High Frequency Batch-oriented Computations over Large Sliding Time Windows [☆]

Leonardo Aniello[a], Leonardo Querzoni[a], Roberto Baldoni[a]

[a] *Cyber Intelligence and Information Security Research Center and Department of Computer, Control, and Management Engineering "Antonio Ruberti" University of Rome "La Sapienza" Via Ariosto 25, 00185, Rome, Italy Email: aniello,querzoni,baldoni@dis.uniroma1.it*

Abstract

Today's business workflows are very likely to include batch computations that periodically analyze subsets of data relative to specific time ranges in order to provide strategic information for stakeholders and other interested parties. The frequency of these computations directly impacts on how much updated such information can be, which provides an effective measure of their usefulness. This in turn drives towards solutions that allow to carry out batch elaborations more often, so as to always have updated information as soon as possible. How often they can be executed usually depends either on application-specific requirements or on some other constraints, typically about completion time because it's uncommon to start a new batch computation before the previous one has completed, and the typical amounts of data to elaborate in these scenarios are so large that a computation can take very long.

In this paper we propose a model for batch processing on sliding time window event computations that allows the definition of multiple metrics for performance optimization. These metrics specifically take into account the organization of input data to minimize its impact on computation latency. The model is then instantiated on Hadoop, a batch processing engine based on the MapReduce paradigm, and a set of strategies for efficiently arranging input data is described and evaluated.

Keywords: event processing, batch processing, time window based computations

1. Introduction

Event processing is a constantly evolving research area which keeps growing to adapt to emerging technologies and paradigms [1]. With event processing, events produced by possibly different sources are usually collected in bunches

[☆]This is an extended version of a paper from the same authors appeared at SAC 2013.

delimited by time windows and then elaborated to produce other events as output. Several real applications require indeed to recognize particular patterns within specific time lapses or to produce periodic reports on what happened in precise time ranges. A relevant example for the first case is represented by Intrusion Detection Systems (IDSs), which keep monitoring network traffic data searching for known malicious signatures, trace them and raise alerts whenever too many suspect activities occur within a defined time interval. Port scan detection techniques based on the activities observed in specific time windows are investigated in [2] and [3]. In this scenario, where a large number of network probes can produce high rate event streams, it is advisable to adopt large time windows (hours, days) to catch *slow attacks*; at the same time, it is necessary to have frequent (every few seconds) analysis results to promptly react to alarms. An example of the second case is represented by algorithmic trading, an application scenario where result latency is critical to profitability. Such scenario is characterized by large data volumes (millions of trades per day), medium to large observation windows (hours, days) and frequent (down to a second) updates to quickly catch highly volatile financial opportunities. We refer to this kind of event processing as Time Window Based Computations (TWBCs).

Event processing engines managing TWBCs must cope with an ever increasing number of event sources and continuously growing data rates. To keep up with this trend, processing engines must be able to manage *huge input data volumes*. Output of such computations are often used to take the best decision about some next action to be performed. Therefore, it is crucial to get these results as soon as possible, otherwise they are likely to become obsolete before they can be actually used. To cope with this requirement, processing engines must be *timely* in the production of their output.

Event processing engines can adopt two possible approaches for TWBCs with respect to the relationship between when events arrive and when they are elaborated. If events are processed as soon as they enter the engine, we talk about *online* event processing. Conversely, if events are first stored and then periodically processed in batches, we talk about *batch* event processing. This latter approach is usually preferred when timeliness requirements are not that strict, indeed an inner characteristic of batch processing resides in the delays to be paid in order to get updated results, because of its periodical nature. On the other hand, running computations every so often allows to cope with load spikes, failures and imbalances much more easily than the online approach does. Furthermore, a batch approach enables the decoupling of data loading and data elaboration, which provides higher flexibility to accommodate for possible distinct requirements.

Batch processing is heavily employed within business workflows of many medium to large companies for periodical ETL (Extract, Transform, Load) operations where large data sets produced daily up to hourly have to be moved, analyzed and archived so as to provide the proper means for enforcing specific business intelligence strategies. These scenarios are representative examples of the challenges and opportunities that the emerging BigData trend is fostering. Some well Known companies that are employing this kind of approach are

Oracle [4], Dell [5], MicroStrategy [6] and Cisco [7].

In this paper we focus on the batch approach and investigate which are its pros and cons in order to understand whether present batch-oriented computation frameworks can properly meet the previously introduced requirements for TWBCs. In particular, the possibility to increase the frequency of computations on sliding time windows is thoroughly checked into so as to get a better understanding about the class of use cases where a batch approach of this kind can be effectively employed. To this respect the contributions of this paper are:

- the definition of a simple model for batch processing in TWBCs that includes a set of important performance metrics. These metrics provide the basis for fundamental optimizations;
- an analysis of the impact of input data organization on these metrics, that shows how a smart subdivision of incoming events in data batches can help in maximizing performance;
- an instantiation of the model in the Hadoop framework accompanied by *ad-hoc* input data organization strategies that aim at reducing computation latency;
- and, finally, a prototype-based experimental evaluation that highlights strengths and weaknesses of the proposed strategies.

The rest of the paper is structured as follows: Section 2 discusses the related work, Section 3 introduces batch processing in TWBCs and our model, discussing the related performance metrics and two possible strategies for arranging input data; Section 4 describes the instantiation of the model in the Hadoop batch processing framework and an ad-hoc strategy for improving performances; Section 5 discusses the experimental evaluation results; finally, Section 6 concludes the paper.

2. Related Work

The developments in the area of distributed event processing happened during last decade have been based mostly on the concept of continuous queries, which run unceasingly over streams of events provided by external sources. These queries are compiled in a network of processing elements that can be distributed over available resources. Several projects have been on this line, although the structures used to model the compiled query are named differently. Among the most cited, we find InfoSphere [8, 9] (networks of InfoPipes), Aurora [10, 11] (networks of processing boxes), TelegraphCQ [12] (networks of dataflow modules), STREAM [13] (query plans composed by operators, queues and synopses), Borealis [14] (networks of query processors), and System S [15] (Event Processing Network (EPN) of Event Processing Agents (EPA)).

In this paper we adopt the jargon introduced by the latter. The reconfiguration of an EPN at runtime introduces several issues. The main one is the rebalancing of the load among nodes.

Shah et al. [16] define a dataflow operator called flux, which is integrated in an EPN and takes care of repartitioning stateful operators while the processing is running. Its limitations concern the dependance on configuration parameters that need to be tuned manually and the lack of fault tolerance mechanisms.

Gu et al. [17] propose a mechanism to process Multiway Windows Stream Joins (MWSJs) which distributes tuples to distinct nodes to allow for parallel processing. Their algorithm is specific for MWSJs. Xing et al. [18] describe an algorithm for placing operators such that no replacement is required at runtime. They deem that operators cannot be moved at all. Xing et al. [19] introduce a load distribution algorithm for minimizing latency and avoiding overloading by minimizing load variance and maximizing load correlation. Liu et al. [20] propose a dynamic load balancing operators for stateful algorithm, which spills state to disk or moves the operators to other nodes to resolve imbalances. Lakshmanan et al. [21] present a stratified approach where the EPN is partitioned horizontally in strata and operators can be moved within a single stratum only.

Stateful operators in a continuous query pose the question of addressing the problem of memory constraints. The most studied case is that of the joins over distinct event flows or data streams, which require the usage of some time or count based window in order to avoid maintaining the whole history of input data. Time windows cannot guarantee a consequent bound on required memory because of the variability of input event rate. Employing load shedding as a solution [22, 23, 24, 25] could not be feasible in several scenarios where the accuracy of the processing is a main requirement, for example decision support, intelligence or disaster recovery. In this case, the employment of some disk-based storage is required, as described in several works [26, 27, 28, 29] which however deal with the processing of finite data sets.

There exist some projects addressing the topic of distributed processing of data stored to secondary storage without employing continuous queries.

DataCutter [30] is a middleware which breaks down on-demand clients' requests into processing filters in charge of carrying out required computations. It has been devised to carry out complex processing over large distributed data sets stored to disk.

The MapReduce paradigm [31] implemented in Google and its open source implementation Hadoop [32] have received a great interest by the community and a lot of related projects [33, 34, 35]) have been developed adopting a similar approach.

Dryad [36] is a project developed by Microsoft which organizes the processing as a dataflow graph with computational vertices and communication channels. The computation is batch and can elaborate files stored to a distributed file system.

The possibility of employing a batch approach is touched on in [37], where it is proposed as an appropriate solution when the computation is too much slow compared to event arrival, so that executing the elaboration for each event doesn't allow to keep up with event rate. In this case events are buffered and computation runs periodically on the current batch.

3. Batch processing for Time Window Based Computations

3.1. Time Window Based Computations

Etzion and Niblett in [1] define an *event* as "an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain". Event processing is performed by feeding events in the form of streams in a processing engine. A stream is "a set of associated events", often "a temporally totally ordered set". The ordering within a stream is defined by a timestamp associated to each event. By elaborating groups of events the engine can output new events that represent the the result of its computation.

With the name *time window based computation* (TWBC) we refer to the elaboration of a set of events that happened within a specific time window. The length of the window and the way such length changes over time depend on the scenario of interest. We can have either a single fixed-length time period, or a fixed-length time period that repeats in regular fashion, or windows that are opened or closed by particular events in the event stream, or, finally, windows that are opened at regular intervals where each window is opened at a specified time after its predecessor [1]. In this paper we focus on this latter type of time windows.

A new window is started every ΔT time units, and here we assume that $\Delta T \leq T$[1]. If $\Delta T = T$ then a new window is opened whenever the current one is closed: at any time there is a single window opened and each event belongs to one and only one window. We refer to this case as *juxtaposed windows*. If $\Delta T < T$ then a new window is opened before the previous one is closed, so at any time t (at steady state) there are $n_{ow}(t)$ open windows where $\lfloor \frac{T}{\Delta T} \rfloor \leq n_{ow}(t) \leq \lceil \frac{T}{\Delta T} \rceil$. We refer to this case as *interleaved windows*. Each event e having t_e as timestamp belongs to $n_{ow}(t_e)$ different windows. We concentrate on the cases where $n_{ow}(t)$ is fixed to N, that is where T is a multiple of ΔT. Such assumption comes from the observation that often the length of the window is a multiple of the window period.

In general, given a time window TW_i which begins at time t_i and ends at $t_i + T$, we want to decrease the latency l_i between the end of TW_i and the availability of the related result R_i (see Figure 1). This latency depends on several distinct factors regarding both the time to produce the result itself and the synchronization between the end of the window and the beginning of the computation, as will be shown in next section.

3.2. Batch processing

TWBCs can be process using either an *online* or a *batch* approach. With the former approach events are kept in memory and processed as soon as they

[1]Cases where $\Delta T > T$, that imply a voluntary loss of events in the periods of length $\Delta T - T$ that occur between the end of a window and the start of the next one, are of little interest and thus ignored in this paper.

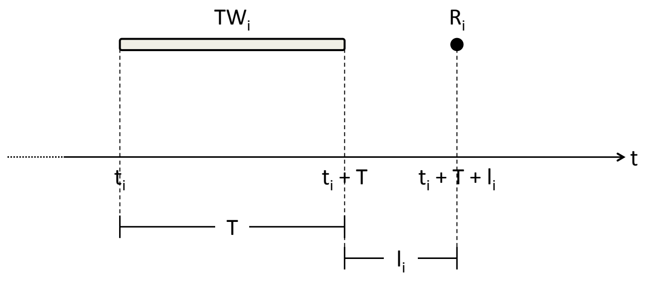

Figure 1: Placement in time of a time window and the related result.

enter in the system in order to minimize the output delay. Conversely, with batch processing incoming events are first stored in a secondary storage (e.g. in a disk-based database or, more commonly, in a file system) and then periodically processed in batches. The frequency of these computations depends on application specific requirements and on the feasibility of running many concurrent computations. While the online approach allows for continuous output, periodical batch computations can only produce periodical output.

Batch processing provide some advantages with respect to online solutions that make it suited to several application scenarios. The amount of data that must be analyzed within a time window, being a function of both the window size and the event rate, can easily grow to huge amount. Storing this data in secondary storage instead of main memory can allow to support applications with massive data rates and large time windows with simpler and less expensive computing infrastructures. Moreover, systems based on the online approach process events as they enter the system; this can easily limit system scalability in applications where processing is computationally intensive and data rates are large. Batch processing, on the other side, defer computation to the end of a time window; incoming events are directly stored in the secondary storage thus allowing very large input rates.

Existing batch processing solutions can be adopted to perform TWBCs. We found that the class of technologies that naturally fits what we need is the one described in Section 2 that includes the projects focused on distributed batch processing of data kept on secondary storage [30, 31, 32, 33, 34, 35, 36]. All them allow to execute distributed complex computations on huge volumes of data. Such data is organized in large files partitioned over available storage nodes. File systems are generally preferred to DBMSs because the operations we need to execute on input data are very simple and don't require most of the high level functionalities provided by today's DBMSs. Events pertaining to each window are stored in files which become the input of the processing engines in charge of producing the output for such time windows.

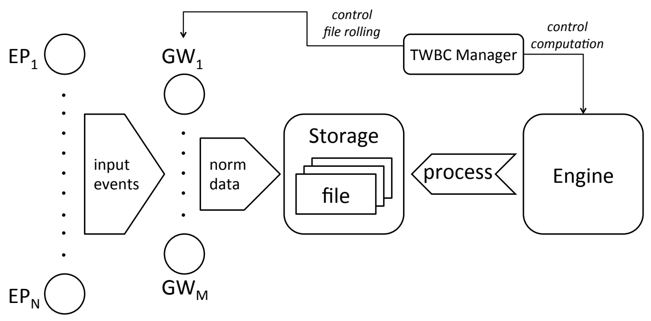

Figure 2: High-level architecture of a batch-oriented event processing system.

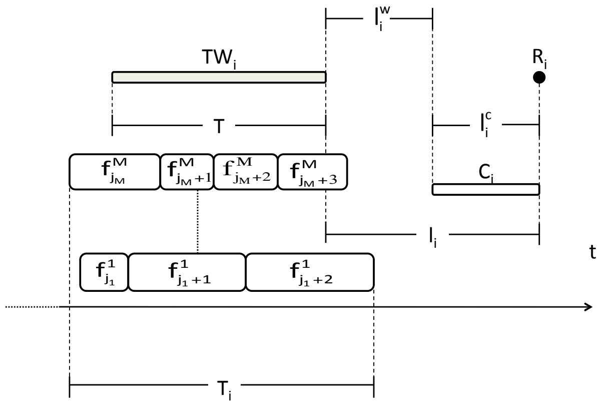

Figure 3: Performance metrics in batch-oriented TWBCs.

3.3. Computation Model

A high level architecture which highlights the interactions between *event producers* (EPs) and the processing engine is shown in Figure 2. Several EPs provide *input events* to a set of *gateways* (GWs) in charge of *normalizing* and storing them in the *storage*. Data in the storage is organized in *files*. A new event is appended to an open file f. Upon certain conditions, a file f is closed and a new file f' is created. We refer to this operation as *file rolling*. At any time, there can be several files that are being written to by distinct GWs and the temporal coverages of different files can overlap. In Figure 3 a time window TW_i is shown together with the temporal coverages of the files containing its events ($f_{j_1}, \cdots, f_{j_1+2}$ to $f_{j_M}, \cdots, f_{j_M+3}$) and the position in time of the related computation C_i. The trigger of file rollings is performed by a *TWBC Manager*, which is also responsible of starting the computations by properly controlling the processing *engine*. The engine simply runs the desired computation over all the files intersecting the time window of interest. The output of the computation is not shown here because it is not relevant for our purposes.

A crucial issue in this setting is represented by the synchronization between events and time windows. If event timestamps were set by the EPs, an accurate synchronization would be impossible as the EPs' clocks and the TWBC Manager's clock in general present non negligible skews. Various synchronization techniques (e.g. [38]) can be used to mitigate this problem. Event processing systems usually rely on their internal clocks, so an event is considered included within a certain time window according to the time such event enters the engine. In our model, we assume that events are timestamped by GWs as they are written to file. Note that this solution does not completely solve the synchronization issue but effectively alleviates it as synchronizing multiple GWs, components usually deployed in a controlled environment within a single administrative domain, is a reasonably easy to solve task. In general, synchronization guarantees depend on the employed technologies and the type and extent of the deployment. Whether such guarantees allow for a properly accurate computation depends on the specific application.

The latency l_i, that separates the end of the time window TW_i from the output of the result R_i, is constituted by two distinct temporal components (see Figure 3):

- the *wait latency* l_i^w representing the time between the end of the time window TW_i and the beginning of the computation C_i;
- the *computation latency* l_i^c representing the time it takes for the *computation* C_i to complete.

Note that that nothing prevents C_i from beginning before the file containing the last of event in TW_i is closed, assuming that the chosen technologies allow to elaborate files while they are being written to. In that case, also the wait latency doesn't depend on when such last file is closed. Figure 3 represents a simple case where the computation begins after the closure of such last file.

In order to reduce the wait latency l_i^w, the computation should be started as soon as a time window ends. By making the TWBC Manager in charge of determining when time windows begin and end, we can minimize it.

The reduction of the computation latency is more complex at this level of abstraction. As shown in Figure 3, the computation of TW_i requires to process all the files that contain events in TW_i. These files cover a time range of length T_i, where in general $T_i \geq T$. It is up to the computation itself to filter out the events outside the time window. Since distinct files can overlap in time and also be partitioned without taking into account the temporal order of contained events, the processing engine has to read the content of all these files in order to decide which events must be processed. If we assume that (i) the length of the computation increases as the size of data read from the storage grows (IO bound computation) and that (ii) the size of stored data grows with the length of the time interval it covers, then we can conclude that a possible way of reducing the computation latency is to include in the processing all and only the events that are included in the time window of interest. A convenient metric able to capture this aspect is the *time efficiency*, defined as the ratio T/T_i, which represents an approximation of the fraction of processed events that actually are within the time window. The approximation is based on the assumption that event rate is stable during the period T. The time ratio is a real number in the range $[0, 1]$ where 1 represents the optimum, i.e. only events in TW_i are processed.

3.4. Input data organization strategies

We can define proper strategies for organizing input data in files with the aim of optimizing the metrics introduced in the previous section. A strategy defines when file rollings are triggered and thus determines the positioning of events in file and how these files are organized for batch processing. We first present the strategy for juxtaposed windows ($\Delta T = T$) and then that for interleaved windows ($T = N \cdot \Delta T$).

Juxtaposed Windows. The case where $\Delta T = T$ is the simplest one and the strategy we propose for it is straightforward but yet allows for the optimization of all the metrics.

Figure 4 illustrates this strategy. We assume there are M GWs, each writing events to distinct files. During time window TW_i, the GW_g writes its incoming events to file f_i^g. When TW_i ends, the TWBC Manager issues a file rolling to all the GWs and tells the Engine to start the computation. This picture represents the collocation of time windows (TW_i), temporal coverage of files (f_i^g) and computations (C_i) on a timeline that spans a period of four consecutive windows. All the events related to time window TW_i are stored in files f_i^x, where $x = 1...M$. At time t_i, files $f_i^1, \cdots, f_i^M$ are opened. At time $t_{i+1} = t_i + T$, all these files are closed and the computation C_i for TW_i is started. The computation ends at time $t_{i+1} + l_i^c$. The figure is simplified so that it seems that the ideal relation $l_i = l_i^c$ holds. In practice, there is some operational delay between the end of TW_i and the beginning of C_i (that is l_i^w). If the Engine cannot process files while they are written to, some degree of coordination is necessary between

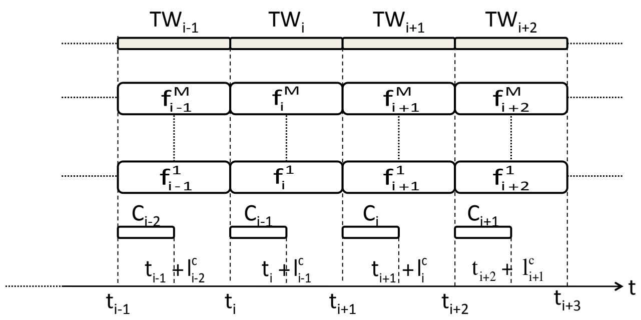

Figure 4: Strategy for juxtaposed time windows.

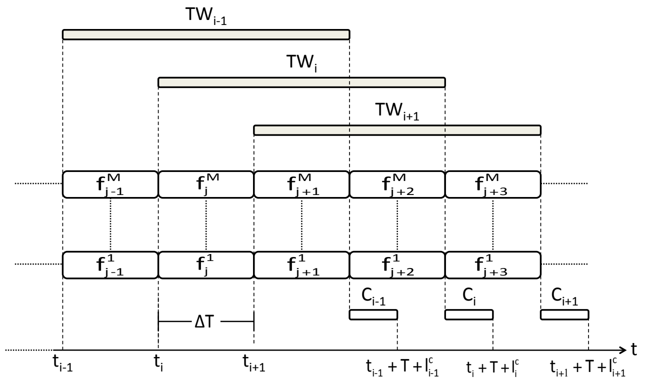

Figure 5: Strategy for interleaved time windows.

the TWBC Manager and the GWs in order to start the computation only after files $f_i^1, \cdots, f_i^M$ are closed, but these are implementation details which depend on the technologies employed. Summing up, we can state that (i) the wait latency is minimized and (ii) the time efficiency is optimized ($T = T_i$).

In Figure 4 we are implicitly assuming that $l_i \leq \Delta T$, in order to avoid that computations execute concurrently. This is not a mandatory requirement, but running each computation in isolation allows to use the whole computational power provided by the underlying infrastructure, which possibly means that the latency can be further minimized.

Interleaved Windows. In reference to Section 3.1, when $\Delta T < T$ we talk about *interleaved* time windows. In particular, we concentrate on the case $T = N \cdot \Delta T$, where $N \in \mathbb{N}$, $N > 1$. Also In this case we can define a simple strategy, which optimizes all the metrics and can be considered as a generalization of the strategy described in the previous section.

Figure 5 shows that each time window is covered by $N \cdot M$ distinct files. In this figure we have reported the time windows, the temporal coverage of files and the computations related only to three consecutive time windows in a setting where $\Delta T = T/3$ ($N = 3$). We have not included occurrences related to other time windows that actually happen in the time span shown in the figure in order to avoid complicating the figure itself. Consecutive time windows begin with a delay of ΔT, so $t_{i+1} = t_i + \Delta T$. Assuming that TW_0 and the temporal coverage of f_0^x, for $x = 1...M$, begin at the same instant t_0, and making each GW use one file for each period spanning ΔT time units, we have that TW_i is covered by files f_i^x to f_{i+N-1}^x, for $x = 1...M$. Since T is a multiple of ΔT, we obtain an optimum time coverage of the periods of length T, which means that (i) the wait latency is minimized and (ii) the time efficiency is optimized ($T = T_i$).

An interesting aspect of interleaved windows is that the constraint $l_i \leq \Delta T$ becomes much more difficult to comply with respect to juxtaposed windows, because the amount of data to be processed is the same, but the time available for the computation is $1/N$.

4. Implementation with Hadoop

Starting from the model introduced in the previous section we implemented a batch processing engine for TWBCs using Hadoop (Section 4.1) as processing engine and HDFS (Section 4.2) for input data storage. In this section we analyze which additional factors have to be taken into account for the optimization of performance metrics with this specific setup.

4.1. Hadoop and the MapReduce paradigm

Apache Hadoop [32] is a Java-based open source framework that allows the distributed processing of large data sets across clusters of computers. It is based on the MapReduce programming model [31] which lets a user define the desired computation in terms of *map* and *reduce* functions. The underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks.

A Hadoop computation is called *job* and its work is decomposed in *map* tasks and *reduce* tasks. Input data is fed to a variable set of map tasks that perform part of the overall elaboration and produce an halfway output.

Such halfway output is then computed by a fixed number (cluster-wise configuration parameter) of reduce tasks for the provision of the final output. In this case, the EPN of the computation simply consists in a two stage digraph where map tasks are at the first stage and reduce tasks at the second one.

A Hadoop deployment consists of (i) a single *JobTracker* in charge of creating and scheduling tasks and monitoring job progress and (ii) a set of *TaskTrackers* which execute the tasks allocated by the JobTracker and report to it several status information.

One of the key characteristics of Hadoop is its ability to allocate map tasks where input data are placed, so as to minimize data transfers and decrease the overall job latency. Such locality awareness fits very well with the need of timely and smartly reconfiguring the allocation of maps (EPAs) to resources since before starting each job a proper task placement is enforced.

4.2. Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [39] is the default storage used by Hadoop and has been designed to properly support read/write access patterns typical of Hadoop jobs. Data in HDFS is organized in files and directories. Like in a standard file system, each file is broken into block-sized (64 MB by default) chunks, which are stored as independent pieces to simplify the storage subsystem and to fit well with replication for providing fault tolerance and availability.

An HDFS installation includes (i) a single *NameNode* which manages the whole namespace of stored data and controls how data is spread over available resources and (ii) a set of *DataNodes* responsible of storing the actual data.

The default data replication factor is 3, which means that in the cluster there are in total 3 replicas for each block. In our implementation we set the replication factor to 1 because we want to be as fast as possible in storing events, and replicating data would consume too much time. This choice penalizes the fault tolerance of our system but in the scenarios of interest it is more important to keep up with input data rates than loosing some input data because of failures of cluster nodes.

In our implementation, the GWs are in charge of (i) receiving events from the EPs and (ii) converting them in a format suitable for being written to HDFS files. Data in a HDFS file cannot be read by map tasks until such file is closed, so some coordination is needed to ensure that required files have been closed before starting a job.

Although in the scenarios of interest we want to deal with several EPs providing events to a set of GWs, our current implementation supports a single GW. This has no impact on our evaluation of a strategy because the metrics we want to optimize are independent from the number of GWs.

4.3. Performance Metrics Optimization

In our implementation, we developed a Java application which encapsulates the functionalities of both a GW and the TWBC Manager. Such application is thus in charge of (references to items of Figure 2 are reported in parentheses)

- receiving input events (input events)
- converting them in ASCII format and writing them to a HDFS file (norm data)
- deciding when to execute a file rolling (control file rolling)
- triggering the execution of Hadoop jobs (control computation)

As explained in Section 3.3, a key aspect for the optimization of performance metrics is the synchronization between time windows and computations. In our implementation based on Hadoop, a coordination between file rollings and jobs executions is also required. By assigning to the Java application the duties of both the GW and the TWBC Manager, we have the possibility of enforcing such synchronization and minimizing the wait latency. For what concerns the computation latency, the time efficiency is important because Hadoop jobs are known to be IO bound, as shown by Wlodarczyk et al. [40].

Besides the synchronization issues and the optimization of time efficiency, a strategy has further influence on the computation latency. The general problem of data acquisition in Hadoop has been studied in part by Jia et al. [41]. The important point emerging from this papers is that Hadoop works much more better with a small number of large files with respect to a large number of small files. Hadoop divides the input of a job into fixed-size pieces called *input splits*, then creates one map task for each split, which runs the user-defined map function for each record in the split. In a Hadoop cluster configured with a block size B, for a file of size S Hadoop considers $\lceil S/B \rceil$ splits [2]. Since a limited number of map tasks can run concurrently and since each map task involves a management overhead, the overall performance would improve by using files having size B. In this way, in fact, each map task works on a chunk of size B and the number of allocated map tasks is minimized.

More formally, we consider a data set of total size D which is organized in N files of size d_i $(i = 1...N)$, such that

$$\sum_{i=1}^{N} d_i = D \tag{1}$$

The number of splits required for such data is

$$S^{(N)} = \sum_{i=1}^{N} \left\lceil \frac{d_i}{B} \right\rceil \tag{2}$$

where $S^{(l)}$ indicates the number of splits using l files. To show how the number of splits decreases with the number of files, we consider what would happen if we used a single file of size D.

$$S^{(1)} = \left\lceil \frac{D}{B} \right\rceil \tag{3}$$

The comparison between the terms in equations (2) and (3) requires to express both D and d_i as a function of B. Let $b = D\ div\ B$ and $r = D\ mod\ B$, where div and mod are the quotient and the remainder of the division, respectively. Then we can write

$$D = b \cdot B + r \tag{4}$$

[2]This relation has been obtained using default values for the configuration parameters that control the way input splits are computed: *minimumSize* and *maximumSize*; they constraint the minimum and maximum size of a split, respectively. The formula used to compute the split size is $max(minimumSize, min(maximumSize, B))$, where B is the block size [32].

where $r < B$. Similarly, for $i = 1, \cdots, N$ we can write

$$d_i = b_i \cdot B + r_i \tag{5}$$

where $\forall i,\ b_i = d_i\ div\ B,\ r_i = d_i\ mod\ B,\ r_i < B$. Replacing equations (4) and (5) into (1) we have

$$b \cdot B + r = B \cdot \sum_{i=1}^{N} b_i + \sum_{i=1}^{N} r_i \tag{6}$$

We state that

$$b \geq \sum_{i=1}^{N} b_i \tag{7}$$

which in turn by (6) implies $r \leq \sum_{i=1}^{N} r_i$, and we prove it by contradiction as follows. Let us assume that $b < \sum_{i=1}^{N} b_i$, so we can write

$$b = \sum_{i=1}^{N} b_i - a \tag{8}$$

where $a \in \mathbb{N}$, $a > 0$. Replacing (8) in (6) we have

$$\begin{aligned} (\sum_{i=1}^{N} b_i - a) \cdot B + r &= B \cdot \sum_{i=1}^{N} b_i + \sum_{i=1}^{N} r_i \Longrightarrow \\ r &= a \cdot B + \sum_{i=1}^{N} r_i \Longrightarrow \\ r &\geq B \end{aligned}$$

which is in contradiction with the definition of r. Replacing (4) and (5) in (3) and (2) respectively,

we can write

$$S^{(1)} = b + 1 \tag{9}$$

$$S^{(N)} \leq N + \sum_{i=1}^{N} b_i \tag{10}$$

The $\leq$ relation in (10) comes from the fact that $r_i = 0$ can hold for some i. Subtracting (9) from (10) we have

$$S^{(N)} - S^{(1)} \leq N - 1 + (\sum_{i=1}^{N} b_i - b) \leq N - 1 \tag{11}$$

where the last $\leq$ relation comes from (7).

Equation (11) expresses the fact that using N files instead of one can make the number of splits increase by up to $N-1$. The worst case is when $b = \sum_{i+1}^{N} b_i$.

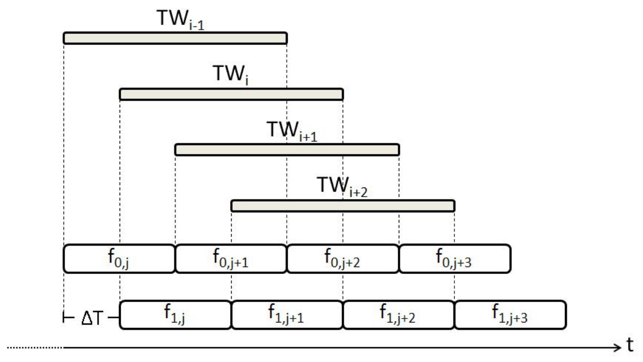

Figure 6: Rolling Strategy for interleaved time windows with Multi Files Flows.

In a Hadoop cluster with L TaskTrackers configured with MS available map slots each, at most $L \cdot \mathrm{MS}$ map tasks can run at the same time. Depending on the value of D, it could be impossible to run all the map tasks in a single round. Indeed, if $D > B \cdot L \cdot \mathrm{MS}$, at least $L \cdot \mathrm{MS} + 1$ map tasks are required, regardless of how data is organized in files. However, the goal remains to arrange input data so that resulting map tasks can be executed in the minimum number of rounds, and at the same time ensuring that the load is fairly distributed among TaskTrackers. The last point is important since the reducer tasks start their work whenever all map tasks are completed, so load imbalances can make a TaskTracker employ more time than the others to complete the map phase, which in turn causes the beginning of reduce phase to delay.

As discussed in [42], the best solution for loading this kind of input data to HDFS is employing some technology like Chukwa, that is something in charge of collecting external streams and writing them to HDFS files. While Chukwa proves to be very useful when there are several geographically distributed GWs, simpler scenarios with a single local GW can be managed by a single application like ours which keeps writing data to HDFS as new events arrive.

Multi Files Flows. What emerged so far is that in a Hadoop based implementation a possible problem of the interleaved window strategy described in Section 3.4 is that N can be very large and involve high computation latencies. In Section 5 we will see some scenarios where this actually happens. Trivially using larger files doesn't work. In reference to Figure 5, if for example we set each file to cover $2 \cdot \Delta T$, then for time windows TW_j with j even we would have a wait latency of ΔT (due to the fact that a Hadoop job cannot read open files), which is unacceptable. The synchrony between time windows, HDFS files rollings and Hadoop jobs has to be kept in order to optimize wait latency. This also implies that each time window still has to be perfectly fit by HDFS files, that is the time efficiency has to be optimal.

The solution we propose consists in replicating data so as to (i) use a lower number of bigger files and (ii) provide each time window with the HDFS files required to have a perfect coverage. As shown in Figure 6, data is replicated K times ($K = 2$ in the figure) using K distinct *files flows*. K is required to be a divisor of N, that is $N \ mod \ K = 0$. Since we have a single GW, we don't need to specify which GW writes which file. In this case we use the notation $f_{w,j}$ to indicate the j-th HDFS file of the w-th files flow. With K files flows, each file can cover a time interval of $K \cdot \Delta T$ and the number of files required to cover a time window becomes N/K. Files in flow w are closed with a delay of ΔT with respect to files in flow $w - 1$. In this way, each time window TW_i has a perfect coverage with the files $f_{w,j}$ to $f_{w,j+\frac{N}{K}-1}$ where $w = i \ mod \ k$ and $j = i \ div \ k$. In reference to Figure 6, TW_{i-1} is covered by $f_{0,j}$ and $f_{0,j+1}$ while TW_i is covered by $f_{1,j}$ and $f_{1,j+1}$ and so on.

Compared to the solution for interleaved windows reported in Section 3.4, this strategy allows to decrease the number of required files by a factor K at the cost of replicating input data K times.

5. Experimental Evaluation

Our experimental evaluations are aimed at validating the model introduced so far, giving a hint about the exhibited computation latencies and comparing the two strategies defined for interleaved time windows. We didn't evaluate the strategy for juxtaposed windows because it can be considered as a special case of interleaved windows with $N = 1$, and the evaluations of these strategies become interesting when N is large.

We carried out several evaluations simulating a scenario in which a fictional traffic monitoring application is requested to produce statistics every minute ($\Delta T = 1$ minute) about the packets observed in the last hour ($T = 1$ hour, $N = 60$). We vary input packet rate and observe the latency of the jobs. For each fixed packet rate we measured the latency of the first 12 jobs. Latencies of subsequent jobs did not reveal any further insights on the performance of the system and are thus not shown. As a warm up phase, we let the system load data for 1 hour before starting the first job. In this way each job actually works on a time window of 1 hour.

The equations defined in Section 4.3 can be simplified by introducing some assumptions based on the properties of our evaluations. Since packet rate is kept fixed for each run, the size of the input data stored for each ΔT can be considered constant, so we can assume that

$$\forall i, \ d_i = d \tag{12}$$

With the largest packet rate we used, the size of the input data stored for each ΔT was at most 40 MB, which is less than the size of a block (64 MB), so we can also assume

$$d < B \tag{13}$$

We can use (12) and (13) to rewrite (1), (2) and (3) as follows, respectively

$$D = N \cdot d$$

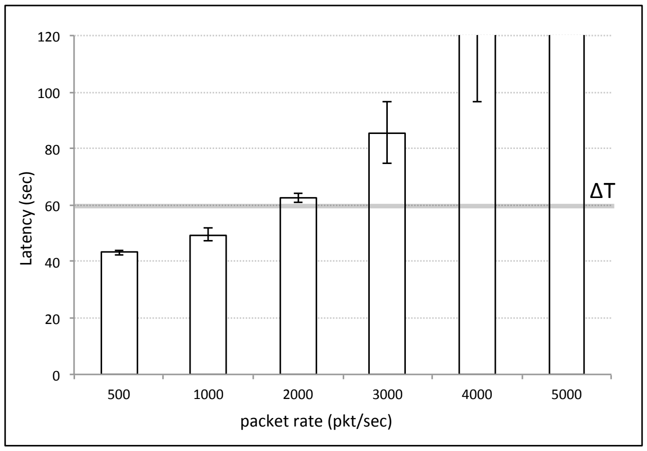

Figure 7: Average latencies and standard deviations for the basic interleaved windows strategy.

$$S^{(N)} = \sum_{i=1}^{N} \lceil \frac{d}{B} \rceil = N$$

$$S^{(1)} = \lceil \frac{N \cdot d}{B} \rceil$$

We use a simple TCP traffic data analysis computation, where packets are filtered out on the basis of the value of TCP flags and are then written to an output HDFS file. The filter we considered is such that almost any packet is dropped. The number of reducers is set to 1. This kind of computation is executed by a very simple event processing network with a set of map tasks which execute IO-bound work and send filtered data to a single reduce task in charge of writing the resulting data to a file. The choice of using this kind of filter has been driven by the observation that different rolling strategies can only impact the performance of map tasks, so, in order to better observe their effectiveness, we consider a computation where the contribution of reduce tasks to the latency is negligible.

We setup a Hadoop deployment with 7 nodes. In one node we placed the JobTracker and the NameNode. In each of the other nodes we place a TaskTracker and a DataNode, so as to enable the JobTracker to allocate map tasks where data actually is and minimize data transfers. We used another node where we installed our Java-based application. Each node was a Virtual Machine (VM) running Ubuntu 10.04 and equipped with 2x2800 MHz CPUs, 3 GB of RAM and 15 GB of disk storage. The networking infrastructure was based on a 10 Gbit LAN.

Basic Interleaved Windows Strategy. The evaluations for the basic interleaved windows strategy (as introduced in Section 3.4) are aimed at showing (i) how the engine is able to keep up with increasing input data rates and (ii) what happens

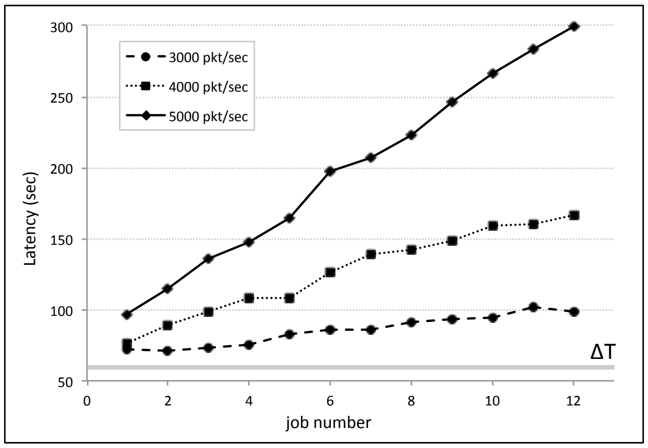

Figure 8: Observed latencies when packet rate is greater than 2000 pkt/sec.

when the constraint $l_i \leq \Delta T$ is violated, i.e. the time needed for computing over a time window is greater than the time-span between the start of two subsequent windows. The average latencies and related standard deviations registered in these experiments are reported in Figure 7. These results confirm that Hadoop jobs are IO bound: as the packet rate grows, input data size grows as well and the latencies become larger. When packet rate is set at 3000 pkt/sec the standard deviation is quite large (10.86 sec.), which denotes a high variability in the observed latencies. The latency values for packet rates 4000 and 5000 pkt/sec are purposely left out of scale to highlight how such variability grows with input data size.

We say that there is *stability* when the computation latency doesn't keep growing job after job. It is to note that when packet rate is set at 2000 pkt/sec the average latency is 63 seconds, i.e. slightly greater than ΔT, stability is nevertheless preserved. This happens because the overlapping between consecutive jobs is quite small and unable to make the engine run out of available resources. With packet rates 3000 pkt/sec or greater we observe that stability doesn't hold anymore, as shown in Figure 8, where it is made clear that the latencies keep increasing as new jobs are executed. In this case the initial latencies are more than 10 seconds larger than ΔT and subsequent jobs are progressively delayed. The consequence is that the engine becomes unable to deliver acceptable performances.

Multi Files Flows Strategy. The evaluation of the Multi Files Flows Rolling Strategy is aimed at showing how decreasing the number of files positively impacts on the computation latency. We used again the same packet rates reported in Section 5 and we tracked average latencies and related standard deviations with K (replication factor) varying from 1 to 6. We didn't tested larger values for K because it would have required too much space on disk and generated too much overhead network traffic due to the high level of data replication.

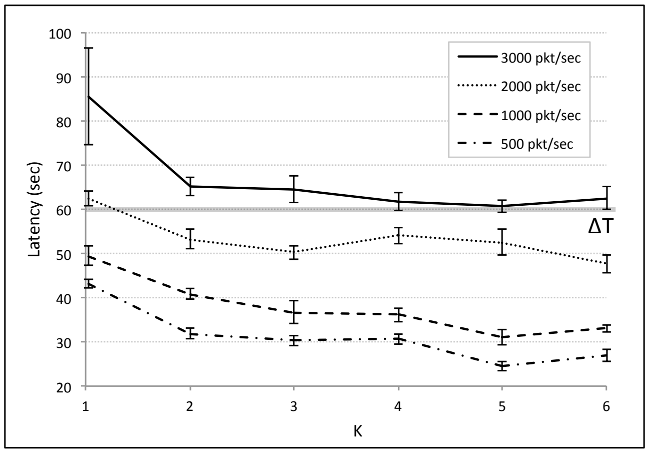

Figure 9: Average latencies varying both input packet rate and K.

Results are reported in Figure 9. Each curve represents the average latency for a specific packet rate with different values of K with error bars used to plot the standard deviations. Introducing data replication, that is moving from $K = 1$ to $K = 2$, makes stability hold even with a packet rate set at 3000 pkt/sec. This can be noted by looking at the difference between the standard deviations for $K = 1$ and $K = 2$. With reference to the equations derived before in this Section, using a specific K we can organize input data in N/K files and the resulting number of splits is

$$S^{(N/K)} = \frac{N}{K} \cdot \lceil \frac{K \cdot d}{B} \rceil \tag{14}$$

The relations between packet rates, K, number of splits and file sizes are reported in Table 1. Let us define the set div_N as the set of divisors of N. The value of K which minimizes the number of splits is

$$K_{opt} = \operatorname*{argmin}_{K \in div_N} S^{(N/K)} \tag{15}$$

Figure 10 plots the distance in percentage from the optimum number of splits as K varies for some distinct ratios d/B, so as to provide results that are oblivious from the specific dimensions that can come into play in individual scenarios. As such picture makes evident, just using small values for K allows to obtain numbers that are not far from the minimums, which means that a convenient tradeoff between storage requirements and resulting mappers count can be firstly foreseen and then achieved.

Figure 11 shows how the number of splits varies in function of K (a logarithmic scale is used) for several packet rates. As already stated, the number of splits commonly decreases with the increase of K, except for some unlucky cases where the dimension of resulting files doesn't fit well with the block size

K	500 p/s	1000 p/s	2000 p/s	3000 p/s
1 60 files	60 split 4 MB	60 split 8 MB	60 split 16 MB	60 split 24 MB
2 30 files	30 split 8 MB	30 split 16 MB	30 split 32 MB	30 split 48 MB
3 20 files	20 split 12 MB	20 split 24 MB	20 split 48 MB	40 split 72 MB
4 15 files	15 split 16 MB	15 split 32 MB	15 split 64 MB	30 split 96 MB
5 12 files	12 split 20 MB	12 split 40 MB	24 split 80 MB	24 split 120 MB
6 10 files	10 split 24 MB	10 split 48 MB	20 split 96 MB	30 split 144 MB

Table 1: Number of splits and file sizes as packet rate and K change.

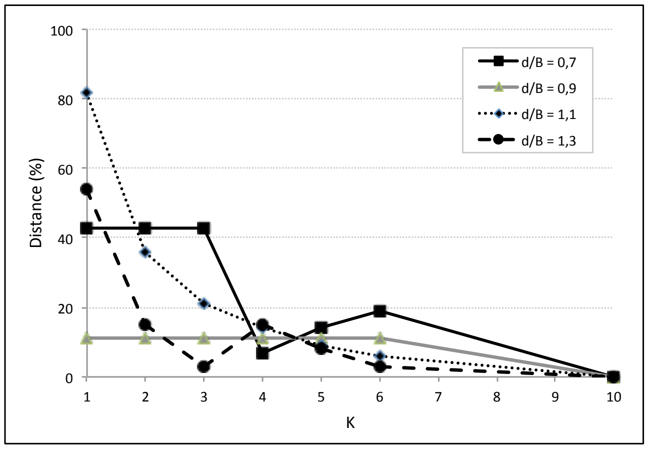

Figure 10: Distance (%) from the optimum for the number of splits in function of K, for distinct ratios d/B.

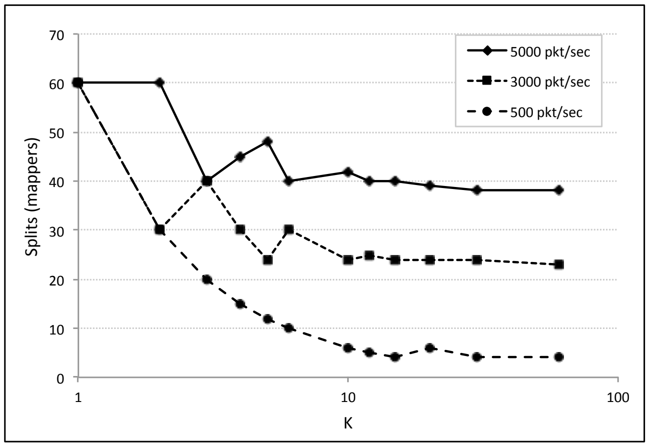

Figure 11: Number of splits in function of K for some packet rates.

B, but we will see soon that this is not necessarily a problem. With large packet rates, obtaining a near optimum number of splits doesn't require the use of large values of K. For example, when packet rate is 5000 pkt/sec and $K = 3$ there are 40 splits while the optimum is 38. When packet rate is 3000 pkt/sec, $K = 5$ allows to have 24 splits, that is only 1 more than the optimum.

Optimizing the number of splits with lower packet rates require larger values of K. Indeed with 500 pkt/sec a good result can be achieved setting $K = 10$, which entails 6 splits while the optimum is 4. This trend mainly depends on the fact that when the packet rate is small the size of files is small as well, and several files can be merged within a single split, that is the decrease of the number of files coincides with the decrease of the number of splits. Conversely, when the packet rate is large merging files is not likely to decrease the number of splits, because the size of merged files is probably greater than the block size. For example, when packet rate is 5000 pkt/sec, $d = 40$ MB and 60 splits are required. By setting $K = 2$, the size of a single file becomes 80 MB, which requires 2 splits, so even if the number of files has been halved, the total number of splits remains the same.

For what concerns the latencies reported in Figure 9, for packet rates 500 and 1000 pkt/sec the situation is quite easy to analyze because $K \cdot d \leq B$ for $K \leq 6$, so the (14) becomes $S^{(N/K)} = N/K$. This implies that the number of splits decreases monotonically as K is increased, which in turn makes the average latency decrease as well, except for the case $K = 6$ where a small growth occurs. The reason of such worsening is the imbalance of the allocation of map tasks to TaskTrackers, as previously introduced in Section 4.3. Looking at Table 1, when $K = 5$ there are 12 splits with the same size, which allows for a fair allocation of 2 map tasks for each TaskTracker (with 6 TaskTrackers in the cluster). Each TaskTracker can execute such tasks in parallel because there are 2 available slots for map tasks for each TaskTracker. When $K = 6$ we have instead 10 equal-sized splits, which makes a fair allocation impossible, in fact 2 slots remain not allocated. In this case, each TaskTracker runs in parallel 1 or 2 map tasks that work on splits larger than those of the case $K = 5$, which causes the small increase in the overall latency.

The case for packet rate 2000 pkt/sec is a little bit more complex. For K up to 3, everything is fine and the average latency keeps decreasing. When $K = 4$ latency grows by 4 seconds (8%) despite the number of splits is minimized (15 splits is indeed the minimum for $K \leq 6$). The reason again is the imbalance in the allocation of map tasks to TaskTrackers. The first 12 map tasks can run in parallel, while the remaining 3 have to run after the first batch completes. Furthermore, these 3 tasks work on block-sized splits, which makes their completion time not marginal. Therefore, the critical path in the map phase is the sequence of two map tasks each working on a block-sized split (64 MB + 64 MB). A possible solution would consist in reorganizing the last 3 splits so as to fairly spread the load among all the TaskTrackers, but such feature is not provided by Hadoop. When $K = 5$ the number of splits raises to 24 but the average latency is 1 second lower. Each of the 12 files has size 80 MB, so is divided in 2 splits, one with size 64 MB and the other 16 MB. In total there are 12 block-sized splits and 12 splits of 16 MB. The allocation in this case can be very fair because each map slot can be assigned in sequence to two map tasks, the first works on a 64 MB split and the other on 16 MB one, which would shorten the critical path (64 MB + 16 MB). In reality, the locality awareness of Hadoop (see Section 4.1) affects such a theoretically optimal allocation strategy because the placement of some map tasks is driven by the actual position of the input data they need to work on. Depending on how the blocks of input data are distributed over the DataNodes, this can entail an allocation where 2 tasks working on a 64 MB split are assigned to the same slot, making the critical path equal to the one for $K = 4$. Furthermore, the presence of two classes of splits having so different sizes (64 MB vs 16 MB) can make some slots become free much sooner than others, forcing the JobTracker to assign them available splits without concerning too much about whether the new allocation could cause future imbalances. Among the 12 jobs we ran for $K = 5$, for 7 of them we observed a 64 MB + 64 MB critical path while for the other 5 the critical path was 64 MB + 16 MB + 16 MB. Such variability is also highlighted by the higher value of the standard deviation. In this case, the potential benefits of an optimal allocation are offset by an unlucky distribution of blocks over the cluster, which, on average, makes the performance improve only marginally. When $K = 6$ the average latency is improved by 5 seconds (9%) with respect to $K = 5$. Such an enhancement is due to (i) the reduction of splits and (ii) the reduction of the difference between the sizes of the splits, which helps to prevent allocations resulting in imbalances. Indeed, each file is 96 MB and is divided into 2 splits of 64 MB and 32 MB, which is much less likely to cause the imbalances we observed for $K = 5$. All the 12 jobs we ran exhibited a critical path of 64 MB + 32 MB.

For packet rate 3000 pkt/sec we get stability using $K = 2$ and we registered a critical path of 48 MB + 48 MB + 48 MB. Setting $K = 3$, the critical path reduces to 64 MB + 64 MB + 8 MB but the great difference between split sizes causes imbalances which make the latency improve negligibly and the standard deviation increase. When $K = 4$, in most of the jobs we observed a critical path of 64 MB + 64 MB while in a few we reported 64 MB + 64 MB + 32 MB, but the reduced difference between the splits and the lower number of mappers (10 less) entails a decrease of the latency of 3 seconds. Going up to $K = 5$, the situation improves further because we get no imbalances thanks to the very small variation between split sizes (64 MB vs 56 MB) and the number of map tasks allows for an optimal allocation to the TaskTrackers. Finally, using $K = 6$, a small worsening (2 seconds) is noticed due the increment of both the difference between split sizes (64 MB vs 16 MB) and the number of the map tasks, which together entails for a worse allocation to TaskTrackers.

The table does not report the results for packet rates 4000 and 5000 pkt/sec because we didn't manage to make stability hold using $K \leq 6$.

Figure 9 gives evidence that major improvements are achieved with small values of K (2 or 3) and that larger settings of K don't provide relevant enhancements. What we can get from these evaluations is that we can apply the Multi Files Flows Rolling Strategy for successfully decreasing the computation latency at the price of replicating data with a reasonable replication factor.

6. Conclusions

Coping with today's event streams is getting more and more challenging because they keep increasing both in number and rate. Solutions adopting an online approach exist but several complexities arise when they have to manage on-the-fly reconfiguration to adapt to input evolution. From an high level point of view, a batch approach is able to provide opportunities to major such complexities, provided that the latencies it exhibits are in line with application requirements. The possibility to place computation near to the data to process allows to decrease data transfers and to consequently improve performances. Its periodic nature enables to properly tune the frequency of computations on the basis of application specific requirements.

In this context we introduced a model for batch processing in time window based computation where the importance of some metrics (wait latency, computation latency and time efficiency) for performance optimization is highlighted. The model underlines the importance of properly organizing input data in files in order to reduce latencies. We instantiated the model using the Hadoop distributed computing platform and evaluated the performance of a simple event processing application using different input organization strategies. The results show how several factors strongly impacts the feasibility of batch processing for TWBC, but also outlines that this is a viable solution in important application scenarios.

In that regard, the batch approach we propose can be employed in *collaborative environment*, which represent an interesting emerging paradigm in the field of cloud computing. A collaborative environment consists of a set of participants willing to share

- their own data, so as to collectively take advantage from the availability of much more data to extract useful information from;
- their own resources, in order to provide required computational, network and disk power without turning to any third party.

The advantages and the obstacles of collaborative approaches are deeply explored in [43], where the focus is mainly on the financial context. Our batch oriented Hadoop based solution can be applied in such context by installing Hadoop and HDFS software on the resources provided by the participants.

An additional key functionality, that is required to cope with the variable nature of input event streams, is the ability to seamlessly add and remove nodes from the cluster so as to properly scale in and out as input rates change. Our solution can be integrated with existing services like Amazon Elastic Compute Cloud [44].

7. Acknowledgments

The work presented in this paper has been partially funded by the Italian Ministry for Education, University and Research through the PRIN project TENACE.

References

[1] O. Etzion, P. Niblett, Event Processing in Action, Manning Publications Co., 2010.

[2] L. Aniello, G. A. Di Luna, G. Lodi, R. Baldoni, A Collaborative Event Processing System for Protection of Critical Infrastructures From Cyber Attacks, in: SAFECOMP '11.

[3] R. Baldoni, G. A. Di Luna, L. Querzoni, Collaborative Detection of Coordinated Portscan, in: ICDCN '13. To appear.

[4] Oracle big data appliance x3-2, `http://www.oracle.com/technetwork/server-storage/engineered-systems/bigdata-appliance/overview/index.html`, 2012. [last accessed March 22, 2013].

[5] Cloudera and dell partner to deliver complete apache hadoop solution, `http://www.cloudera.com/content/dam/cloudera/documents/Dell-Solution-Brief.pdf`, 2011. [last accessed March 22, 2013].

[6] Cloudera and microstrategy announce integration between business intelligence and apache hadoop, `http://finance.yahoo.com/news/Cloudera-MicroStrategy-iw-812071667.html?x=0`, 2011. [last accessed March 22, 2013].

[7] Cloudera enterprise with cisco unified computing system, http://files.cloudera.com/cisco/SolutionBrief_Cisco_Oct_2012.pdf, 2012. [last accessed March 22, 2013].

[8] R. Koster, A. P. Black, J. Huang, J. Walpole, C. Pu, Infopipes for composing distributed information flows, in: Proceedings of the 2001 international workshop on Multimedia middleware M3W.

[9] C. Pu, K. Schwan, Infosphere project: System support for information flow applications, ACM SIGMOD Record 30 (2001) 25–34.

[10] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management, The VLDB Journal 12 (2003) 120–139.

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, S. Zdonik, Scalable Distributed Stream Processing, in: CIDR 2003 - First Biennial Conference on Innovative Data Systems Research, Asilomar, CA.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. A. Shah, Telegraphcq: Continuous dataflow processing for an uncertain world, in: CIDR 2003 - First Biennial Conference on Innovative Data Systems Research.

[13] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, J. Widom, Stream: The stanford data stream management system, Technical Report, Stanford University, 2004.

[14] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the borealis stream processing engine, in: CIDR 2005 - Second Biennial Conference on Innovative Data Systems Research, pp. 277–289.

[15] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, C. Venkatramani, Spc: a distributed, scalable platform for data mining, in: Proceedings of the 4th international workshop on Data mining standards, services and platforms, DMSSP '06, ACM, New York, NY, USA, 2006, pp. 27–37.

[16] M. Shah, J. Hellerstein, S. Chandrasekaran, M. Franklin, Flux: an adaptive partitioning operator for continuous query systems, in: Proceedings of the 19th International Conference on Data Engineering, pp. 25 – 36.

[17] X. Gu, P. S. Yu, H. Wang, Adaptive load diffusion for multiway windowed stream joins, in: Proceedings of the 23rd International Conference on Data Engineering,, pp. 146–155.

[18] Y. Xing, J.-H. Hwang, U. Çetintemel, S. B. Zdonik, Providing resiliency to load variations in distributed stream processing, in: Proceedings of the 32nd international conference on Very large data bases, pp. 775–786.

[19] Y. Xing, S. Zdonik, J.-H. Hwang, Dynamic load distribution in the borealis stream processor, Proceedings of the 21st International Conference on Data Engineering, 0 (2005) 791–802.

[20] B. Liu, M. Jbantova, E. A. Rundensteiner, Optimizing state-intensive non-blocking queries using run-time adaptation, in: Proceedings of the 23rd International Conference on Data Engineering - Workshop, IEEE Computer Society, Washington, DC, USA, 2007, pp. 614–623.

[21] G. T. Lakshmanan, Y. G. Rabinovich, O. Etzion, A stratified approach for supporting high throughput event processing applications, in: Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems.

[22] B. Gedik, K.-L. Wu, P. S. Yu, L. Liu, Adaptive load shedding for windowed stream joins, in: Proceedings of the 14th ACM international conference on Information and knowledge management, CIKM '05, ACM, New York, NY, USA, 2005, pp. 171–178.

[23] U. Srivastava, J. Widom, Memory-limited execution of windowed stream joins, in: Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 324–335.

[24] A. Das, J. Gehrke, M. Riedewald, Approximate join processing over data streams, in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, SIGMOD '03, ACM, New York, NY, USA, 2003, pp. 40–51.

[25] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker, Load shedding in a data stream manager, in: Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '2003, VLDB Endowment, 2003, pp. 309–320.

[26] B. Liu, Y. Zhu, E. Rundensteiner, Run-time operator state spilling for memory intensive long-running queries, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 347–358.

[27] T. Urhan, M. J. Franklin, Xjoin: A reactively-scheduled pipelined join operator, IEEE Data Engineering Bulletin 23 (2000) 2000.

[28] M. F. Mokbel, M. Lu, W. G. Aref, Hash-merge join: A non-blocking join algorithm for producing fast and early join results., in: Z. M. Özsoyoglu, S. B. Zdonik (Eds.), Proceedings of the 20th International Conference on Data Engineering, IEEE Computer Society, 2004, pp. 251–262.

[29] S. D. Viglas, J. F. Naughton, J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, in: Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '03, VLDB Endowment, 2003, pp. 285–296.

[30] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, J. Saltz, Distributed processing of very large datasets with DataCutter, Parallel Computing 27 (2001) 1457–1478.

[31] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.

[32] T. White, Hadoop: The Definitive Guide, O'Reilly Media, original edition, 2009.

[33] H. Liu, D. Orban, Gridbatch: Cloud computing for large-scale data-intensive batch applications, in: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid, pp. 295–305.

[34] C. Miceli, M. Miceli, S. Jha, H. Kaiser, A. Merzky, Programming abstractions for data intensive computing on clouds and grids, in: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 478–483.

[35] P. Wang, D. Meng, J. Han, J. Zhan, B. Tu, X. Shi, L. Wan, Transformer: A new paradigm for building data-parallel programming models, IEEE Micro 30 (2010) 55–64.

[36] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys '07, ACM, New York, NY, USA, 2007, pp. 59–72.

[37] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '02, ACM, New York, NY, USA, 2002, pp. 1–16.

[38] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (1978) 558–565.

[39] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–10.

[40] T. W. Wlodarczyk, Y. Han, C. Rong, Performance analysis of hadoop for query processing, in: Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications - Workshops, pp. 507–513.

[41] B. Jia, T. W. Wlodarczyk, C. Rong, Performance considerations of data acquisition in hadoop system, Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (2010) 545–549.

[42] J. Boulon, et al., Chukwa, a large-scale monitoring system, in: Cloud Computing and its Applications, pp. 1–5.

[43] R. Baldoni, G. Chockler, Collaborative Financial Infrastructure Protection: Tools, Abstractions, and Middleware, Springer Publishing Company, Incorporated, 2012.

[44] Amazon elastic compute cloud, `http://aws.amazon.com/ec2/`, 2006.