_1 = 2$ and $v_1 = s$ triggering the first update of the counter $c = 2$. Next, the timepoint $T_2 = 3$ together with $c_2 = 2$ and $v_2 = e$ is taken generating first an output tuple $([0, 3], 2)$ and then decrementing the counter $c = 0$, because $c = c - c_2 = 2 - 2 = 0$. Then the two timepoints $T_3 = 3$ and $T_4 = 3$, having the same value as the previous timepoint, do not generate a new output tuple, but only change the value of the counter, i.e. $v_3 = v_4 = s$, $c_3 = 1$ and $c_4 = 1$ incrementing the counter c by $c = c + c_3 + c_4 = 2$. The next timepoint $T_5 = 9$ causes the generation of the next output tuple $([3, 9], 2)$. The reducer continuous until all timepoints in the subset are processed writing to the output file the merge result of the tuples in the source file.

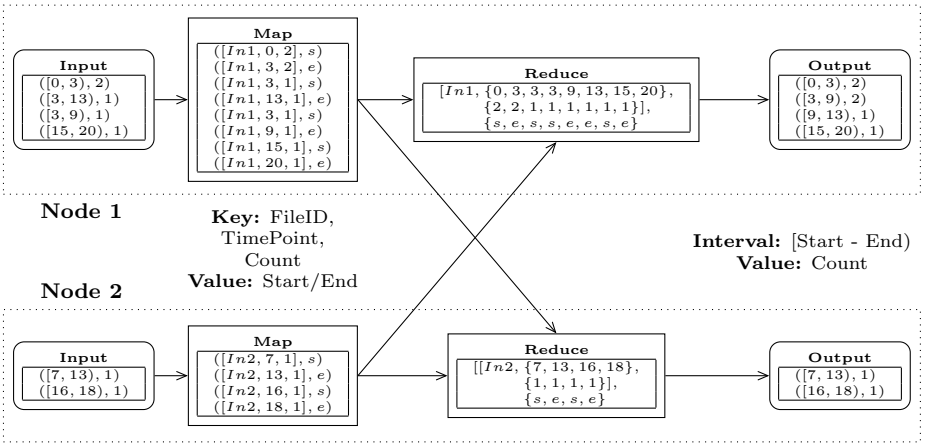


Figure 22: Graph of *HTA-Merge* Algorithm (Iteration 2)

4.3 Iteration 3: Merge Partial Result Files

The third iteration (Figure 24) is called only if the second iteration used more than one reduce-task, i.e produced more than one output file. This MapReduce iteration is repeated as long as the previous run created more than one output file. This is done by forcing the Hadoop framework to reduce the number of reduce-tasks by half after every iteration, leading to a logarithmic ($\log_2$) decrement in the number of partial result files.

4.3.1 Map

Definition 11. (Mapper) *Let id2 be a identification number of two consecutive input file and let $[T_s, T_e]$ be the time interval produced by the previous iteration with $T_s < T_e$. Further assume that c is the previously calculated aggregation result valid during the associated time-interval. Then the map-function is defined in the following way:*

$$Mapper([T_s, T_e], c) \rightarrow \{([id2, T_s, c], \text{'s'})\} \cup \{([id2, T_e, c], \text{'e'})\}$$

As a first step, the map-function (Figure 23) reads a tuple r from the assigned InputSplit. As a next step, a new identification number id_2 is generated, still using the name of the source file as main reference. The key difference is that these numbers are formed in such a way, that mappers reading from the InputSplit of two consecutive files come up with exactly the same identifier. This can be done, because the output files of the previous iteration are consecutively numerated and so the numeration can be estimated, i.e. if the previous run used 4 reduce-tasks the output file produced by them are numerated from 'out_1' to 'out_4'. The resulting number id_2, together with the timestamp T and the counter value c, form the key of the key-value pair. At the end, a char identifying the timestamp as starting or ending point is connected to the key.

Function: `Mapper`$([T_s, T_e], c)$)

Input: The tuple r read from the InputSplit
Output: The intermediate key-value pairs produced from the input tuples $([T_s, T_e], c)$

Let id_2 be the common identifier of two sequential Partial Input File;
begin
    `Emit`$([id_2, T_s, c]$, `'s'`);
    `Emit`$([id_2, T_e, c]$, `'e'`);

Figure 23: *HTA-Merge*: Mapper (Iteration 3)

Example 15. For the example we assume, that the identification number id_2, commonly used by both mappers, has the value $In1$. The rest of the procedure is basically the same as in the previous iteration. The tuple $([0, 3), 2)$ is read, the timeinterval is split $T_s = 0$ and $T_e = 3$ and the two resulting key-value pairs $([In1, 0, 2], s)$ and $([In1, 3, 2], e)$ are sent to the partitioner. The same is done for all other tuples contained in the input file.

4.3.2 Shuffle

The rest of the third iteration reuses the functions of the second iteration as previously defined. The partitioner (Figure 20) assigns, based on the identification number id_2, a valid reducer number to the single key-value pairs and the group-comparator groups the composite keys by the identification number id_2 and sorts the timestamps T.

Example 16. Notice, that due to the common number $id_2 = In1$ all intermediate key-value pairs from both nodes are sent to the reducer of node 1. This is an essential step of the final merging process of two files, each containing a partial result of the final relation.

4.3.3 Reduce

For this MapReduce iteration of the *HTA-Merge* algorithm the reduce function of the second iteration can be completely reused. The valid time interval is created by using two different consecutive timestamps $[T_i, T_{i+1})$. As before, consecutive timestamps with the same value cause only an update of the counter, without generating a result tuple. The updating of the counter c is done as in the second iteration, i.e. the two values c_i and c are added or subtracted depending on the assigned char in the value part v_i. At the end of the MapReduce

cycle, the amount of produced output-files is retrieved. In case there are still multiple output files the MapReduce cycle starts again, continuously reducing the number of reduce-tasks and files by half. Otherwise, if only one result file remains the final aggregation relation is available and the algorithm stops.

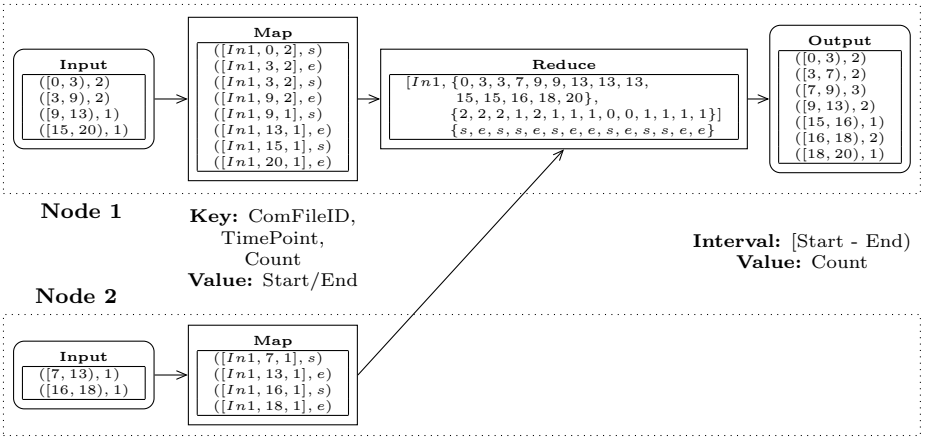


Figure 24: Graph of *HTA-Merge* Algorithm (Iteration 3)

5 Hadoop Temporal Aggregation - Partition

The two presented techniques use multiple MapReduce iterations to calculate the temporal aggregate result of the given input file. The third developed algorithm, *HTA-Partition*, performs all computations in one iteration, leading to a more complex MapReduce cycle. During the map-phase, the approach generates an additional 'overflow' key-value pair for every partition between the partition assigned to the starting and the one assigned to the ending point of the input tuple. In the reduce phase, the summed up values of the overflow pairs represent the total amount of tuples entering the current partition. At the end the result relation containing the final aggregation values and the valid time intervals is written to the output file.

5.1 Map

This approach, as just the second iteration of the *HTA-Standard* algorithm, also needs the partitions file (subsubsection 6.1.1) to divide the timeline. Afterwards, the independent partitions can be processed in parallel. Assume that the file is created successfully and that all Datanode can access and read the contained information.

Definition 12. (Mapper) *Let r be the currently processed tuple read from the input file and p be the total number of partitions. Further assume that T_s is the starting and T_e is the ending timepoint of the tuple r. Further let P_s be the partition number including T_s with $P.T_s < T_s < P.T_e$ and P_e the respective partition number for T_e. The boolean variable ov set to 'true' tags the key-value*

pair as an overflow pair. Then the map-function is defined as follows:

$$\begin{aligned} Mapper(r) \rightarrow & \{([P_s, T_s, ov = false, p], \text{'s'})\} \cup \\ & \{([P_e, T_e, ov = false, p], \text{'e'})\} \cup \\ & \bigcup_{i=P_s+1}^{P_e} ([i, i.T_s, ov = true, p], \text{'i'}) \end{aligned}$$

where i assumes all consecutive partition numbers form $(P_s + 1)$ to P_e.

The map-function (Figure 25) starts by separating the starting T_s and ending timepoints T_e of the read tuple r from the assigned InputSplit. As a next step, the corresponding partitions P_s and P_e are fetched, using the data from the partitions file. Then two key-value pairs, one for each timestamp, are created which contain as key the partition number P, the timestamp T, a boolean value ov used to identify the pair as 'overflow' and the total amount of partitions p. The character 's' or 'e' stored as value is used to tag the contained timepoint as starting or ending point of the time-interval.

As a next step, the map-function generates so called overflow-keys for ever partition between the starting partition P_s and the endpoint-partition P_e. If they are equal, nothing has to be done and the mapper processes the next tuple. Otherwise, first the starting partition P_s becomes the current partition P_c. Then a loop is used to generate an additional key-value pair for every partition between the starting partition P_s and the ending partition number P_e. These generated pairs contain the following information as key: the current partition number P_c, the starting timestamp of the current partition $P_c.T_s$, the boolean value ov set to true, indicating that this is an overflow pair and the total amount of partitions p. Due to the restriction that the basic structure of the pairs has to be same, the char 'i' is used as value, introducing an additional identification token for the overflow pairs. After each iteration of the loop the current partition P_c gets assigned the next partition in chronological order $P_c = P_c + 1$.

The amount of additional tuples generated in the loop depends directly on the total number of partitions and the length of the tuples. In the worst case, if every tuple spams from the first partition P_1 to the last partition P_p we generate p additional tuples. So the number of total key-value pairs is the sum of the additional pairs and the two generated from the starting and ending timestamp, times the amount of tuples in the input file n, i.e. equal to $(n*(p+2)) = O(n*p)$.

Example 17. The example relation and the temporal partitions are shown in Figure 26. The computation start by assigning the tuples r_1, r_2 and r_3 to the mapper of node 1. The map function, starts with r_1, separates the timepoints $T_s = 0$ and $T_e = 3$ and retrieves the respective partition number, which is in both cases $P1$. Then using the total number of partitions $p = 4$ the following two intermediate key-value pairs are generated: $([1, 0, false, 4], s)$ and $([1, 3, false, 4], e)$. Since both timepoints are located in the same partition, nothing more has to be done. The next tuple r_2 is retrieved and again the interval is divided into $T_s = 0$ and $T_e = 13$. Again the corresponding partitions $P_s = P1$ and $P_e = P3$ are looked up and two pairs emitted: $([1, 0, false, 4], s)$ and $([3, 13, false, 4], e)$. Now we can see that the timepoints are assigned to different partitions, which trigger the generation of the overflow pairs. The

Function: `Mapper`$(r[A_1, \ldots, A_n, T_s, T_e])$

Input: The Tuples read from the Input Source
Output: The IKV-Pairs produced from the Input Tuples

Let $\mathcal{P}$ be the set of partitions stored in the partitions file;
Let p be the total amount of Partitions;
begin
    $P_s \leftarrow$ Partition of $r.T_s$;
    $P_e \leftarrow$ Partition of $r.T_e$;
    $ov \leftarrow false$;
    `Emit`$([P_s, r.T_s, ov, p]$, `'s'`$)$;
    `Emit`$([P_e, r.T_e, ov, p]$, `'e'`$)$;
    if $P_s \neq P_e$ **then**
        $P_c \leftarrow P_s$;
        while $P_c \leq P_e$ **do**
            $ov \leftarrow true$;
            `Emit`$([P_c, P_c.T_s, ov, p]$, `'i'`$)$;
            $P_c \leftarrow P_c + 1$;

Figure 25: *HTA-Partition*: Mapper

first additional key-value pair is $([2, 5, true, 4], i)$, where $P2$ is the next partition following $P1 + 1 = P2$. $T = 5$ is the starting point of the partition $P2$ and the boolean value 'true' and the char 'i' are identifiers for the overflow pair. In order to complete the processing of the tuple r_2 the following last additional pair is produced: $([3, 10, true, 4], i)$, where $P3$ is the next chronological partition and $T = 10$ is the starting point.

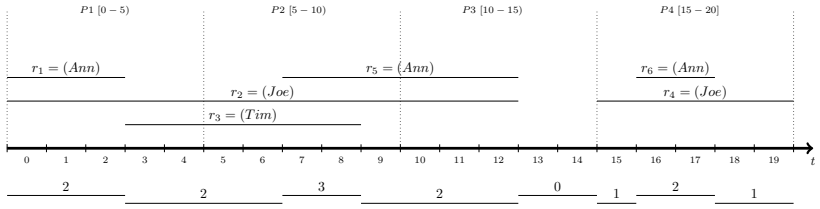


Figure 26: Example Relation with Temporal Partitions

5.2 Shuffle

The now invoked partition-function (Figure 27) is the same as previously used by the *HTA-Standard* algorithm (subsection 3.2). As described, this algorithm cannot use a modulo-function in the partition-task, because this would lead to an inappropriate partition distribution of the key-value pair subsets among the available reducers. The function calculates the $offset$ by dividing the total number of partitions p trough the total amount of available reducers R. As a next step, the partition number P is extracted from the key and divide by the calculated $offset$. The resulting number is the reduce-task to which the key-value pair is sent.

The group-comparator groups the set of key-value pairs into subsets of equal partition number P. The sorting process considers as first parameter the boolean value, i.e. all pairs containing 'true' are ranked before the one holding the value 'false'. Further, the pairs within the two

boolean groups are sorted by timestamp, placing earlier timepoints before later ones. The subset sent to the respective reducer has the following form: $([P, \{T_1, \ldots, T_m\}, \{ov_1, \ldots, ov_m\}, p], \{v_1, \ldots, v_m\})$.

Function: `Partitioner`$([P, T, ov, p], R)$

Input: The key of the current key-value pair and the total number of reducers
Output: The assigned reducer number

begin
  $offset \leftarrow \frac{p}{R}$;
  $num \leftarrow \lfloor \frac{(P-1)}{offset} \rfloor$;
  return num;

Figure 27: *HTA-Partition*: Partitioner

Example 18. The partitioner calculates the $offset = \frac{p}{R} = \frac{4}{2} = 2$, which is constant for all key-value pairs in the system, because both p and R are constant throughout the system. Each partition number is decremented by 1 and divided by the offset in order to retrieve the reducer number. For the example this means that pairs with the partition number $P1 \Rightarrow \lfloor \frac{1-1}{2} = 0 \rfloor$ and $P2 \Rightarrow \lfloor \frac{2-1}{2} = 0 \rfloor$ are send to the reducer of node 1. In the same way, the partitioner determines that the partitions $P3$ and $P4$ are assigned to node 2. The group-comparator now groups the intermediate key-value pairs by partition number P. This leads to the following grouping of the 4 key-vlaue pairs containing the partition number $P1$: $([1, \{0, 0, 3, 3\}, \{f, f, f, f\}, 4], \{s, s, s, e\})$. The contained key-value pairs are sorted prioritizing the boolean variable ov when 'true' and as second parameter the timepoint value T.

5.3 Reduce

Definition 13. (Reducer) *Let $T = \{T_1, \ldots, T_m\}$ be the set of timestamp all having the same partition number P and p be the total number of partitions. In addition, let $V = \{v_1, \ldots, v_k, v_{k+1}, \ldots, v_m\}$ and $O = \{ov_1, \ldots, ov_k, ov_{k+1}, \ldots, ov_m\}$ be the related sets with the same cardinality m. The subset $O' = \{ov_1, \ldots, ov_k\}$ contains k booleans with the value 'true' and the subset $V' = \{v_1, \ldots, v_k\}$ is composed of the k associated values. Further let $T'' = \{T_j, T_{j+1} \ldots, T_{j+l}\} \subset T$ and $|T''| = l$ be a maximum sequence of consecutive identical timepoints, i.e. $T_i = T_{i+1}$ for $i = j, \ldots, (j + l)$. The subset $V'' = \{v_j, v_{j+1} \ldots, v_{j+l}\}$ is the respective set of values associated with T'' having the same cardinality l. Then the reducer is defined as follows:*

$$Reducer([P, T, O, p], V) \rightarrow \bigcup_{i=k+1}^{m-1} ([T_i, T_{i+1}], c_i + inc)) \cup \{([T_m, P.T_e], c_i + inc))\}$$

where c_i is the value returned by the function f_5 with $c_0 = 0$ and inc is the increment returned by f_6.

$$c_i = f_5(V'') = \sum_{\substack{v \in V'' \wedge \\ v = 's'}} (c_{i-1} + 1) + \sum_{\substack{v \in V'' \wedge \\ v = 'e'}} (c_{i-1} - 1)$$

$$inc = f_6(V') = \sum_{i=1}^{k} v_i$$

Since the *HTA-Partition* algorithm uses only one MapReduce-iteration, the reduce-function (Figure 28) creates the file containing the result relation, i.e. the time-interval and the related temporal aggregation value. As a first step, the function initializes the time-interval t, the counter c and the increment variable *inc*. Then the function starts processing the subset of the assigned intermediate key-value pairs all having the same associated partition number P. Afterwards, the boolean parameter *ov* associated with the key of the pair is checked. In the case the variable *ov* is true, meaning that the currently processed pairs is of type 'overflow', the starting timestamp of the time-interval $t.T_s$ is updated with the currently processed timestamp $T_i = P.T_s$. Since it is guaranteed that the whole subset is totally sorted it can be assumed, that all pairs of type 'overflow' are at the beginning of the set and in chronological order. Therefore, the amount of pairs is counted using a loop, incrementing the variable *inc* by 1 as long as the processed pair is of type overflow. In the worst case this loop can run $O(n)$ times, where n is the total amount of tuples in the input file, which is the case when all tuples cross the currently processed partition.

When all additional pairs have been processed, the check at the beginning prevents the other tuples to enter this part of the algorithm and forwards them to the next test. This analyzes if the currently processed timepoint T_i is strictly greater than the starting timestamp $t.T_s$. In the case this is not true i.e. two or more consecutive timestamps have the same value, the algorithm jumps to the `UpdateCounter(`$v_i, c, 1$`)` (Figure 16), which updates the counter value c according to the passed parameters v_1 and 1. In the other case, a further check is done, which prevents the generation of an output tuple when the aggregate value is equal to 0. Assuming that $(c + inc) \neq 0$, the current timepoint T_i becomes the ending timestamp $t.T_e$ and a new tuple is written to the output file consisting of the time-interval t and the aggregation value $(c + inc)$. As a next step, the starting timepoint of the interval $t.T_s$ is updated with the timestamp T_i and the counter c with the result of the `UpdateCounter(`$v_i, c, 1$`)` function.

Once all timepoints in the assigned subset are processed a last check is performed analyzing if the valid time interval of the last written output tuple reached the ending point of the current partition $P.T_e$. In the case the last tuple did not reach the end of the partition, a last tuple has to be produced, breaching the gap between the last output tuple and the ending point of the partition. This is done by overwriting $t.T_e$ with the ending timestamp of the partition $P_r.T_e$ and writing the tuple to the output file.

Example 19. The reducer of node 1 receives two subsets having the partition numbers $P1$ and $P2$. The first subset $([1, \{0, 0, 3, 3\}, \{f, f, f, f\}, 4], \{s, s, s, e\})$ does not contain any overflow pairs, leading to $inc = 0$. next, the contained timestamps $T_1 = 0$ and $T_2 = 0$, having the same value do not generate any

```
Function: Reducer([P, {T_1, ..., T_m}, {ov_1, ..., ov_m}, p_tot], {v_1, ..., v_m})
Input: A subset of intermediate key-value pairs with the same partition number P
Output: Writes output tuple to the output file
begin
    t[T_s.T_e] ← [0, 0];
    c ← 0;
    inc ← 0;
    for i = 1 to m do
        if ov_i = true then
            t.T_s ← T_i;
            while ov_i = true ∧ v_i = "i" do
                inc ← inc + 1;
                i ← i + 1;
        if T_i > t.T_s then
            if (c + inc) ≠ 0 then
                t.T_e ← T_i;
                Output(t, (c + inc));
            t.T_s ← T_i;
            c ← UpdateCounter(v_i, c, 1);
        else
            c ← UpdateCounter(v_i, c, 1);
    if (c + inc) ≠ 0 ∧ t.T_e < P.T_e then
        t.T_e ← P.T_e;
        Output(t, (c + inc));
```

Figure 28: *HTA-Partition*: Reducer

output tuple, but only cause the following update of the counter: $v_1 = s, v_2 = s \Rightarrow c = c + 1 + 1 = 2$. The next timestamp $T_3 = 3$ triggers the generation of a output tuple $(t, c) = ((0, 3], 2)$, followed by an update of the counter $c = c + 1 = 3$. The last timestamp of the set $T_4 = 3$ has the same value as the previous one and the associated value $v_3 = e$. Therefore, a decrement of the counter $c = c - 1 = 2$ is caused. Since the last generated output tuple does not reach the ending point $P.T_e = 5$ of the partition $P1$, an additional tuple $([3, 5), 2)$ is written to the file. Next, the subsequent subset $([2, \{5, 5, 7, 9\}, \{t, t, f, f\}, 4], \{i, i, s, e\})$ is processed by node 1. This time there are overflow pairs at the beginning, leading to a update of $inc = 2$. Further, the starting point of the interval $t.T_s$ gets assigned the start timeinstant of the current partition $t.T_s = P.T_s$. The next timestamp $P.T_s < T_3 = 5 < 7$ being greater as the starting point of t immediately generates the new output tuple $(t, c + inc) = ((5, 7], 2)$. As a next step, the counter is updated as follows: $c = c + 1 = 1$. The last timestamp $T_4 = 9$ produces again a new tuple $(t, c + inc) = ((7, 9], 3)$ and updates the counter to its final value $c = c - 1 = 2$. At the end since the last tuple did not reach the ending point of partition $P2$ an additional record $(t, c + inc) = ((9, 10], 2)$ is created by using the last value of the counter c.

6 Empirical Evaluation

In this section we provide the results of the experimental evaluation together with the measured execution time of the three algorithm using different input files. Further we will compare the MapReduce approaches with another temporal aggregation algorithm, namely the *Bucket & Balance Tree* algorithm.

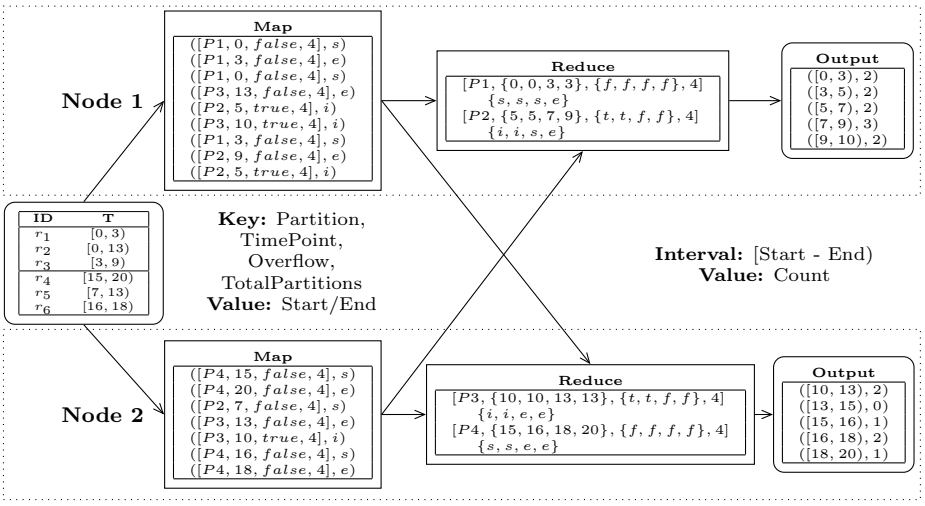


Figure 29: Graph of *HTA-Partition* Algorithm

6.1 Setup and Data Sets

The Hadoop-cluster used for the experimental evaluation consists of 5 Datanodes, connected using a LAN-network with a capability of 100 Mbit/sec. Due to the very small amount of nodes, one node has to function as Namenode, Jobtracker and Datanode at the same time. Each machine has 2 GB RAM, a single core processor with 2.7 GHz and uses a Linux based operating system. Further we use Hadoop version 1.1.2 with a default 64 MB data block size running on Java 1.6.

The input files contain data relations having from 10,000,000 up to 500,000,000 tuples. Further, the files originate from two scenarios that use a different time granularity and associate various non-temporal attributes with each data-record. A CSV file format is used, separating the attributes with a semicolon and the single tuples using new lines. Each tuple consists of 12 fields, 10 for the non-temporal attributes and 2 for the valid timestamp. The size of the files depends on the number of contained tuples and is between 1 GB with $10 * 10^6$ tuples and 60 GB with $500 * 10^6$ data records. Due to the random number generator used to produce the different data files, all records and long-lived tuples are evenly distributed.

The runtime of all tree approaches is measured and compared with each other. As explained in detail, the *HTA-Standard* (section 3) and the *HTA-Partition* algorithm (section 5) need a partitions file (subsubsection 6.1.1) in order to group the timepoints and so form independent subsets. Two different forms of the partitions file are used during the empirical evaluation of the approaches, the *linear* and the *probability* partitions file. They contain the same amount of temporal partitions, but the first one uses a constant and the second one a variable partition range. Overall, the following list of algorithms is empirically evaluated:

- *HTA-Standard* Linear: from section 3 with a linear partitions file

- *HTA-Standard* Prob: from section 3 with a probability partitions file
- *HTA-Partition* Linear: from section 5 with a linear partitions file
- *HTA-Partition* Prob: from section 5 with a probability partitions file
- *HTA-Merge*: from section 4

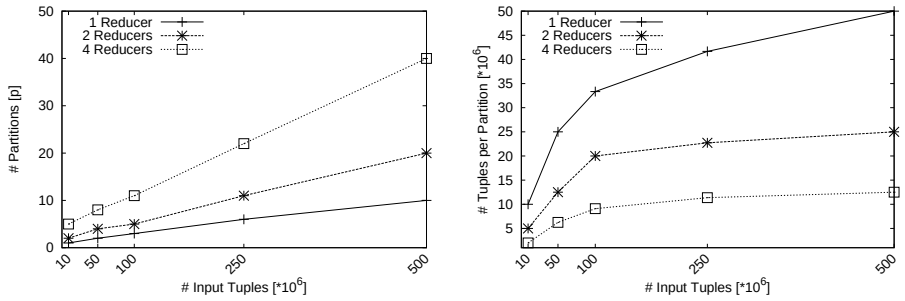
Further a comparison between the developed algorithms and the *Bucket* algorithm (subsubsection 6.1.2) is provided. Since the bucket algorithm alone does only split the input file into buckets, the main memory based *Balanced Tree* algorithm is used to compute the final ITA result. The approach is compared with the slowest and fastest MapReduce in both scenarios. This leads to the following list of comparisons:

- Comparison of *Bucket & Balanced Tree Aggregation* algorithm with the slowest approach of scenario 1, *HTA-Partition* Prob
- Comparison of *Bucket & Balanced Tree Aggregation* algorithm with the fastest approach of scenario 1, *HTA-Standard* Linear
- Comparison of *Bucket & Balanced Tree Aggregation* algorithm with the slowest approach of scenario 2, *HTA-Merge*
- Comparison of *Bucket & Balanced Tree Aggregation* algorithm with the fastest approach of scenario 2, *HTA-Standard* Linear

6.1.1 Partitioning the Timeline

The so called partitions file is used by the algorithms in order to divide the timeline into single partitions. The main purpose of these partitions is to enable parallel processing without introducing an error in the calculations of the temporal aggregation values. Further, this technique forms completely independent chunks that can be evenly distributed among the available datanodes. In order to construct an appropriate partitions file the function needs some information about the dataset, such as minimal and maximal timeinstants and the approximated number of tuples. The two timepoints are needed to initialize the start and end of the timeline, the amount of tuples influences the number of partitions created (Figure 30a) and so the amount of tuples contained in each partition (Figure 30b). Two different strategies can be used to construct the partitions file.

Linear Partitioning: As just the name suggest a linear functions is used to determine the number of partitions p, given the number of tuples in the input file and the amount of available reducers R. The entire timeline, from min to max is simply divided into p parts of equal length, whereas the starting point of the first partition is always min and the ending point of the last partition is always max. Notice, this method works best, if the tuples are evenly distributed over the whole timeline, because then each partition contains a similar amount of timepoints.



(a) Number of Partitions per Reducer

(b) Number of Tuples in each Partition

Figure 30: Number of Partitions and Tuples per Partition

Probability Partitioning: The probability partitioning is a refinement of the linear partitioning, taking into consideration the distribution of the tuples over the timeline. The distribution is determined by randomly extracting an user defined amount of tuples from the input file, creating a sample file. Next, the timeline is divided again in p partitions all having the same range. Afterwards, the distribution of the tuples in the sample file is used to modify the length of the single partitions, creating shorter partitions in case of tuple accumulations and longer partitions in case of sparse distribution. This technique creates partitions containing a similar amount of timepoints despite an antisymmetric data distribution. Notice, the bigger the sample file is, the exacter is the estimation of the tuple distribution in the original input file, but causing a increment in creation time of the sample file and the calculation of the probabilities.

6.1.2 Bucket & Balanced Tree Aggregation Algorithm

The bucket algorithm presented in the work of Bongki Moon et al. [14] is a sequential approach that allows the processing of large data file using an arbitrary main memory based temporal aggregation algorithm. The approach splits the timeline of the input relation into so called buckets, which entirely fit in main-memory. Long-lived tuples overlapping an entire bucket are only assigned to the buckets where the tuple starts and ends reducing the amount of additional tuples needed to preserve the correctness of the algorithm. The additional information about the long-lived tuples is kept in a separate, so called meta array, which has to be considered in the calculations of the final aggregation relation. After the construction of the buckets, the Balanced Tree Aggregation algorithm, also presented in [14], is used to generate the final ITA result. This approach loads a entire bucket in main memory and constructs a dynamic tree, where the nodes contain the start and end timepoints of the tuples and two partial aggregation values. The result is computed in a final in-order traversal of the tree, extracting the valid time intervals and the associated aggregation values.

6.2 Scenario 1: Employee-Project Records

The first scenario involves a company that recorded all hours spent by their employees on the different project they developed over the past 10 years. This

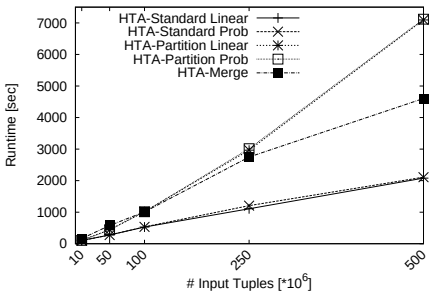
means that the involved timeline has a range of 87,611 hours (10 years) and the tuples have a defined maximal length of 8,760 hours (1 year). Further each record has 10 non-temporal attributes associated with the time-interval in which the tuple was valid. Further there are many long-lived tuples having a maximal length of about 10% of the total timeline. Further, with increasing tuple count, the result of the aggregation function becomes very large, because of the low number of possible timepoints. This first setup was chosen, because due to the relative short timeline the tuples tend to accumulate when their number is increased, leading to many overlaps and minimizing the occurrence of gaps in the result relation to almost zero.

6.2.1 Comparison of MapReduce Algorithms

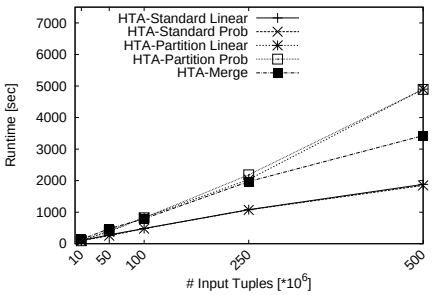
The runtime of the three algorithms is presented in the charts contained in Figure 31. The first chart (Figure 31a) shows the results using only a single reducer, which means that the partitioner assigns all key-value pairs to the same Datanode running on a single machine in the cluster. In this situation, the *HTA-Partition* algorithm shows the worst behavior. The amount of time needed to calculate the aggregation relation increases rapidly with bigger input files. The performance results can be explained by the fact, that this approach generates additional so called 'overflow' tuples during the map phase. Due to the relatively long tuples, the total number of intermediate key-value pairs that have to be handled, first by the partitioner and then by the reducer, increase dramatically. As expected, measurements of the number of tuples being generated by the map-task resulted in a drastic percental increment of overflow tuples as the number of input tuples is augmented. The main advantage of the technique to only use a single MapReduce iteration does not save enough time to compensate the additional time spent to process the overflow pairs. Further, using only a single Reducer is not really compatible with the principle used by this algorithm, which is divide the timeline into partitions in order to enable parallel processing.

The *HTA-Merge* approach shows a better behavior than the *HTA-Partition* algorithm. Due to his nature, only using one reducer and having a short timeline, shows off the advantage of the merge approach. The first benefit is that in this situation the *HTA-Merge* takes only two MapReduce iterations to complete the final result. Further, the amount of key-value pairs that have to be handled by the cluster does not considerably increase. Each chunk of input file determined by the byte-offset is bound by a worst case complexity $O(2x - 1)$, where x is the amount of tuples in the chunk. This bound applies also to the second MapReduce iteration, where the partial results are merged together to form the final aggregation relation. A clear disadvantage of this technique is the very small size of the chunks formed using the offset as reference, leading to many subsets which contain only a few tuples.

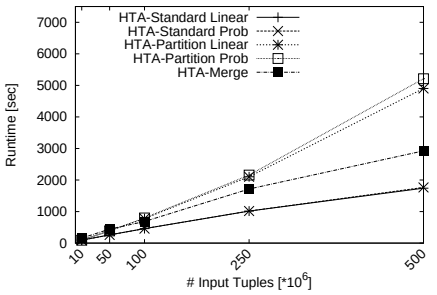
The algorithm with the best performance is the *HTA-Standard*. This technique uses the most MapReduce iterations, one more than the *HTA-Merge*, and needs some additional tuples to be able to generate the final result relation. Despite this two drawbacks, the needed time is substantially shorter compared to the previous algorithms. The first reason is that the second and third MapReduce iteration have to deal with considerably less tuples than the first iteration step. While the merging step of the *HTA-Merge* approach has to process a file having a similar length as the input file, the second iteration, thanks to the



(a) Runtimes Using 1 Reducer



(b) Runtimes Using 2 Reducers



(c) Runtime Using 4 Reducers

HTA-Standard

# Tuples	Linear 2R	Linear 4R	Prob 2R	Prob 4R
10,000,000	4.80%	9.58%	3.98%	8.86%
50,000,000	-1.45%	6.12%	7.59%	6.14%
100,000,000	8.71%	13.36%	10.49%	14.23%
250,000,000	2.49%	8.53%	11.12%	15.77%
500,000,000	9.46%	16.23%	12.34%	16.32%

HTA-Partition

# Tuples	Linear 2R	Linear 4R	Prob 2R	Prob 4R
10,000,000	12.90%	12.94%	12.33%	14.21%
50,000,000	13.25%	10.52%	7.45%	12.86%
100,000,000	18.25%	23.56%	18.96%	21.44%
250,000,000	31.35%	29.08%	27.51%	28.42%
500,000,000	30.92%	30.89%	31.26%	26.78%

HTA-Merge

# Tuples	2R	4R
10,000,000	17.25%	3.40%
50,000,000	17.45%	23.45%
100,000,000	21.54%	32.22%
250,000,000	28.33%	37.60%
500,000,000	25.57%	36.50%

(d) Runtime Improvements Compared to 1 Reducer

Figure 31: Runtime of all 3 Algorithms using Different Numbers of Reducers (Scenario 1)

grouping by the timepoints, has to handle a file with the length similar to the amount of time instants. Further, the additionally generated key-value pairs depend on the number of partitions p, not on the amount of tuples n as in the *HTA-Partition* approach, which are considerably less. A last advantage is that the arithmetic functions used in the three are all very elementary and therefore very time-efficient.

The graphs in Figure 31b and Figure 31c show the behavior of the developed approaches using the same input file, but two and four Reducers respectively. The Table in Figure 31d shows the performance improvements compared with the measured execution time using a single reducer. So all three approaches benefit form the augmentation of the reducers, which splits the workload for a single node during the reduce phase by half and quarter respectively.

Despite the fact, that the *HTA-Standard* technique has the worst improvement, it still remains the fastest of the three developed algorithms. With increasing amount of tuples, the percental performance improvement increases, having a highest value of about 16% compared to the single reducer run. Due to the increment in the number of reducers, also the amount of partitions is increased, leading to more additional 'transfer' key-value pairs generated by the second iteration of the *HTA-Standard* approach. This means that an increment in the number of pairs to be handled by Hadoop has more impact than the splitting of the work during the reduce-phase of the MapReduce iteration.

The *HTA-Partition* algorithm has with up to 30% improvement compared to the runtime with only one reducer, the second best percental reduction of the execution time. Nevertheless, it remains the slowest approach, because of the

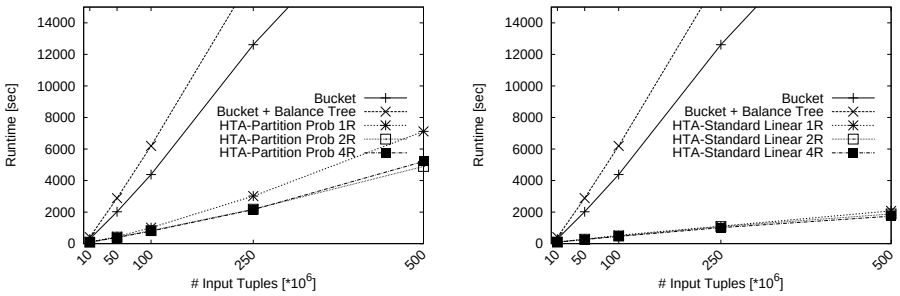
previously mentioned disadvantages. Due to the fact that with the Reducers also the amount of partitions increased, the amount of 'overflow' key-value pair is even bigger than in the run with only one reducer. The main reason for the improvement in time is, since the algorithm uses only one MapReduce iteration, the division of the workload to more Datanodes. So the considerably increased amount of intermediate key-value pairs can be processed in parallel. Notice, the performance using two and four Reducers is nearly the same, meaning that the time saved with the parallel computation is more or less the same as the time needed to process the additional 'overflow' tuples.

The approach with the best improvement achieved using an increased number of reducers is the *HTA-Merge* algorithm. This is quite surprising, because every additional reducer-task generates an additional output file, which has again be merged until only a single file remains. The main reason for this behavior is that due to the small amount of possible time instants, the files to be merged contain only a few tuples leading to a fast merging process. The measured execution time lead to the statement, that the time saved by using more datanodes and process the key-value pairs in parallel is much larger, then the time needed to do an additional MapReduce iteration.

The two different creation techniques of the partitions file have only very little or no effect on the runtime of the algorithms. The partitions file containing all partitions with the same range is equally effective than the file containing time intervals adjusted considering the tuple distribution. The main reason for this is the creation method (i.e. the random number generator), used to generate the input files, which always evenly distributes the single tuples.

6.2.2 Comparing to Bucket & Balance Tree Algorithm

Figure 32 shows a comparison of the runtime of the *Bucket and Balance Tree* algorithm with the MapReduce approach with the worst execution time (Figure 32a) and with the best execution time (Figure 32b) respectively. In both comparisons there is a clear difference between the measured performance of the two different techniques.



(a) Compare *HTA-Partition* Prob to Bucket & Balance Tree Algorithm

(b) Compare *HTA-Standard* Linear to Bucket & Balance Tree Algorithm

Figure 32: Compare Slowest and Fastest MapReduce Approach in Scenario 1 to Bucket and Balance Tree Algorithm

The performance of the *HTA-Partition* algorithm using a single reducer is by far the slowest approach using the data files of scenario 1 but is still by a

factor of 5 faster as the *Bucket and Balance Tree* algorithm. The fastest of the MapReduce approaches, the *HTA-Standard* Linear, outperforms the *Bucket and Balance Tree* algorithm by a factor of 20. The main reason for the long run time of the *Bucket and Balance Tree* algorithm is the needed division of the input file into small buckets which fit in main memory. Since the timeline in scenario 1 are very short, and the size of each bucket has to be relatively small, a bucket includes only a short time range. Due to the short range and the long tuples nearly every tuple is split into two records, each assigned to a different bucket. Therefore, the size of all buckets together is almost doubled compared to the initial size of the original input file. Further, due to the splitting of the tuples, the buckets contain many equal timepoints leading to many updates of the variables contained in the nodes of the balanced tree during the calculation of the final ITA result relation.

6.3 Scenario 2: Phone-Company Records

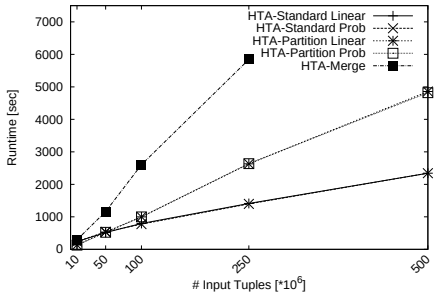
The second scenario involves the data recorded by a telephone company. All calls over a period of one year were registered using a granularity of seconds and having a maximal length of a day. This setup leads to a timeline with a range of 31,536,000 and a maximal tuple length of 86,400 seconds. Each data record consists of 10 non-temporal attributes and the related time-interval marking its validity. The second scenario, in difference to the first one, involves a relative long timeline, while the single tuples are short, leading to many possible gaps and few overlappings even when the amount of tuples increases. In addition the amount of long-lived tuples is very low because the tuples have a maximal length of 0.27% compared to length of the involved timeline. Further, the amount of possible time-intervals in the output file increases drastically, since the values of the aggregation function might change at every time-instant.

6.3.1 Comparison of MapReduce Algorithms

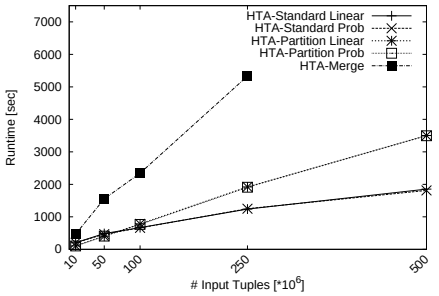
The execution time of the approaches are presented in the charts in Figure 33. The structure is the same as before, the first diagram (Figure 33a) shows the measured runtime when only a single Datanode is involved in the final reduce phase. In contrast to the previous scenario, now the *HTA-Merge* algorithm has by far the worst performance. The reason for this deterioration is the merging processes needed to produce the final result. Due to the increment in timepoints, the size of all partial results augmented together with the possible generated intervals associated with each partial aggregation value. During the execution of the data file containing $500 * 10^6$ tuples the size of the partial results even exceeded the capacity of the distributed file system and therefore could not be completed

The *HTA-Partition* algorithm shows a huge performance improvement compared to the previous scenario. The reason for this reduction in running time by nearly half, is the length of the tuples contained in the input file. The short tuple do not cross any or only a few temporal partitions notably reducing the amount of additionally generated key-value pairs.

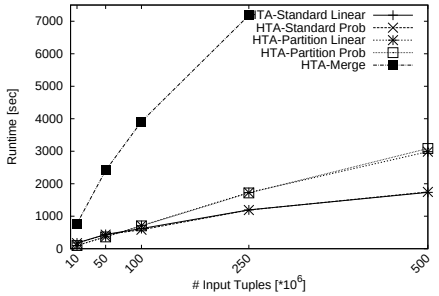
The fastest approach is again the *HTA-Standard* algorithm, regardless the slight increment in execution time compared to the first scenario. This augmentation is caused by the raised amount of possible timepoints. This leads to a



(a) Runtimes Using 1 Reducer



(b) Runtimes Using 2 Reducers



(c) Runtime Using 4 Reducers

HTA-Standard

# Tuples	Linear 2R	Linear 4R	Prob 2R	Prob 4R
10,000,000	14.35%	25.71%	15.51%	28.08%
50,000,000	5.89%	14.68%	13.12%	19.25%
100,000,000	16.07%	22.03%	14.91%	24.65%
250,000,000	12.33%	15.44%	10.67%	14.86%
500,000,000	20.92%	25.93%	22.65%	25.43%

HTA-Partition

# Tuples	Linear 2R	Linear 4R	Prob 2R	Prob 4R
10,000,000	21.52%	25.95%	18.75%	23.40%
50,000,000	24.14%	30.94%	22.25%	30.77%
100,000,000	21.50%	28.76%	23.28%	29.62%
250,000,000	27.02%	34.28%	27.51%	34.88%
500,000,000	27.89%	38.51%	27.47%	36.02%

HTA-Merge

# Tuples	2R	4R
10,000,000	-58.88%	-162.30%
50,000,000	-33.78%	-107.31%
100,000,000	10.03%	-50.21%
250,000,000	8.77%	-22.91%

(d) Runtime Improvements Compared to 1 Reducer

Figure 33: Runtime of all 3 Algorithms using Different Numbers of Reducers (Scenario 2)

longer first iteration of the algorithm, where the key-value pairs are formed using the timepoints as main reference. Despite the increased amount of arithmetic computations, the fact that fewer tuples per timepoint are involved decreases the execution time of the other two MapReduer iterations.

The graphical illustrations in Figure 33b and Figure 33c show the different execution time of the three approaches while increasing the used Datanodes in the reduce-phase first to 2 and then to 4. Figure 33d shows the percental improvements of the performance compared to the results measured with a single reducer. An unexpected outcome of the experiments is that the execution time of the *HTA-Merge* algorithm, in contrast to the first scenario, further increased, leading to a negative improvement. This means, that the measured time drastically increased making the *HTA-Merge* approach by far the slowest algorithm in processing the data of scenario 2.

The *HTA-Standard* shows a better improvement compared to the execution time measured during the first scenario. The increased amount of timepoints to be handled by the approach is processed in parallel, further decreasing the overall runtime.

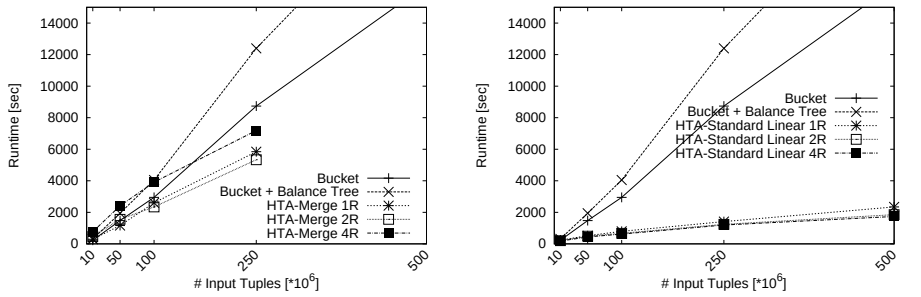
The algorithm with the best measured improvement is the *HTA-Partition* approach. The best measured reduction is with nearly 40% the best overall improvement. The reason for this is that the amount of key-value pairs is not heavily increased, because most tuples do not cross any or only a few temporal partitions. This leads to a noticeable decrement of the work done during the shuffle phase and by the single reduce tasks. Notice, in this scenario, the execution time of the *HTA-Partition* algorithm approaches the runtime of the

HTA-Standard approach. It can be assumed that a further increment in reduce tasks and size of the input file, further benefits the *HTA-Partition* algorithm, decreasing the execution time outperforming all other approaches.

As just in the previous scenario, also in this the different techniques used to create the partitions file has no effect on the performance of the algorithms.

6.3.2 Comparing to Bucket & Balance Tree Algorithm

In Figure 34 a graphical illustration of the comparison of the *Bucket and Balance Tree* algorithm and the slowest (Figure 34a) and fastest (Figure 34b) MapReduce approaches is shown using the data files of scenario 2.



(a) Compare *HTA-Merge* to Bucket & Balance Tree Algorithm

(b) Compare *HTA-Standard* Linear to Bucket & Balance Tree Algorithm

Figure 34: Compare Slowest and Fastest MapReduce Approach in Scenario 2 to Bucket and Balance Tree Algorithm

In the second scenario, the difference between the performance of the slowest MapReduce algorithm *HTA-Merge*and the *Bucket and Balance Tree* is not so obvious. However, due to the additional time needed by the *Bucket and Balance Tree* algorithm to calculate the *ITA* result, it can be assumed that the MapReduce approach has a slightly better performance.

However, the MapReduce algorithm, *HTA-Standard* Linear, with the best execution time outruns the other approach by a factor of 6. In general we notice a improvement in the runtime of the *Bucket and Balance Tree* algorithm. The cause is that the short tuples and the relatively long timeline of the second scenario benefit the bucketing process. The shortness of the data records makes it more likely that the starting and ending timepoint of the tuples are in the same bucket. This leads to no or only a small increment in the total size of the data.

6.4 Summary

In general we can say, that an increment in the number of reducers is beneficial for all three MapReduce algorithms. Further we noticed, that the improvements of the runtime become bigger, when the amount of tuples in the input file increased. This is a positive effect, because the algorithms are designed to handle very large data files containing vast amount of tuples. Nevertheless, due to the very restrictive amount of disk-space available to us, the creation and handling of bigger files was impossible. Further our cluster has only a size of

5 Datanodes, which is extremely small compared to other clusters composed of hundreds or even thousands of connected machines. The comparison with the main-memory based *Bucket and Balance Tree* algorithm showed, that the developed approaches can compete with existing techniques. In addition, a further benefit of the MapReduce approaches became clear, which is the relative insensitivity regarding the structure of the input relation and the contained amount of long-lived tuples.

7 Conclusion and Future Work

In this thesis we provided three different algorithms to calculate the instant temporal aggregates for large raw data files using a small cluster. In order to compute the calculations in parallel, the algorithms implement the abstract programming model MapReduce and were executed on the open source framework Hadoop. For every technique we explained in detail the general concept and described the single methods together with some implementation details. Additionally, we evaluated the algorithms using different scenarios having a varying time granularity and so testing diverse aspects of the techniques. Further we compared the developed approaches with an existing sequencial algorithm designed to handle large datasets. The results showed that our algorithms have a good performance and outrun the other approach by a factor of 5 to 20.

Regarding future work, some general Hadoop improvement techniques can be applied to the cluster, such as enabling the compression of the temporarily stored intermediate key-value pairs and define a so called *combiner* function. The possibility to compress the output of the mapper and the result file is already integrated in the framework and can simply be activated by setting some configuration flags. A smaller size of the temporal and final data decreases the time needed for all performed reading and writing tasks. Further the additional combiner can be used to get rid of sequences of identical timestamps. The function combines the values associated with identical timepoints to a single merged value, i.e. $(0, s)(0, s)$ becomes $(0, ss)$. These two improvements could noticeably reduce the execution time of the algorithms.

A further improvement involves the integration of a relational database, such as *Oracle* or *PostgreSQL*, in the distributed file system (HDFS) used by Hadoop. To achieve this task an additional programs is needed, namely *Sqoop*. This software allows users to extract data from a relational database and import it into Hadoop.

Since currently the algorithms can only evaluate the aggregation function *count* one of the next steps is to extend the approaches to allow the execution of a selectable aggregation operator. The needed changes to achieve this task are kept to a minimum and basically involve the re-engineering of the `UpdateCounter(...)` function and some other small changes.

Notice, that the tree techniques until this point are designed to construct temporal aggregation relations without considering a possible group-by clause. In order to extend our work to enable the use of a so called grouping attribute the following steps are essential. First we have to include the different values of the group-by clause in the key of every key-value pair. Further, the partition-function has to be changed in order to consider this additional information during the assignment of the pairs. Notice, that the partitioning is a delicate

matter, because we have to ensure that identical grouping values are sent to the same reducer, while enabling large accumulations to be divided into several partitions. Next, a sort-comparator orders the intermediate key-value pairs considering the grouping attribute and the timestamp during the determination of the sort order. As a last step, each reducer has to generate a valid time interval and calculate the associated aggregation values. Due to the fact that during the sort-phase, the grouping attributes were considered, intermediate key-value pairs having the same grouping value are consecutively ordered. This means, that that the reducer can process the pairs one after the other, generating a result relation per grouping attribute.

References

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[2] Michael Böhlen, Johann Gamper, and Christian S Jensen. Multi-dimensional aggregation for temporal data. In *Advances in Database Technology-EDBT 2006*, pages 257–275. Springer, 2006.

[3] Michael H Böhlen, Johann Gamper, and Christian S Jensen. Towards general temporal aggregation. In *Sharing Data, Information and Knowledge*, pages 257–269. Springer, 2008.

[4] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[7] Robert Epstein. Techniques for processing of aggregates in relational database systems. Technical report, Technical Report UCB/ERL, 1979.

[8] Dengfeng Gao, Jose Alvin G Gendrano, Bongki Moon, Richard T Snodgrass, Minseok Park, Bruce C Huang, and Jim M Rodrigue. Main memory-based algorithms for efficient parallel aggregation for temporal databases. *Distributed and Parallel Databases*, 16(2):123–163, 2004.

[9] Jose Alvin G Gendrano, Bruce C Huang, Jim M Rodrigue, Bongki Moon, and Richard T Snodgrass. Parallel algorithms for computing temporal aggregates. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 418–427. IEEE, 1999.

[10] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[11] Nick Kline and Richard T Snodgrass. Computing temporal aggregates. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 222–231. IEEE, 1995.

[12] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM)*, 29(3):699–717, 1982.

[13] Donald Miner and Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems.* " O'Reilly Media, Inc.", 2012.

[14] Bongki Moon, Ines Fernando Vega Lopez, and Vijaykumar Immanuel. Efficient algorithms for large-scale temporal aggregation. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):744–759, 2003.

[15] Shamkant B Navathe and Rafi Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1):147–175, 1989.

[16] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[17] Srinath Perera. *Hadoop MapReduce Cookbook*. Packt Publishing Ltd, 2013.

[18] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007.

[19] Eric Sammer. *Hadoop Operations*. O'Reilly Media, Inc., 2012.

[20] Richard T. Snodgrass, Santiago Gomez, and Jr LE McKenzie. Aggregates in the temporal query language tquel. *Knowledge and Data Engineering, IEEE Transactions on*, 5(5):826–842, 1993.

[21] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[22] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: Friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[23] Paul A Tuma. Implementing historical aggregates in tempis. *Master's Thesis*, 1992.

[24] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[25] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Mapreduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

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

[27] Xinfeng Ye and John A Keane. Processing temporal aggregates in parallel. In *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, volume 2, pages 1373–1378. IEEE, 1997.