Adaptive Online Scheduling in Storm

Leonardo Aniello
aniello@dis.uniroma1.it

Roberto Baldoni
baldoni@dis.uniroma1.it

Leonardo Querzoni
querzoni@dis.uniroma1.it

Research Center on Cyber Intelligence and Information Security and
Department of Computer, Control, and Management Engineering Antonio Ruberti
Sapienza University of Rome

ABSTRACT

Today we are witnessing a dramatic shift toward a data-driven economy, where the ability to efficiently and timely analyze huge amounts of data marks the difference between industrial success stories and catastrophic failures. In this scenario Storm, an open source distributed realtime computation system, represents a disruptive technology that is quickly gaining the favor of big players like Twitter and Groupon. A Storm application is modeled as a topology, i.e. a graph where nodes are operators and edges represent data flows among such operators. A key aspect in tuning Storm performance lies in the strategy used to deploy a topology, i.e. how Storm schedules the execution of each topology component on the available computing infrastructure. In this paper we propose two advanced generic schedulers for Storm that provide improved performance for a wide range of application topologies. The first scheduler works offline by analyzing the topology structure and adapting the deployment to it; the second scheduler enhance the previous approach by continuously monitoring system performance and rescheduling the deployment at run-time to improve overall performance. Experimental results show that these algorithms can produce schedules that achieve significantly better performances compared to those produced by Storm's default scheduler.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Distributed systems

Keywords

distributed event processing, CEP, scheduling, Storm

1. INTRODUCTION

In the last few years we are witnessing a huge growth in information production. IBM claims that "every day, we create 2.5 quintillion bytes of data - so much that 90% of the data in the world today has been created in the last two years alone" [15]. Domo, a business intelligence company, has recently reported some figures [4] that give a perspective on the sheer amount of data that is injected on the internet every minute, and its heterogeneity as well: 3125 photos are added on Flickr, 34722 likes are expressed on Facebook, more than 100000 tweets are done on Twitter, etc. This apparently unrelenting growth is a consequence of several factors including the pervasiveness of social networks, the smartphone market success, the shift toward an "Internet of things" and the consequent widespread deployment of sensor networks. This phenomenon, know with the popular name of *Big Data*, is expected to bring a strong growth in economy with a direct impact on available job positions; Gartner says that the business behind Big Data will globally create 4.4 million IT jobs by 2015 [1].

Big Data applications are typically characterized by the three *V*s: large *volumes* (up to petabytes) at a high *velocity* (intense data streams that must be analyzed in quasi real-time) with extreme *variety* (mix of structured and unstructured data). Classic data mining and analysis solutions quickly showed their limits when faced with such loads. Big Data applications, therefore, imposed a paradigm shift in the area of data management that brought us several novel approaches to the problem represented mostly by NoSQL databases, batch data analysis tools based on Map-Reduce, and complex event processing engines. This latter approach focussed on representing data as a real-time flow of events proved to be particularly advantageous for all those applications where data is continuously produced and must be analyzed on the fly. Complex event processing engines are used to apply complex detection and aggregation rules on intense data streams and output, as a result, new events. A crucial performance index in this case is represented by the average time needed for an event to be fully analyzed, as this represents a good figure of how much the application is quick to react to incoming events.

Storm [2] is a complex event processing engine that, thanks to its distributed architecture, is able to perform analytics on high throughput data streams. Thanks to these characteristics, Storm is rapidly conquering reputation among large companies like Twitter, Groupon or The Weather Channel. A Storm cluster can run *topologies* (Storm's jargon for an application) made up of several processing components. Components of a topology can be either *spouts*, that act as event producers, or *bolts* that implement the processing logic. Events emitted by a spout constitute a *stream* that can be transformed by passing through one or multiple bolts where its events are processed. Therefore, a topology repre-

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

sents a graph of stream transformations. When a topology is submitted to Storm it schedules its execution in the cluster, i.e., it assigns the execution of each spout and each bolt to one of the nodes forming the cluster. Similarly to batch data analysis tools like Hadoop, Storm performance are not generally limited by computing power or available memory as new nodes can be always added to a cluster.

In order to leverage the available resources Storm is equipped with a *default scheduler* that evenly distributes the execution of topology components on the available nodes using a round-robin strategy. This simple approach is effective in avoiding the appearance of computing bottlenecks due to resource overusage caused by skews in the load distribution. However, it does not take into account the cost of moving events through network links to let them traverse the correct sequence of bolts defined in the topology. This latter aspect heavily impacts the average event processing latency, i.e., how much time is needed for an event injected by a spout to traverse the topology and be thus fully processed, a fundamental metric used to evaluate the responsiveness of event processing applications to incoming stimuli.

In this paper we target the design and implementation of two general purpose Storm schedulers that, like the default one, could be leveraged by applications to improve their performance. Differently from the default Storm scheduler, the ones introduced in this paper aim at reducing the average event processing latency by adapting the schedule to specific application characteristics[1].

The rationale behind both schedulers is the following: (i) identify potential *hot edges* of the topology, i.e., edges traversed by a large number of events, and (ii) map an hot edge to a fast inter-process channel and not to a slow network link, for example by scheduling the execution of the bolts connected by the hot edge on a same cluster node. This rationale must take into account that processing resources have limited capabilities that must not be exceeded to avoid an undesired explosion of the processing time experienced at each topology component. Such pragmatic strategy has the advantage of being practically workable and to provide better performances.

The two general purpose schedulers introduced in this paper differ on the way they identify hot edges in the topology. The first scheduler, named *offline*, simply analyzes the topology graph and identifies possible sets of bolts to be scheduled on a same node by looking at how they are connected. This approach is simple and has no overhead on the application with respect to the default Storm scheduler (but for negligible increased processing times when the schedule is calculated), but it is oblivious with respect to the application workload: it could decide to schedule two bolts on a same node even if the number of events that will traverse the edge connecting them will be very small. The second scheduler, named *online*, takes this approach one step further by monitoring the effectiveness of the schedule at runtime and re-adapting it for a performance improvement when it sees fit. Monitoring is performed at runtime on the scheduled topology by measuring the amount of traffic among its components. Whenever there is the possibility for a new schedule to reduce the inter-node network traffic, the scheduled is calculated and transparently applied on the cluster preserving the application correctness (i.e. no events are discarded during this operation). The online scheduler thus provides adaptation to the workload at the cost of a more complex architecture. We have tested the performance of our general purpose schedulers by implementing them on Storm and by comparing the schedules they produce with those produced by the default Storm scheduler. The tests have been conducted both on a synthetic workload and on a real workload publicly released for the DEBS 2013 Grand Challenge. The results show how the proposed schedulers consistently delivers better performance with respect to the default one promoting them as a viable alternative to more expensive ad-hoc schedulers. In particular tests performed on the real workload show a 20% to 30% performance improvement on event processing latency for the online scheduler with respect to the default one proving the effectiveness of the proposed topology scheduling approach.

The rest of this paper is organized as follows: Section 2 introduces the reader to Storm and its default scheduler; Section 3 describes the *offline* and *online* schedulers; Section 4 reports the experiments done on the two schedulers. Finally, Section 5 discusses the related work and Section 6 concludes the paper.

2. STORM

Storm is an open source distributed realtime computation system [2]. It provides an abstraction for implementing event-based elaborations over a cluster of physical nodes. The elaborations consist in queries that are continuously evaluated on the events that are supplied as input. A computation in Storm is represented by a *topology*, that is a graph where nodes are operators that encapsulate processing logic and edges model data flows among operators. In the Storm's jargon, such a node is called a *component*. The unit of information that is exchanged among components is referred to as a *tuple*, that is a named list of values. There are two types of components: (i) *spouts*, that model event sources and usually wrap the actual generators of input events so as to provide a common mechanism to feed data into a topology, and (ii) *bolts*, that encapsulate the specific processing logic such as filtering, transforming and correlating tuples.

The communication patterns among components are represented by *streams*, unbounded sequences of tuples that are emitted by spouts or bolts and consumed by bolts. Each bolt can subscribe to many distinct streams in order to receive and consume their tuples. Both bolts and spouts can emit tuples on different streams as needed. Spouts cannot subscribe to any stream, since they are meant to produce tuples only. Users can implement the queries to be computed by leveraging the topology abstraction. They put into the spouts the logic to wrap external event sources, then compile the computation in a network of interconnected bolts taking care of properly handling the output of the computation. Such a computation is then submitted to Storm which is in charge of deploying and running it on a cluster of machines.

An important feature of Storm consists in its capability to scale out a topology to meet the requirements on the load to sustain and on fault tolerance. There can be several instances of a component, called *tasks*. The number of tasks for a certain component is fixed by the user when it configures the topology. If two components communicate through one or more streams, also their tasks do. The way such a communication takes place is driven by the *grouping* chosen

[1]The source code of the two schedulers can be found at `http://www.dis.uniroma1.it/~midlab/software/storm-adaptive-schedulers.zip`

by the user. Let A be a bolt that emits tuples on a stream that is consumed by another bolt B. When a task of A emits a new tuple, the destination task of B is determined on the basis of a specific grouping strategy. Storm provides several kinds of groupings

- *shuffle grouping*: the target task is chosen randomly, ensuring that each task of the destination bolt receives an equal number of tuples
- *fields grouping*: the target task is decided on the basis of the content of the tuple to emit; for example, if the target bolt is a stateful operator that analyzes events regarding customers, the grouping can be based on a customer-id field of the emitted tuple so that all the events about a specific customer are always sent to the same task, which is consequently enabled to properly handle the state of such customer
- *all grouping*: each tuple is sent to all the tasks of the target bolt; this grouping can be useful for implementing fault tolerance mechanisms
- *global grouping*: all the tuples are sent to a designated task of the target bolt
- *direct grouping*: the source task is in charge of deciding the target task; this grouping is different from fields grouping because in the latter such decision is transparently made by Storm on the basis of a specific set of fields of the tuple, while in the former such decision is completely up to the developer

2.1 Worker Nodes and Workers

From an architectural point of view, a Storm cluster consists of a set of physical machines called *worker nodes* whose structure is depicted in Figure 1. Once deployed, a topology consists of a set of threads running inside a set of Java processes that are distributed over the worker nodes.

A Java process running the threads of a topology is called a *worker* (not to be confused with a worker node: a worker is a Java process, a worker node is a machine of the Storm cluster). Each worker running on a worker node is launched and monitored by a *supervisor* executing on such worker node. Monitoring is needed to handle a worker failure.

Each worker node is configured with a limited number of *slots*, that is the maximum number of workers in execution on that worker node. A thread of a topology is called *executor*. All the executors executed by a worker belong to the same topology. All the executors of a topology are run by a specific number of workers, fixed by the developer of the topology itself. An executor carries out the logic of a set of tasks of the same component, tasks of distinct components live inside distinct executors. Also the number of executors for each component is decided when the topology is developed, with the constraint that the number of executors has to be lower than or equal to the number of tasks, otherwise there would be executors without tasks to execute. Figure 1 details what a worker node hosts.

Requiring two distinct levels, one for tasks and one for executors, is dictated by a requirement on dynamic rebalancing that consists in giving the possibility at runtime to scale out a topology on a larger number of processes (workers) and threads (executors). Changing at runtime the number

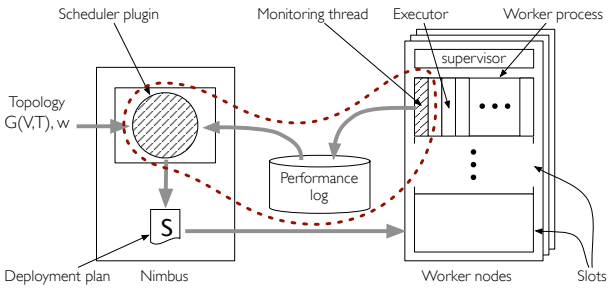

Figure 1: A Storm cluster with the Nimbus process controlling several Worker nodes. Each worker node hosts a *supervisor* and a number of *workers* (one per slot), each running a set of *executors*. The dashed red line shows components of the proposed solution: the offline scheduler only adds the plugin to the Nimbus process, while the online scheduler also adds the performance log and monitoring processes.

of tasks for a given component would complicate the reconfiguration of the communication patterns among tasks, in particular in the case of fields grouping where each task should repartition its input stream, and possibly its state, accordingly to the new configuration. Introducing the level of executors allows to keep the number of tasks fixed. The limitation of this design choice consists in the topology developer to overestimate the number of tasks in order to account for possible future rebalances. In this paper we don't focus on this kind of rebalances, and to keep things simple we always consider topologies where the number of tasks for any component is equal to the number of executors, that is each executor includes exactly one task.

2.2 Nimbus

Nimbus is a single Java process that is in charge of accepting a new topology, deploying it over worker nodes and monitoring its execution over time in order to properly handle any failure. Thus, Nimbus plays the role of master with respect to supervisors of workers by receiving from them the notifications of workers failures. Nimbus can run on any of the worker nodes, or on a distinct machine.

The coordination between nimbus and the supervisors is carried out through a ZooKeeper cluster [17]. The states of nimbus and supervisors are stored into ZooKeeper, thus, in case of failure, they can be restarted without any data loss.

The software component of nimbus in charge of deciding how to deploy a topology is called *scheduler*. On the basis of the topology configuration, the scheduler has to perform the deployment in two consecutive phases: (1) assign executors to workers, (2) assign workers to slots.

2.3 Default and Custom Scheduler

The Storm default scheduler is called *EvenScheduler*. It enforces a simple round-robin strategy with the aim of producing an even allocation. In the first phase it iterates through the topology executors, grouped by component, and allocates them to the configured number of workers in a round-robin fashion. In the second phase the workers are evenly assigned to worker nodes, according to the slot availability of each worker node. This scheduling policy produces

workers that are almost assigned an equal number of executors, and distributes such workers over the worker nodes at disposal so that each one node almost runs an equal number of workers.

Storm allows implementations of custom schedulers in order to accommodate for users' specific needs. In the general case, as shown in Figure 1, the custom scheduler takes as input the structure of the topology (provided by nimbus), represented as a weighted graph $G(V,T), w$, and set of user-defined additional parameters (α, β, ...). The custom scheduler computes a *deployment plan* which defines both the assignment of executors to workers and the allocation of workers to slots. Storm API provides the *IScheduler* interface to plug-in a custom scheduler, which has a single method *schedule* that requires two parameters. The first is an object containing the definitions of all the topologies currently running, including topology-specific parameters provided by who submitted the topology, which enables to provide the previously mentioned user-defined parameters. The second parameter is an object representing the physical cluster, with all the required information about worker nodes, slots and current allocations.
A Storm installation can have a single scheduler, which is executed periodically or when a new topology is submitted. Currently, Storm doesn't provide any mean to manage the movement of stateful components, it's up to the developer to implement application-specific mechanism to save any state to storage and properly reload them once a rescheduling is completed. Next section introduces the Storm custom scheduler we designed and implemented.

3. ADAPTIVE SCHEDULING

The key idea of the scheduling algorithms we propose is to take into account the communication patterns among executors trying to place in the same slot executors that communicate each other with high frequency. In topologies where the computation latency is dominated by tuples transfer time, limiting the number of tuples that have to be sent and received through the network can contribute to improve the performances. Indeed, while sending a tuple to an executor located in the same slot simply consists in passing a pointer, delivering a tuple to an executor running inside another slot or deployed in a different worker node involves much larger overheads.

We developed two distinct algorithms based on such idea. One looks at how components are interconnected within the topology to determine what are the executors that should be assigned to the same slot. The other relies on the monitoring at runtime of the traffic of exchanged tuples among executors. The former is less demanding in terms of required infrastructure and in general produces lower quality schedules, while the latter needs to monitor at runtime the cluster in order to provide more precise and effective solutions, so it entails more overhead at runtime for gathering performance data and carrying out re-schedulings.

In this work we consider a topology structured as a directed acyclic graph [8] where an upper bound can be set on the length of the path that any input tuple follows from the emitting spout to the bolt that concludes its processing. This means that we don't take into account topologies containing cycles, for example back propagation streams in online machine learning algorithms [10].

A Storm cluster includes a set $\mathcal{N} = \{n_i\}$ of worker nodes ($i = 1...N$), each one configured with S_i available slots ($i = 1...N$). In a Storm cluster, a set $\mathcal{T} = \{t_i\}$ of topologies are deployed ($i = 1...T$), each one configured to run on at most W_i workers ($i = 1...T$). A topology t_i consists of a set C_i of interconnected components ($i = 1...T$). Each component c_j ($j = 1...C_i$) is configured with a certain level of parallelism by specifying two parameters: (i) the number of executors, and (ii) the number of tasks. A component is replicated on many tasks that are executed by a certain number of executors. A topology t_i consists of E_i executors $e_{i,j}$ ($i = 1...T$, $j = 1...E_i$).

The actual number of workers required for a topology t_i is $\min(W_i, E_i)$. The total number of workers required to run all the topologies is $\sum_{i=1}^{T} \min(W_i, E_i)$. A schedule is possible if enough slots are available, that is $\sum_{i=1}^{N} S_i \geq \sum_{i=1}^{T} \min(W_i, E_i)$.

Both the algorithms can be tuned using a parameter α that controls the balancing of the number of executors assigned per slot. In particular, α affects the maximum number M of executors that can be placed in a single slot. The minimum value of M for a topology t_i is $\lceil E_i/W_i \rceil$, which means that each slot roughly contains the same number of executors. The maximum number of M corresponds to the assignment where all the slots contain one executor, except for one slot that contains all the other executors, so its value is $E_i - W_i + 1$. Allowed values for α are in $[0, 1]$ range and set the value of M within its minimum and maximum: $M(\alpha) = \lceil E_i/W_i \rceil + \alpha(E_i - W_i + 1 - \lceil E_i/W_i \rceil)$.

3.1 Topology-based Scheduling

The *offline scheduler* examines the structure of the topology in order to determine the most convenient slots where to place executors. Such a scheduling is executed before the topology is started, so neither the load nor the traffic are taken into account, and consequently no constraint about memory or CPU is considered. Not even the stream groupings configured for inter-component communications are inspected because the way they impact on inter-node and inter-slot traffic can be only observed at runtime. Not taking into account all these points obviously limits the effectiveness of the offline scheduler, but on the other hand this enables a very simple implementation that still provides good performance, as will be shown in Section 4. A partial order among the components of a topology can be derived on the basis of streams configuration. If a component c_i emits tuples on a stream that is consumed by another component c_j, then we have $c_i < c_j$. If $c_i < c_j$ and $c_j < c_k$ hold, then $c_i < c_k$ holds by transitivity. Such order is partial because there can be pairs of components c_i and c_j such that neither $c_i > c_j$ or $c_i < c_j$ hold. Since we deal with acyclic topologies, we can always determine a linearization ϕ of the components according to such partial order. If $c_i < c_j$ holds, then c_i appears in ϕ before c_j. If neither $c_i < c_j$ nor $c_i > c_j$ hold, then they can appear in ϕ in any order. The first element of ϕ is a spout of the topology. The heuristic employed by the offline scheduler entails iterating ϕ and, for each component c_i, placing its executors in the slots that already contain executors of the components that directly emit tuples towards c_i. Finally, the slots are assigned to worker nodes in a round-robin fashion.

A possible problem of this approach concerns the possibility that not all the required workers get used because, at each step of the algorithm, the slots that are empty get ig-

nored since they don't contain any executor. The solution employed by the offline scheduler consists in forcing to use empty slots at a certain point during the iteration of the components in ϕ. When starting to consider empty slots is controlled by a tuning parameter β, whose value lies in $[0, 1]$ range: during the assignment of executors for the i-th component, the scheduler is forced to use empty slots if $i > \lfloor \beta \cdot C_i \rfloor$. For example, if traffic is likely to be more intense among upstream components, then β should be set large enough such that empty slots get used when upstream components are already assigned.

3.2 Traffic-based Scheduling

The *online scheduler* produces assignments that reduce inter-node and inter-slot traffic on the basis of the communication patterns among executors observed at runtime. The goal of the online scheduler is to allocate executors to nodes so as to satisfy the constraints on (i) the number of workers each topology has to run on ($\min(W_i, E_i)$), (ii) the number of slots available on each worker node (S_i) and (iii) the computational power available on each node (see Section 3.2.1), and to minimize the inter-node traffic (see Section 3.2.1). Such scheduling has to be performed at runtime so as *to adapt the allocation to the evolution of the load in the cluster*. Figure 1 shows the integration of our online scheduler within the Storm architecture. Notice that the performance log depicted in the picture is just a stable buffer space where data produced by monitoring components running at each slot can be placed before it gets consumed by the custom scheduler on Nimbus. The custom scheduler can be periodically run to retrieve this data emptying the log and checking if a new more efficient schedule can be deployed.

3.2.1 Measurements

When scheduling the executors, taking into account the computational power of the nodes is needed to avoid any overload and in turn requires some measurements. We use the CPU utilization to measure both the load a node is subjected to (due to worker processes) and the load generated by an executor. Using the same metric allows us to make predictions on the load generated by a set of executors on a particular node. We also want to deal with clusters comprising heterogeneous nodes (different computational power), so we need to take into account the speed of the CPU of a node in order to make proper predictions. For example, if an executor is taking 10% CPU utilization on a 1GHz CPU, then migrating such executor on a node with 2GHz CPU would generate about 5% CPU utilization. For this reason, we measure the load in Hz. In the previous example, the executor generates a load of 100MHz (10% of 1GHz).

We use L_i to denote the load the node n_i is subjected to due to the executors. We use $L_{i,j}$ to denote the load generated by executor $e_{i,j}$. We use CPU_i to denote the speed of the CPU of node n_i (number of cores multiplied by single core speed).

CPU measurements have been implemented by leveraging standard Java API for retrieving at runtime the CPU time for a specific thread (*getThreadCpuTime(threadID)* method of *ThreadMXBean* class). With these measures we can monitor the status of the cluster and detect any imbalance due to node CPU overloads. We can state that if a node n_i exhibits a CPU utilization trend such that $L_i \geq B_i$ for more than X_i seconds, then we trigger a rescheduling. We refer to B_i as the capacity (measured in Hz) and to X_i as the time window (measured in seconds) of node n_i. One of the goals of a scheduling is the satisfaction of some constraints on nodes load.

We don't consider the load due to IO bound operations such as reads/writes to disk or network communications with external systems like DBMSs. Event-based systems usually work with data in memory in order to avoid any possible bottleneck so as to allow events to flow along the operators network as fast as possible. This doesn't mean that IO operations are forbidden, but they get better dealt with by employing techniques like executing them on a batch of events instead of on a single event and caching data in main memory to speed them up.

In order to minimize the inter-node traffic, the volumes of tuples exchanged among executors have to be measured. We use $R_{i,j,k}$ to denote the rate of the tuples sent by executor $e_{i,j}$ to executor $e_{i,k}$, expressed in tuples per second ($i = 1...T; j, k = 1...E_i; j \neq k$). Summing up the traffic of events exchanged among executors deployed on distinct nodes, we can measure the total inter-node traffic. Once every P seconds, we can compute a new scheduling, compare the inter-node traffic such scheduling would generate with the current one and, in case a reduction of more than R% in found, trigger a rescheduling.

3.2.2 Formulation

Given the set of nodes $\mathcal{N} = \{n_i\}$ ($i = 1...N$), the set of workers $\mathcal{W} = \{w_{i,j}\}$ ($i = 1...T$, $j = 1... \min(E_i, W_i)$) and the set of executors $\mathcal{E} = \{e_{i,j}\}$ ($i = 1...N$,$j = 1...E_i$), the goal of load balancing is to assign each executor to a slot of a node. The scheduling is aimed at computing (i) an allocation $A_1 : \mathcal{E} \rightarrow \mathcal{W}$, which maps executors to workers, and (ii) an allocation $A_2 : \mathcal{W} \rightarrow \mathcal{N}$, which maps workers to nodes.

The allocation has to satisfy the constraints on nodes capacity

$$\forall k = 1...N \sum_{\substack{A_2(A_1(e_{i,j}))=n_k \\ i=1...T; j=1...E_i}} L_{i,j} \leq B_k \quad (1)$$

as well as the constraints on the maximum number of workers each topology can run on

$$\forall i = 1...T$$
$$|\{w \in \mathcal{W} : A_1(e_{i,j}) = w, j = 1...E_i\}| = \min(E_i, W_i) \quad (2)$$

The objective of the allocation is to minimize the inter-node traffic

$$\min \sum_{\substack{j,k:A_2(A_1(e_{i,j}))\neq A_2(A_1(e_{i,k})) \\ i=1...T; j,k=1...E_i}} R_{i,j,k} \quad (3)$$

3.2.3 Algorithm

The problem formulated in Section 3.2.2 is known to be NP-complete [9, 18]. The requirement of carrying the rebalance out at runtime implies the usage of a quick mechanism to find a new allocation, which in turn means that some heuristic has to be employed. The following algorithm is based on a simple greedy heuristic that place executors to node so as to minimize inter-node traffic and avoid load imbalances among all the nodes. It consists of two consecutive phases.

In the first phase, the executors of each topology are partitioned among the number of workers the topology has been

Data:
$\mathcal{T} = \{t_i\}$ $(i = 1...T)$: set of topologies
$\mathcal{E} = \{e_{i,j}\}$ $(i = 1...T; j = 1...E_i)$: set of executors
$\mathcal{N} = \{n_i\}$ $(i = 1...N)$: set of nodes
$\mathcal{W} = \{w_{i,j}\}$ $(i = 1...T; j = 1... \min(E_i, W_i))$: set of workers
$L_{i,j}$ $(i = 1...T; j = 1...E_i)$: load generated by executor $e_{i,j}$
$R_{i,j,k}$ $(i = 1...T; j, k = 1...E_i)$: tuple rate between executors $e_{i,j}$ and $e_{i,k}$

begin
  `// First Phase`
  foreach *topology* $t_i \in \mathcal{T}$ **do**
    `// Inter-Executor Traffic for topology` t_i
    $IET_i \leftarrow \{\langle e_{i,j}; e_{i,k}; R_{i,j,k}\rangle\}$ sorted descending by $R_{i,j,k}$
    foreach $\langle e_{i,j}; e_{i,k}; R_{i,j,k}\rangle \in IET_i$ **do**
      `// get least loaded worker`
      $w* \leftarrow \text{argmin}_{w_{i,x} \in \mathcal{W}} \sum_{A_1(e_{i,y})=w_{i,x}} L_{i,y}$
      if *!*`assigned(`$e_{i,j}$`)` **and** *!*`assigned(`$e_{i,k}$`)` **then**
        `// assign both executors to` w^*
        $A_1(e_{i,j}) \leftarrow w^*$
        $A_1(e_{i,k}) \leftarrow w^*$
      else
        `// check the best assignment of` $e_{i,j}$ `and` $e_{i,k}$ `to the workers that already`
        `// include either executor and to` w^* `(at most 9 distinct assignments to consider)`
        $\Pi \leftarrow \{w \in \mathcal{W} : A_1(e_{i,j}) = w \vee A_1(e_{i,k}) = w\} \cup \{w^*\}$
        $best\_w\_j \leftarrow null$
        $best\_w\_k \leftarrow null$
        $best\_ist \leftarrow MAX\_INT$
        foreach $\langle w\_j, w\_k\rangle \in \Pi^2$ **do**
          $A_1(e_{i,j}) \leftarrow w\_j$
          $A_1(e_{i,k}) \leftarrow w\_k$
          $ist \leftarrow \sum_{x,y:A_1(e_{i,x}) \neq A_1(e_{i,y})} R_{i,x,y}$
          if $ist < best\_ist$ **then**
            $best\_ist \leftarrow ist$
            $best\_w\_j \leftarrow w\_j$
            $best\_w\_k \leftarrow w\_k$
          end
        end
        $A_1(e_{i,j}) \leftarrow best\_w\_j$
        $A_1(e_{i,k}) \leftarrow best\_w\_k$
      end
    end
  end

  `// Second Phase`
  $IST \leftarrow \{\langle w_{i,x}; w_{i,y}; \gamma_{i,x,j}\rangle : w_{i,x}, w_{i,y} \in \mathcal{W}, \gamma_{i,x,j} = \sum_{A_1(e_{i,j})=w_{i,x} \wedge A_1(e_{i,k})=w_{i,y}} R_{i,j,k}\}$ sorted descending by $\gamma_{i,x,j}$
  foreach $\langle w_{i,x}; w_{i,y}; \gamma_{i,x,j}\rangle$ **do**
    $n* \leftarrow \text{argmin}_{n \in \mathcal{N}} \sum_{A_2(A_1(e_{i,y}))=n} L_{i,y}$
    if *!*`assigned(`$w_{i,x}$`)` **and** *!*`assigned(`$w_{i,y}$`)` **then**
      $A_2(w_{i,x}) \leftarrow n^*$
      $A_2(w_{i,y}) \leftarrow n^*$
    else
      `// check the best assignment of` $w_{i,x}$ `and` $w_{i,y}$ `to the nodes that already`
      `// include either worker and to` n^* `(at most 9 distinct assignments to consider)`
      $\Xi \leftarrow \{n \in \mathcal{N} : A_2(w_{i,x}) = n \vee A_2(w_{i,y}) = n\} \cup \{n^*\}$
      $best\_n\_x \leftarrow null$
      $best\_n\_y \leftarrow null$
      $best\_int \leftarrow MAX\_INT$
      foreach $\langle n\_x, n\_y\rangle \in \Xi^2$ **do**
        $A_2(w_{i,x}) \leftarrow n\_x$
        $A_2(w_{i,y}) \leftarrow n\_y$
        $int \leftarrow \sum_{j,k:A_2(A_1(e_{i,j})) \neq A_2(A_1(e_{i,k}))} R_{i,j,k}$
        if $int < best\_int$ **then**
          $best\_int \leftarrow int$
          $best\_n\_x \leftarrow n\_x$
          $best\_n\_y \leftarrow n\_y$
        end
      end
      $A_2(w_{i,x}) \leftarrow best\_n\_x$
      $A_2(w_{i,y}) \leftarrow best\_n\_y$
    end
  end
end

Algorithm 1: Online Scheduler

configured to run on. The placement is aimed to both minimize the traffic among executors of distinct workers and balance the total CPU demand of each worker.

In the second phase, the workers produced in the first phase have to be allocated to available slots in the cluster. Such allocation still has to take into account both inter-node traffic, in order to minimize it, and node load, so as to satisfy load capacity constraints.

Algorithm 1 presents the pseudo-code for the online scheduler. This is an high level algorithm that doesn't include the implementation of many corner cases but shows instead the core of the heuristic.

In the first phase, for each topology, the pairs of communicating executors are iterated in descending order by rate of exchanged tuples. For each of these pairs, if both the executors have not been assigned yet, then they get assigned to the worker that is the least loaded at that moment. Otherwise, the set Π is built by putting the least loaded worker together with the workers where either executor of the pair is assigned. Π can contain three elements at most: the least loaded and the two where the executors in the pair are currently assigned. All the possible assignments of these executors to these workers are checked to find the best one, that is the assignment that produces the lowest inter-worker traffic. At most, there can be 9 distinct possible assignments to check.

Similarly, in the second phase the pairs of communicating workers are iterated in descending order by rate of exchanged tuples. For each pair, if both have not been allocated to any node yet, then the least loaded node is chosen to host them. If any or both have already been assigned to some other nodes, the set Ξ is built using these nodes and the least loaded one. All the possible allocations of the two workers to the nodes in Ξ are examined to find the one that generates the minimum inter-node traffic. Again, there are at most 9 distinct allocations to consider.

4. EVALUATION

Our experimental evaluation aims at giving evidence that the scheduling algorithms we propose are successful at improving performances on a wide range of topologies. We first test their performance on a general topology that captures the characteristics of a broad class of topologies and show how the algorithms' tuning parameters impact on the efficiency of the computation, comparing the results with those obtained by using the default scheduler. Then, in order to evaluate our solution in a more realistic setting, we apply our scheduling algorithms to the DEBS 2013 Grand Challenge dataset by implementing a subset of its queries. Performance were evaluated on two fundamental metrics: the average latency experienced by events to traverse the entire topology and the average inter-node traffic incurred by the topology at runtime.

All the evaluations were performed on a Storm cluster with 8 worker nodes, each with 5 slots, and one further node hosting the Nimbus and Zookeeper services. Each node runs Ubuntu 12.04 and is equipped with 2x2.8 GHz CPUs, 3 GB of RAM and 15 GB of disk storage. The networking infrastructure is based on a 10 Gbit LAN. These nodes are kept synchronized with a precision of microseconds, which is sufficiently accurate for measuring latencies in the order of milliseconds. Such synchronization has been obtained by leveraging the standard NTP protocol to sync all the nodes with a specific node in the cluster.

4.1 Reference Topology

In this section we analyze how the tuning parameters actually affect the behavior of scheduling algorithms and consequently the performances of a topology. In order to avoid focusing on a specific topology, we developed a reference topology aimed at capturing the salient characteristics of many common topologies.

Workload characteristics.

According to the kind of topologies we deal with in this work (see Section 3), we consider acyclic topologies where an upper bound can be set on the number of hops a tuple has go through since it is emitted by a spout up to the point where its elaboration ends on some bolt. This property allows us to assign each component c_i in the topology a number $stage(c_i)$ that represents the length of the longest path a tuple must travel from any spout to c_i. By grouping components in a same *stage*, we obtain a horizontal stratification of the topology in stages, such that components within the same stage don't communicate each other, and components at stage i receive tuples from upstream components at stages lower than i and send tuples to downstream components at stages greater than i. This kind of stratification has been investigated in [19]. Recent works on streaming MapReduce [21, 20, 11] also focus on the possibility to model any computation as a sequence of alternated map and reduce stages, supporting the idea that a large class of computations can be structured as a sequence of consecutive stages where events always flow from previous to subsequent stages.

These considerations led us to propose the working hypothesis that a chain topology can be employed as a meaningful sample of a wide class of possible topologies. Chain topologies are characterized by two parameters: (i) the number of stages, that is the horizontal dimension and (ii) the replication factor for each stage, that is the vertical dimension corresponding to the number of executors for each topology component. We developed a *reference topology* according to such working hypothesis. Taking inspiration from the MapReduce model [14], in the chain we alternate bolts that receive tuples using shuffle grouping (similar to mappers) to bolts that are fed through fields grouping (similar to reducers). In this way we can also take into account how the grouping strategy impacts on the generated traffic patterns.

Figure 2 shows the general structure of the reference topology. It contains a single spout followed by an alternation of *simple* bolts, that receive tuples by shuffle grouping, and *stateful* bolts, that instead take tuples by fields grouping. Stateful bolts have been named so because their input stream is partitioned among the executors by the value embedded in the tuples, and this would enable each executor to keep some sort of state. In the last stage there is an *ack* bolt in charge of completing the execution of tuples.

Each spout executor emits tuples containing an incremental numeric value at a fixed rate. Using incremental numeric values allows to evenly spread tuples among target executors for bolts that receive input through fields grouping. Each spout executor chooses its fixed rate using two parameters: the average input rate R in tuples per second and its variance V, expressed as the largest difference in percentage of the actual tuple rate from R.

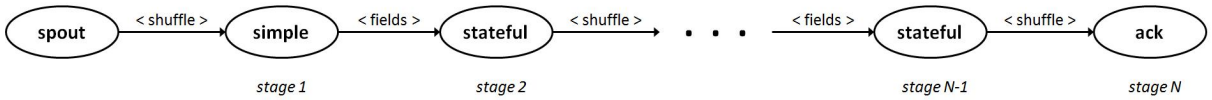

Figure 2: Reference topology.

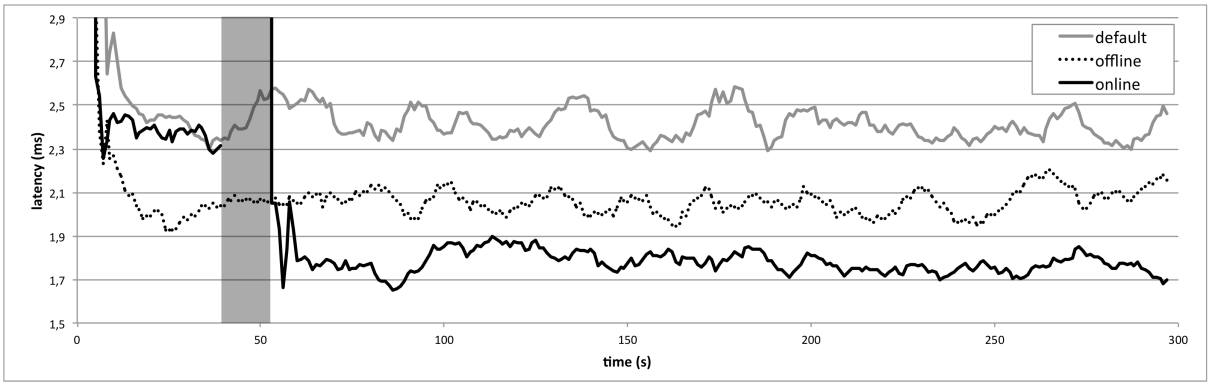

Figure 3: Tuple processing latency over time, for default, offline and online schedulers.

The i-th spout executor sets its tuple rate as $R_i = R(1 - V(1 - 2\frac{i}{C_0-1}))$ where C_0 is the number of executors for the first component, that is the spout itself, and $i = 0, ..., C_0 - 1$. Therefore, each spout executor emits tuples at a distinct fixed rate and the average of these rates is exactly R. In this way, the total input rate for the topology can be controlled (its value is $C_0 \cdot R$) and a certain degree of irregularity can be introduced on traffic intensity (tuned by V parameter) in order to simulate realistic scenarios where event sources are likely to produce new data at distinct rates.

In order to include other factors for breaking the regularity of generated traffic patterns, bolts in the reference topology have been implemented so as to forward the received value with probability 1/2 and to emit a different constant value (fixed for each executor) the rest of the times. The traffic between executors whose communication is setup using fields grouping is affected by this mechanism since it makes the tuple rates much higher for some executor pairs. This choice models realistic situations where certain pairs of executors in consecutive stages communicate more intensively than others.

Evaluation.

The first experiments were focussed at evaluating the runtime behavior of the proposed schedulers with respect to the default one. Figure 3 reports how event latency evolves over time for an experiment. Each points reported in the figure represents the average of latencies for a 10 events window. The reference topology settings used in this test include 7 stages, and variable replication factors: 4 for the spout, 3 for the bolts receiving tuples through shuffle grouping and 2 for the bolts receiving tuples through fields grouping.

At the beginning all schedules experiences a short transient state where the system seems overloaded and this heavily impacts measured latencies. This transient period lasts approximately 15-20 seconds and is characterized by large latencies. In the subsequent 20 seconds time frame (up to second 40) it is possible to observe some characteristic behavior. The performance for all three schedules are reasonably stable, with both the default and online schedulers sharing similar figures, and the offline scheduler showing better results. This result proves how the topology-based optimizations performed by the offline scheduler quickly pay-off with respect to the default scheduler approach. The online scheduler performance in this timeframe are instead coherent with the fact that this scheduler initializes the application using a schedule obtained applying exactly the same approach used by the default scheduler (hence the similar performance). However, during this period the active scheduler collects performance measures that are used later (at second 40) to trigger a re-schedule. The shaded interval in the figure shows a "silence" period used by the active scheduler to instantiate a new schedule. The new schedule starts working at second 50 and quickly converges to performance that are consistently better with respect to both the default and the offline scheduler. This proves that the online scheduler is able to correctly identify cases where a different schedule can improve performance, and that this new schedule, built on the basis of performance indices collected at runtime, can, in fact, provide performance that surpass a workload-oblivious schedule (like the one provided by the offline scheduler).

We then evaluated how the proposed schedulers behave as the number of stages increases for different replication factors. As the number of stages increases, the latency obviously becomes larger as each tuple has to go through more processing stages. With a small replication factor traffic patterns among executors are quite simple. In general, with a replication factor F, there are F^2 distinct streams among the executors of communicating bolts because each executor of a stage possibly communicate to all the executors at the next stage. The offline scheduler does its best to place each of the F executors of a bolt c_i where at least one of F executors of the component c_{i-1} has been already placed, which means that, in general, the latency of up to F streams out

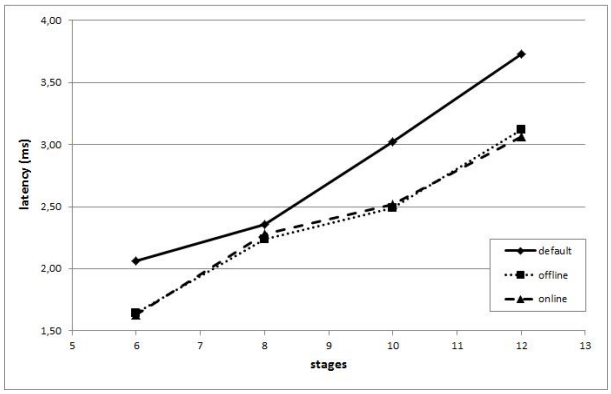

Figure 4: Average latency as the number of stages varies, with a replication factor of 2 for each stage.

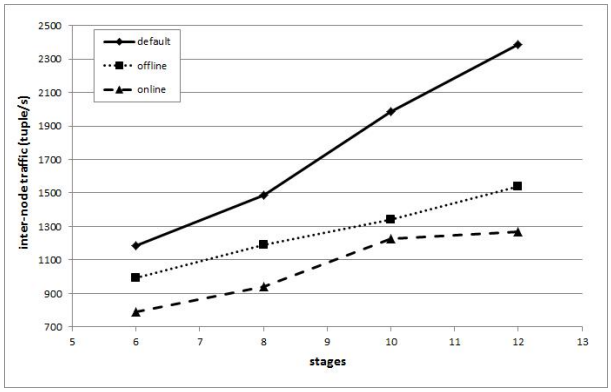

Figure 5: Average inter-node traffic as the number of stages varies, with a replication factor of 2 for each stage.

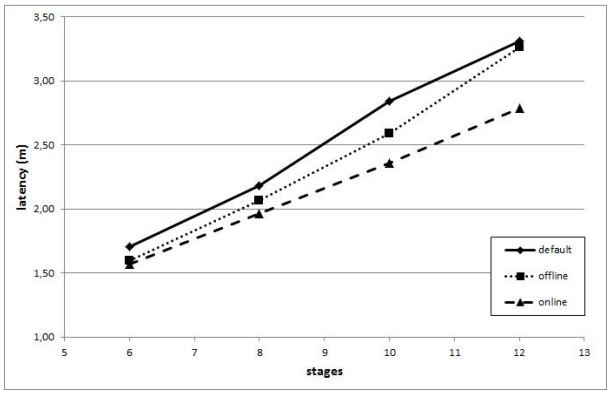

Figure 6: Average latency as the number of stages varies, with a replication factor of 4 for each stage, for default, offline and online schedulers.

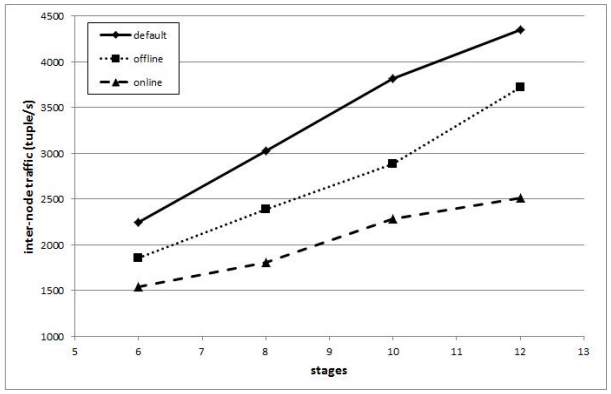

Figure 7: Average inter-node traffic as the number of stages varies, with a replication factor of 4 for each stage, for default, offline and online schedulers.

of F^2 is improved. Therefore, about $1/F$ of the tuples flowing among consecutive components get sent within the same node with a consequent latency improvement. As the replication factor increases, the portion of tuples that can be sent locally gets lower and the effectiveness of the offline scheduler becomes less evident. The precise trend also depends on whether the streams that are optimized are intense or not; however, the offline scheduler is is oblivious with respect to this aspect as it calculate the schedule before the topology is executed. On the other hand, the online scheduler adapts to the actual evolution of the traffic and is able to identify the heaviest streams and consequently place executors so as to make such streams local.

These evaluations have been carried out setting the parameters $\alpha = 0$ and $\beta = 0.5$, considering an average data rate $R = 100$ tuple/s with variance $V = 20\%$.

Figures 4 and 5 report average latency and inter-node traffic for a replication factor 2, i.e. each component is configured to run on 2 executors. Latencies for offline and online schedulers are close and always smaller with respect to the default scheduler. The low complexity of communication patterns allows for only a little number of improvement actions, which are leveraged by both the schedulers with the consequent effect that performances prove to be very similar. The results about the inter-node traffic reflect this trend and also highlight that the online scheduler produces schedules with smaller inter-node traffic. Note that smaller inter-node traffic cannot always be directly related to a lower latency because it also depends on whether and to what extent the most intense paths in the topology are affected.

Figures 6 and 7 the same results for a replication factor set to 4. While the online scheduler keeps providing sensibly lower latencies with respect to the default one, the effectiveness of offline scheduler begins to lessen due to the fact that it can improve only 4 out of 16 streams for each stage. Such a divergence between the performances of offline and online schedulers is also highlighted by the results on the inter-node traffic; indeed the online scheduler provides assignments that generate lower inter-node traffic.

We finally evaluated the impact of the α parameter on the schedules produced by our two algorithms. Figures 8 and 9 report results for a setting based on a 5 stages topology with replication factor 5, $R = 1000$ tuple/s and $V = 20\%$.

In such setting a topology consists of 30 executors, and we varied α from 0 to 0.2, which corresponds to varying the maximum number of executors per slots from 4 to 8. The results show that the offline scheduler slightly keeps improving its performances as α grows, for what concerns both the latency and the inter-node traffic. The online scheduler pro-

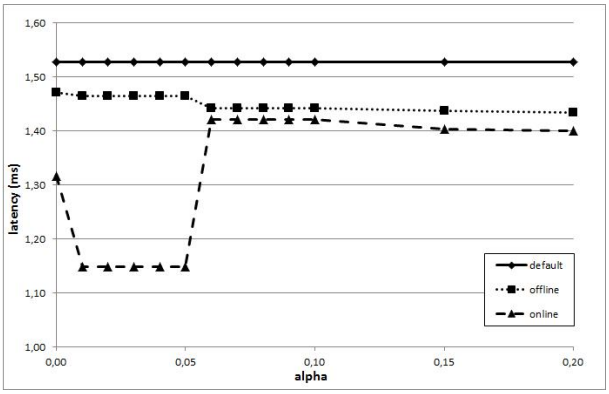

Figure 8: Average latency as α varies for default, offline and online schedulers.

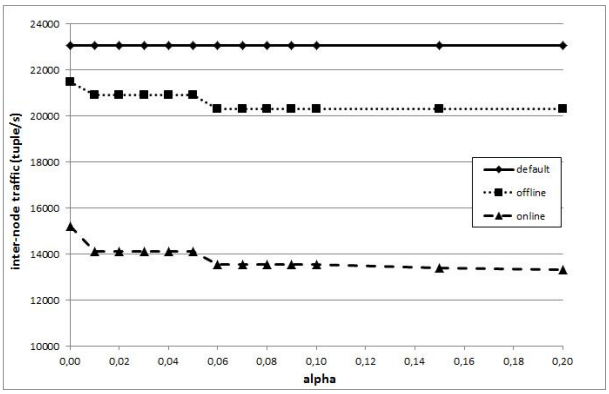

Figure 9: Average inter-node traffic as α varies for default, offline and online schedulers.

vides its best performances when α is 0.05, that is when the upper bound on the number of executors is 5. Larger values for α provide larger latencies despite the inter-node traffic keeps decreasing. This happens because there is a dedicated thread for each worker in charge of dequeuing tuples and sending them to the others workers, and placing too many executors in a single slot makes this thread a bottleneck for the whole worker.

4.2 Grand Challenge Topology

We carried out some evaluations on the scenario described in the Grand Challenge of DEBS 2013, considering in particular a reduced version of the first query. In such scenario, sensors embedded in soccer players' shoes emit position and speed data at 200Hz frequency. The goal of the first query is to perform a running analysis by continuously updating statistics about each player. The instantaneous speed is computed for each player every time a new event is produced by the sensors, a speed category is determined on the basis of computed value, then the global player statistics are updated accordingly. Such statistics include average speed, walked distance, average time for each speed category.

The topology includes three components: (i) a spout for the sensors (*sensor* component in the figure, replication factor 8, with a total of 32 sensors to be simulated), (ii) a bolt that computes the instantaneous speed and receives tuples by shuffle grouping (*speed* component in the figure, replication factor 4), (iii) a bolt that maintains players' statistics and updates them as new tuples are received by fields grouping from the speed bolt (*analysis* component in the figure, replication factor 2).

Figure 10 shows how latency (top) and inter-node traffic (bottom) evolve over time. It can be noticed that most of the time the offline scheduler allows for lower latencies and lighter traffic than the default one, while the online scheduler in turn provides better performances than the offline one as soon as an initial transient period needed to collect performance indices is elapsed. The performance improvement provided by the online scheduler with respect to the default one can be quantified in this setting as oscillating between 20% and 30%. These results confirms that the optimizations performed by the online scheduler are effective also in real workloads where they provide noticeable performance improvements.

5. RELATED WORK

There exist alternative distributed event and stream processing engines besides Storm. Similarly to Storm, these engines allow to model the computation as continuous queries that run uninterruptedly over input event streams. The queries in turn are represented as graphs of interconnected operators that encapsulate the logic of the queries.

System S [6] is a stream processing framework developed by IBM. A query in System S is modeled as an Event Processing Network (EPN) consisting of a set of Event Processing Agents (EPAs) that communicate each other to carry out the required computation. The similarity to Storm is very strong, indeed EPNs can be seen as the equivalent of Storm topologies and EPAs as the analogous of bolts. S4 [23] is a different stream processing engine, developed by Yahoo, where queries are designed as graphs of Processing Elements (PEs) which exchange events according to queries' specification. Again, the affinity with Storm is evident as the PEs definitely correspond to the bolts.

Another primary paradigm of elaboration in the scope of Big Data is the batch oriented MapReduce [14] devised by Google, together with its main open source implementation Hadoop [27] developed by Apache. The employment of a batch approach hardly adapts to the responsiveness requirements of today's applications that have to deal with continuous streams of input events, but there are scenarios [7] where it still results convenient adopting such an approach where the limitations of the batch paradigm are largely offset by the strong characteristics of scalability and fault tolerance of a MapReduce based framework.

An attempt to address the restrictions of a batch approach is MapReduce online [13], that is introduced as an evolution of the original Hadoop towards a design that fits better to the requirements of stream based applications.

Other works [21, 20, 11] try to bridge the gap between continuous queries and MapReduce paradigm by proposing a stream based version of the MapReduce approach [3] where events uninterruptedly flow among the map and reduce stages of a certain computation without incurring in the delays typical of batch oriented solutions.

The problem of efficiently schedule operators in CEP engines has been tackled in several works. Cammert et al. [12] investigated how to partition a graph of operators into subgraphs and how to assign each subgraph to a proper number of threads in order to overcome common complications

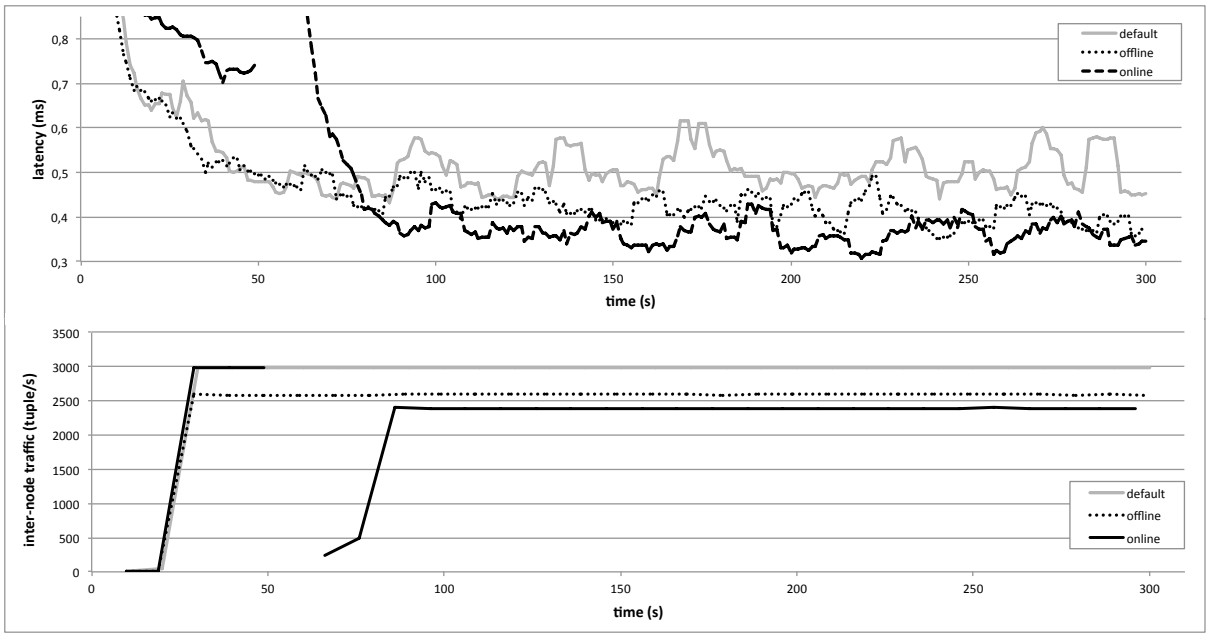

Figure 10: Latency (top) and traffic (bottom) over time for default, offline and online schedulers for the first query of DEBS 2013 Grand Challenge.

concerning threads overhead and operators stall. Moakar et al. [22] explored the question of scheduling for continuous queries that exhibit different classes of characteristics and requirements, and proposed a strategy to take into account such heterogeneity while optimizing response latency. Sharaf et al. [25] focus on the importance of scheduling in environments where the streams to consume are quite heterogeneous and present high skews; they worked on a rate-based scheduling strategy which accounts for the specific features of the streams to produce effective operators schedules.

Hormati et al. [16] and Suleman et al [26] propose works conceived for multi core systems rather than clusters of machines. Differently from our solutions, the former aims at maximizing the throughput by combining a preliminary static compilation with adaptive dynamic changes of the configuration that get triggered by variations in resource availability. The latter focuses on chain topologies with the goal of minimizing execution time and number of used cores by tuning the parallelism of bottleneck stages in the pipeline. On the other hand, Pietzuch et al. [24] are concerned with operator placement within pools of wide-area overlay nodes. They proposed a stream-based overlay network in charge of reducing the latency and leveraging possible reuse of operators. The solution proposed in this paper, differently from [24], does not consider the efficient use of the network as a first class goal, nor does consider operator reuse possible. SODA [28] is an optimized scheduler specific for System S [6] which takes into account several distinct metrics in order to produce allocations that optimize an application-specific measure ("importance") and maximize nodes and links usage. One of the assumptions that drives their scheduling strategy is that the offered load would far exceed system capacity much of the time, an assumption that cannot be made for Storm applications. Xing et al. [30] presented a methodology to produce balanced operator mapping plans for Borealis [5]. They only consider node load and actually ignore the impact of network traffic. In a later work, Xing et at. [29] described an operator placement plan that is resilient to changes in load, but makes the relevant assumption that operators cannot be moved at runtime.

6. CONCLUSIONS

Storm is an emerging technology in the field of Big Data and its employment in real scenarios keeps increasing as well as the open source community that supports and develops it. The wide range of use cases Storm is expected to support and their relevant complexity make the evolution of Storm quite challenging and push for general purpose new features able to fit to most use cases. The work presented in this paper goes along this line. We designed and implemented two generic schedulers for Storm that adapt their behavior according to the topology and the runt-time communication pattern of the application. For Storm users that do not want to create from scratch ad-hoc schedulers for their applications, such adaptive schedulers represent good alternatives to the usage of the default scheduler provided by Storm. Experiments show the effectiveness of the approach as the latency of processing an event is below 20-30% with respect to the default Storm scheduler in both tested topologies.

Acknowledgments

We would like to thank the anonymous reviewers and our paper shepherd, Peter Fischer, for all the useful comments and suggestions that greatly helped us in improving this paper. This work has been partially supported by the TENACE project (MIUR-PRIN 20103P34XC).

7. REFERENCES

[1] Gartner says big data creates big jobs: 4.4 million it jobs globally to support big data by 2015. `http://www.gartner.com/newsroom/id/2207915`.

[2] Storm. http://storm-project.net/.

[3] Streammine3g. https://streammine3g.inf.tu-dresden.de/trac.

[4] Data never sleeps infographic. http://www.domo.com/learn/7/236#videos-and-infographics, 2012.

[5] 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, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, 2005.

[6] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: a distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, 2006.

[7] L. Aniello, L. Querzoni, and R. Baldoni. Input data organization for batch processing in time window based computations. In *Proceedings of the 28th Symposium On Applied Computing*, 2013.

[8] R. Baldoni, G. A. Di Luna, D. Firmani, and G. Lodi. A model for continuous query latencies in data streams. In *Proceedings of the 1st International Workshop on Algorithms and Models for Distributed Event Processing*, 2011.

[9] S. Bansal, P. Kumar, and K. Singh. An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[10] Y. Baram, R. El-Yaniv, and K. Luz. Online choice of active learning algorithms. *The Journal of Machine Learning Research*, 2004.

[11] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, 2011.

[12] M. Cammert, C. Heinz, J. Kramer, B. Seeger, S. Vaupel, and U. Wolske. Flexible multi-threaded scheduling for continuous queries over data streams. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, 2007.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[15] C. Eaton, D. Deroos, T. Deutsch, G. Lapis, and P. Zikopoulos. *Understanding Big Data*. Mc Graw Hill, 2012.

[16] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[17] F. P. Junqueira and B. C. Reed. The life and times of a zookeeper. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures*, 2009.

[18] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1996.

[19] G. T. Lakshmanan, Y. G. Rabinovich, and 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*, 2009.

[20] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, 2011.

[21] A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, C. Fetzer, and A. Brito. Low-overhead fault tolerance for high-throughput data processing systems. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, 2011.

[22] L. A. Moakar, A. Labrinidis, and P. K. Chrysanthis. Adaptive class-based scheduling of continuous queries. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012.

[23] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, 2010.

[24] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[25] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Preemptive rate-based operator scheduling in a data stream management system. In *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, 2005.

[26] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.

[27] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[28] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008.

[29] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, 2006.

[30] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, 2005.