Proactive Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems

Nicoló Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, Bruno Sericola

Proactive Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems

Nicoló Rivetti*§, Emmanuelle Anceaume†, Yann Busnel‡, Leonardo Querzoni§, Bruno Sericola¶
* LINA / Université de Nantes, France - nicolo.rivetti@univ-nantes.fr
† IRISA / CNRS, Rennes, France - emmanuelle.anceaume@irisa.fr
‡ Crest (Ensai) / Inria, Rennes, France - yann.busnel@ensai.fr
§ DIAG / Sapienza University of Rome, Italy - {lastname}@dis.uniroma1.it
¶Inria, Rennes, France - bruno.sericola@inria.fr

Abstract

Shuffle grouping is a technique used by stream processing frameworks to share input load among parallel instances of stateless operators. With shuffle grouping each tuple of a stream can be assigned to any available operator instance, independently from any previous assignment. A common approach to implement shuffle grouping is to adopt a round robin policy, a simple solution that fares well as long as the tuple execution time is constant. However, such assumption rarely holds in real cases where execution time strongly depends on tuple content. As a consequence, parallel stateless operators within stream processing applications may experience unpredictable unbalance that, in the end, causes undesirable increase in tuple completion times. In this paper we propose Proactive Online Shuffle Grouping (POSG), a novel approach to shuffle grouping aimed at reducing the overall tuple completion time. POSG estimates the execution time of each tuple, enabling a proactive and online scheduling of input load to the target operator instances. Sketches are used to efficiently store the otherwise large amount of information required to schedule incoming load. We provide a probabilistic analysis and illustrate, through both simulations and a running prototype, its impact on stream processing applications.

I. Introduction

Stream processing systems are today gaining momentum as a tool to perform analytics on continuous data streams. Their ability to produce analysis results with sub-second latencies, coupled with their scalability, makes them the preferred choice for many big data companies. A stream processing application is commonly modeled as a direct acyclic graph where data operators, represented by vertices, are interconnected by streams of tuples containing data to be analyzed, the directed edges. Scalability is usually attained at the deployment phase where each data operator can be parallelized using multiple instances, each of which will handle a subset of the tuples conveyed by the operator's ingoing stream. The strategy used to route tuples in a stream toward available instances of the receiving operator is embodied in a so-called *grouping* function.

Operator parallelization is straightforward for *stateless* operators, *i.e.*, data operators whose output is only function of the current tuple in input. In this case, in fact, the grouping function is free to assign the next tuple in the input stream, to any available instance of the receiving operator (contrarily to statefull operators, where this choice is typically constrained). Such grouping functions are often called *shuffle grouping*.

Shuffle grouping implementations are designed to balance as much as possible the load on the receiving operator instances as this increases the system efficiency in available resource usage. Notable implementations [11] are based on a simple Round-Robin assignment strategy that guarantees each operator instance will receive the same number of input tuples. This approach is effective as long as the time taken by each operator instance to process a single tuple (tuple *execution time*) is the same for any incoming tuple. In this case, all parallel instances of the same operator will experience over time, on average, the same load.

This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

However, such assumption (*i.e.*, same execution time for all tuples of a stream) does not hold for many practical use cases. The tuple execution time, in fact, may depend on the tuple content itself. This is often the case whenever the receiving operator implements a logic with branches where only a subset of the incoming tuples travels through each single branch. If the computation associated with each branch generates different loads, then the execution time will change from tuple to tuple. As a practical example consider an operator that works on a stream of input tweets and that enriches them with historical data extracted from a database, where this historical data is added only to tweets that contain specific hashtags: only tuples that get enriched require an access to the database, an operation that typically introduces non negligible latencies at execution time. In this case shuffle grouping implemented with Round-Robin, may produce imbalance between the operator instances, and this typically causes an increase in the time needed for a tuple to be completely processed by the application (tuple *completion time*) as some tuple may end-up being queued on some overloaded operator instances, while other instances are available for immediate processing.

In this paper we introduce *Proactive Online Shuffle Grouping* (POSG) a novel approach to shuffle grouping that aims at reducing tuple completion times by accurately scheduling incoming tuples on available operator instances in order to avoid imbalances. To the best of our knowledge POSG is the first solution that explicitly addresses and solves the above stated problem.

The idea at the basis of POSG is simple: by measuring the amount needed by operator instances to process each tuple we can schedule incoming tuples minimizing the completion time. However, making such idea works in practice in a streaming setting is not trivial. In particular, POSG makes use of sketches to efficiently keep track of the huge amount of information related to tuple execution times and then applies a greedy online multiprocessor scheduling algorithm to assign tuples to operator instances at runtime. The status of each instance is monitored in a smart way in order to detect possible changes in the input load distribution and coherently adapt the scheduling. As a result, POSG provides an important performance gain in terms of tuple completion times with respect to Round-Robin for all those settings where tuple processing times are not constant, but rather depend on the tuple content.

In summary, we provide the following contributions:

- we introduce POSG the first solution for shuffle grouping that explicitly addresses the problem of parallel operator instances imbalance under loads characterized by non-uniform tuple execution times; POSG schedules tuple on target operator instances online, with minimal resource usage; it works at runtime and is able to continuously adapt to changes in the input load;
- We study the two components of our solution: (**i**) showing that the scheduling algorithm efficiently approximate the optimal one and (**ii**) providing some error bounds as well as a probabilistic analysis of the accuracy of the tuple execution time tracking algorithm;
- we evaluate POSG's sensibility to both the load characteristic and its configuration parameters with an extensive simulation-based evaluation that points the scenarios where POSG is expected to provide its best performance;
- we evaluate POSG's performance by integrating a prototype implementation with the Apache Storm stream processing framework on Microsoft Azure platform and running it on a real dataset.

After this introduction, the paper starts by defining a system model and stating the problem we intend to attack (Section II); it then introduces POSG (Section III) and shows the results of our probabilistic analysis (Section IV); results from our experimental campaign are reported in Section V and are followed by a discussion of the related works (Section VI); finally, Section VII concludes the paper.

II. System Model

We consider a distributed stream processing system (SPS) deployed on a cluster where several computing nodes exchange data through messages sent over a network. The SPS executes a stream processing application represented by a *topology*: a direct acyclic graph interconnecting operators, represented by nodes, with data streams (DS), represented by edges. Each topology contains at least a *source*, *i.e.,* an operator connected only through outbound DSs, and a *sink*, *i.e.,* an operator connected only through inbound DSs. Each operator O can be parallelized by creating k independent instances $O_0, \cdots, O_{k-1}$ of it and by partitioning its inbound stream O^{in} in k sub-streams $O_0^{in}, \cdots, O_{k-1}^{in}$. Tuples are assigned to sub-streams with a *grouping* function. Several grouping strategies are available, but in this work we restrict our analysis to *shuffle grouping* where each incoming tuple can be assigned to any sub-stream.

Data injected by the source is encapsulated in units called tuples and each data stream is an unbounded sequence of tuples. Without loss of generality, here we assume that each tuple t is a finite set of key/value pairs that can be customized to represent complex data structures. To simplify the discussion, in the rest of this work we deal with streams of unary tuples with a single non negative integer value. For the sake of clarity, and without loss of generality, here we restrict our model to a topology with an operator S (*scheduler*) which schedules the tuples of a DS O^{in} consumed by the instances $O_0, \cdots, O_{k-1}$ of operator O.

We denote as $w_{t,i}$ the execution time of tuple t on operator instance O_i. Abusing the notation, we may omit the instance identifier subscript. We assume that the execution time $w_{t,i}$ depends on the content of the tuple t: $w_{t,i} = g_i(t)$, where g_i is an unknown function that may be different for each operator instance. We simplify the model assuming that the functions $g_i(t)$ depend on a single fixed and known attribute value of t. The probability distribution of such attribute values, as well as the functions $g_i(t)$ are unknown and may change over time. However, we assume that subsequent changes are interleaved by a large enough time frame such that an algorithm may have a reasonable amount of time to adapt.

Let $l(i)$ be the completion time of the i-th tuple of the stream, *i.e.,* the time it takes for the i-th tuple from the moment it is injected by the source in the topology, till the moment the last operator concludes processing it. Then we can define the average completion time as: $\overline{L} = \sum_{i \in [m]} l(i)/m$.

The general goal we target in this work is to minimize the average tuple completion time $\overline{L}$. Such metric is fundamentally driven by three factors: (**i**) tuple execution times at operator instances, (**ii**) network latencies and (**iii**) queuing delays. More in detail, we aim at reducing queuing delays at parallel operator instances that receive input tuples through shuffle grouping.

Typical implementation of shuffle grouping are based on Round-Robin scheduling. However, this tuple to DS sub-streams assignment strategy may introduce additional queuing delays when the execution time of input tuples is not constant. For instance, let a_0, b_1, a_2 be a stream with an inter tuple arrival delay of 1s, where a and b are tuples with the following execution time: $w_a = 10$s and $w_b = 1$s. Scheduling this stream with Round-Robin on $k = 2$ operator instances would assign a_0 and a_2 to instance 1 and b_1 to instance 2, with a cumulated completion time equal to $10+1+10+(10-2) = 29$s (*i.e.,* 8s wasted for the additional queuing delay of a_2). A better schedule would be to assign a_0 to instance 1, while b_1 and a_2 to instance 2. In this case, the cumulated completion time equals to $10 + 1 + 10 = 21$s (*i.e.,* no queuing delay).

III. Proactive Online Shuffle Grouping

Proactive Online Shuffle Grouping is a shuffle grouping implementation based on a simple, yet effective idea: if we assume to know the execution time $w_{t,i}$ of each tuple t on the available operator instances $O_0, \cdots, O_{k-1}$, we can schedule the execution of incoming tuples on such instances with the aim of minimizing the average per tuple completion time at the operator instances. However, the value of $w_{t,i}$ is generally unknown. A common solution to this problem is to build a cost model for tuple execution and then use it to proactively schedule incoming load. However building an accurate cost model usually requires a large amount of *a priori* knowledge on the system. Furthermore, once a model has been built, it can be hard to handle changes in the system or input stream characteristics at runtime. Another common alternative is to periodically collect at the scheduler the load of the operator instances. However, this solution only allows for *reactive* scheduling, where input tuples are scheduled on the basis of a previous, possibly stale, load state of the operator instances; reactive scheduling typically impose a periodic overhead even if the load distribution imposed by input tuples does not change over time.

To overcome all these issues, POSG takes decision based on the estimation $\widehat{\mathcal{C}}_i$ of the execution time assigned to instance i: $\mathcal{C}_i = \sum_{\forall t \in O_i^{in}} w_{t,i}$. In order to do so, POSG computes an estimation $\hat{w}_{t,i}$ of the execution time $w_{t,i}$ of each tuple t on each operator instance i. Then, POSG can also compute the sum of the estimated execution times of the tuples assigned to an instance i, *i.e.,* $\widehat{\mathcal{C}}_i = \sum_{\forall t \in O_i^{in}} \hat{w}_{t,i}$, which in turn is the estimation of $\mathcal{C}_i$. A greedy scheduling algorithm (Section III-A) is then fed with estimations for all the available operator instances. To enable this approach, each operator instance builds a sketch (*i.e.,* a memory efficient data structure) that will track the execution time of the tuples it process. When a change in the stream or instance(s) characteristics affects the tuples execution times on some instances, the concerned instance(s) will forward an updated sketch to the scheduler which will than be able to (again) correctly estimate the tuples execution times. This solution does not require any *a priori* knowledge on the stream or system, and is designed to continuously adapt to changes in the input distribution or

on the instances load characteristics. In addition, this solution is *proactive*, namely its goal is to avoid unbalance through scheduling, rather than detecting the unbalance and then attempting to correct it.

A. Background

Data Streaming model — We present the data stream model [9], under which we analyze our algorithms and derive bounds. A stream is an unbounded sequence of elements $\sigma = \langle t_1, \ldots, t_m \rangle$ called tuples or items, which are drawn from a large universe $[n] = \{1, \ldots, n\}$. The unknown size (or length) of the stream is denoted by m. We denote by p_t the unknown probability of occurrence of item t in the stream and with f_t the unknown frequency[1] of item t, *i.e.,* the number of occurrences of t in the stream of size m.

2-Universal Hash Functions — Our algorithm uses hash functions randomly picked from a 2-universal hash functions family. A collection $\mathcal{H}$ of hash functions $h : \{1, \ldots, n\} \to \{0, \ldots, c\}$ is said to be 2-universal if for every two different items $x, y \in [n]$, for all $h \in \mathcal{H}$, $\mathbb{P}\{h(x) = h(y)\} \leq \frac{1}{c}$, which is the probability of collision obtained if the hash function assigned truly random values to any $x \in [c]$. Carter and Wegman [4] provide an efficient method to build large families of hash functions approximating the 2-universality property.

`Count Min` sketch algorithm — Cormode and Muthukrishnan have introduced in [5] the `Count Min` sketch that provides, for each item t in the input stream an (ε, δ)-additive-approximation $\hat{f}_t$ of the frequency f_t. The `Count Min` sketch consists of a two dimensional matrix $\mathcal{F}$ of size $r \times c$, where $r = \lceil \log \frac{1}{\delta} \rceil$ and $c = \lceil \frac{e}{\varepsilon} \rceil$. Each row is associated with a different 2-universal hash function $h_i : [n] \to [c]$. When the `Count Min` algorithm reads sample t from the input stream, it updates each row: $\forall i \in [r], \mathcal{F}[i, h_i(t)] \leftarrow \mathcal{F}[i, h_i(t)] + 1$. Thus, the cell value is the sum of the frequencies of all the items mapped to that cell. Upon request of f_t estimation, the algorithm returns the smallest cell value among the cell associated with t: $\hat{f}_t = \min_{i \in [r]} \{\mathcal{F}[i, h_i(t)]\}$.

Fed with a stream of m items, the space complexity of this algorithm is $O(\frac{1}{\varepsilon} \log \frac{1}{\delta} (\log m + \log n))$ bits, while update and query time complexities are $O(\log 1/\delta)$. The `Count Min` algorithm guarantees that the following bound holds on the estimation accuracy for each iteam read from the input stream: $\mathbb{P}\{| \hat{f}_t - f_t | \geq \varepsilon(m - f_t)\} \leq \delta$, while $f_t \leq \hat{f}_t$ is always true.

This algorithm can be easily generalized to provide (ε, δ)-additive-approximation of point queries $\mathcal{Q}t$ on stream of udpates, *i.e.,* a stream where each item t carries a positive integer update value v_t. When the `Count Min` algorithm reads the pair $\langle t, v_t \rangle$ from the input stream, the update routine changes as follows: $\forall i \in [r], \mathcal{F}[i, h_i(t)] \leftarrow \mathcal{F}[i, h_i(t)] + v_t$.

Greedy Online Scheduler — A classical problem in the load balancing literature is to schedule independent tasks on identical machines minimizing the makespan, *i.e.,* the *Multiprocessor Scheduling* problem. In this paper we adapt this problem to our setting, *i.e.,* to schedule *online* independent tasks on non uniform machines aiming to minimizing the average per task completion time $\overline{L}$. Online scheduling means that the scheduler does not know in advance the sequence of tasks it has to schedule. The Greedy Online Scheduler algorithm assigns the currently submitted tasks to the less loaded available machine. In Section IV-A we prove that this algorithm closely approximates the optimal algorithm.

B. POSG design

Each operator instance *op* maintains two `Count Min` sketch matrices (Figure 1.A): the first one, denoted as $\mathcal{F}_{op}$, tracks the tuple frequencies $f_{t,op}$; the second, denoted as $\mathcal{W}_{op}$, tracks the tuples cumulated execution times $W_t = w_{t,op} \times f_{t,op}$. Both `Count Min` matrices share the same sizes and hash functions. The latter is the generalized version of the `Count Min` presented in Section III-A where the update value is the tuple execution time when processed by the instance (*i.e.,* $v_t = w_{t,op}$) . The operator instance will update (Listing III.1) both matrices after each tuple execution.

The operator instances are modelled as a finite state machines (Figure 2) with two states: START and STABILIZING. The START state lasts until instance *op* has executed N tuples, where N is a user defined window size

[1]This definition of *frequency* is compliant with the data streaming literature.

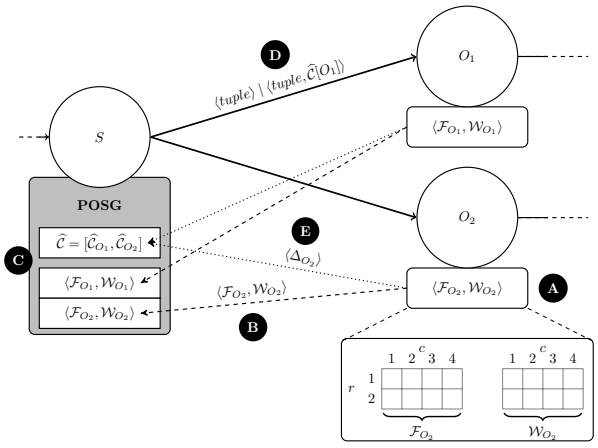

Fig. 1. Proactive Online Shuffle Grouping design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$) and $k = 2$.

Listing III.1: Operator instance op: update $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$.

```
1: init do
2:     F_op matrix of size r × c
3:     W_op matrix of size r × c
4:     r hash functions h_1 ... h_r : [n] → [c] from a 2-universal family (same for all instances).
5: end init
6: function UPDATE(tuple : t, execution time : l)
7:     for i = 1 to c do
8:         F_op[i, h_i(t)] ← F_op[i, h_i(t)] + 1
9:         W_op[i, h_i(t)] ← W_op[i, h_i(t)] + l
10:    end for
11: end function
```

parameter. The transition to the STABILIZING state (Figure 2.A) triggers the creation of a new snapshot $\mathcal{S}_{op}$. A snapshot is a matrix of size $r \times c$ where $\forall i \in [r], j \in [c] : \mathcal{S}_{op}[i,j] = \mathcal{W}_{op}[i,j]/\mathcal{F}_{op}[i,j]$. We say that the $\mathcal{F}$ and $\mathcal{W}$ matrices are stable when the relative error η_{op} between the previous snapshot and the current one is smaller than μ: if

$$\eta_{op} = \frac{\sum_{\forall i,j} |\mathcal{S}_{op}[i,j] - \frac{\mathcal{W}_{op}[i,j]}{\mathcal{F}_{op}[i,j])}|}{\sum_{\forall i,j} \mathcal{S}_{op}[i,j]} \leq \mu \tag{1}$$

is satisfied. Then, each time instance op has executed N tuples, it checks whether Equation 1 is satisfied. **(i)** If not, then $\mathcal{S}_{op}$ is updated (Figure 2.B). **(ii)** Otherwise the operator instance sends the $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices to the scheduler (Figure 1.B), resets them and moves back to the START state (Figure 2.C).

There is a delay between any change in the stream or operator instances characteristics and when the scheduler receives the updated $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices from the affected operator instance(s). This introduces a skew in the cumulated execution times estimated by the scheduler. In order to compensate for this skew, we introduce

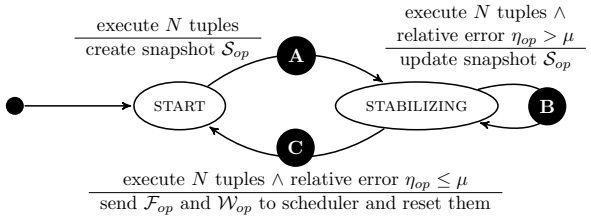

Fig. 2. Operator instance finite state machine.

a synchronization mechanism that kicks in whenever the scheduler receives a new pair of matrices from any operator instance. Notice also that there is an initial transient phase in which the scheduler has not yet received any information from operator instances. This means that in this first phase it has no information on the tuples execution times and is forced to use the Round-Robin policy. This mechanism is thus also needed to initialize the estimated cumulated execution times when the Round-Robin phase ends.

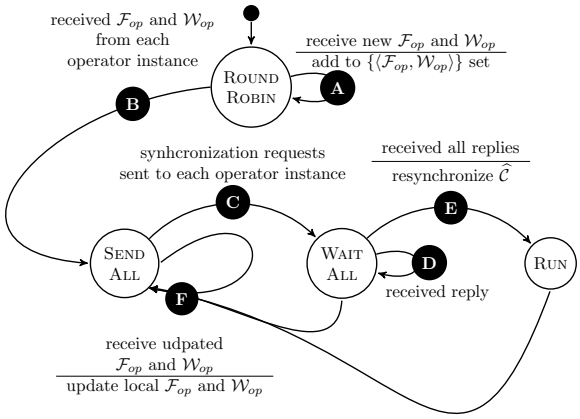

Fig. 3. Scheduler finite state machine.

The scheduler (Figure 1.C) maintains the estimated cumulated execution time for each instance, in a vector $\widehat{\mathcal{C}}$ of size k, and the set of pairs of matrices: $\{\langle \mathcal{F}_{op}, \mathcal{W}_{op} \rangle\}$, initially empty. The scheduler is modelled as a finite state machine (Figure 3) with four states: ROUND ROBIN, SEND ALL, WAIT ALL and RUN.

The ROUND ROBIN state is the initial state in which scheduling is performed with the Round-Robin policy. In this state, the scheduler collects the $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices sent by the operator instances (Figure 3.A). After receiving the two matrices from each instance (Figure 3.B), the scheduler is able to estimate the execution time for each submitted tuple and moves into the SEND ALL state. SEND ALL state goal is to send the synchronization requests towards the k instances. To reduce overhead, requests are piggy backed (Figure 1.D) with outgoing tuples applying the Round-Robin policy for the next k tuples: the i-th tuple is assigned to operator instance $i \mod k$. On the other hand, the estimated cumulated execution time vector $\widehat{\mathcal{C}}$ is updated with the tuple estimated execution time using the UPDATE$\widehat{\mathcal{C}}$ function (Listing III.2). When all the requests have been sent (Figure 3.C), the scheduler moves into the WAIT ALL state. This state collects the synchronization replies from the operator instances (Figure 3.D). Operator instance *op*'s reply (Figure 1.E) contains the difference Δ_{op} between the instance cumulated execution time $\mathcal{C}_{op}$ and the scheduler estimation $\widehat{\mathcal{C}}[op]$. In the WAIT ALL state, scheduling is performed as in the RUN state, using both the SUBMIT and the UPDATE$\widehat{\mathcal{C}}$ functions (Listing III.2). When all the replies for the current epoch have been collected, synchronization is performed and the scheduler moves in the RUN state (Figure 3.E). In the RUN state, the scheduler assigns the input tuple applying the Greedy Online Scheduler algorithm, *i.e.,* assigns the tuple to the operator instance with the least estimated cumulated execution time (SUBMIT function, Listing III.2). Then it increments the target instance estimated cumulated execution time with the estimated tuple execution time (UPDATE$\widehat{\mathcal{C}}$ function, Listing III.2). Finally, in any state except ROUND ROBIN, receiving an updated pair of matrices $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ moves the scheduler into the SEND ALL state (Figure 3.F).

Theorem 3.1 (Time complexity of POSG):
For each tuple read from the input stream, the time complexity of POSG for each instance is $\mathcal{O}(\log 1/\delta)$. For each tuple submitted to the scheduler, POSG time complexity is $\mathcal{O}(k + \log 1/\delta)$.

Proof: By Listing III.1, for each tuple read from the input stream, the algorithm increments an entry per row of both the $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices. Since each has $\log 1/\delta$ rows, the resulting update time complexity is $\mathcal{O}(\log 1/\delta)$ By Listing III.2, for each submitted tuple, the scheduler has to retrieve the index with the smalled value in the vector $\widehat{\mathcal{C}}$ of size k, and to retrieve the estimated execution time for the submitted tuple. This operation requires to read entry per row of both the $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices. Since each has $\log 1/\delta$ rows, the resulting update time complexity is $\mathcal{O}(k + \log 1/\delta)$ ■

Listing III.2: Scheduler: submit t and update $\widehat{\mathcal{C}}$.

```
1: init do
2:     vector Ĉ of size k
3:     A set of ⟨F_op, W_op⟩ matrices pairs
4:     Same hash functions h_1 ... h_r of the operator instances
5: end init
6: function SUBMIT(tuple : t)
7:     return arg min_{op∈[k]} {Ĉ[op]}
8: end function
9: function UPDATEĈ(tuple : t, operator : op)
10:    i ← arg min_{i∈[r]} {F_op[i, h_i(t)]}
11:    Ĉ[op] ← Ĉ[op] + (W_op[i, h_i(t)]/F_op[i, h_i(t)])
12: end function
```

Theorem 3.2 (Space Complexity of POSG):
The space complexity of POSG for the operator instances is $\mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$, while the space complexity for the scheduler is $\mathcal{O}\left(k\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$.

Proof: Each operator instance stores two matrices of size $\log\frac{1}{\delta}\times\frac{e}{\varepsilon}$ of counters of size $\log m$. In addition, it also stores a hash function with a domain of size n. Then the space complexity of POSG on each operator instance is $\mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$. The scheduler stores the same matrices, one for each instance, as well as a vector of size k. Then the space complexity of POSG on the scheduler is $\mathcal{O}\left(k\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$. ■

Theorem 3.3 (Communication complexity of POSG):
The communication complexity of POSG is of $\mathcal{O}\left(\frac{m}{N}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)+k\log m\right)\right)$ bit and $\mathcal{O}\left(k\frac{m}{N}\right)$ messages.

Proof: After executing N tuples, an operator instance may send the $\mathcal{F}_{op},\mathcal{W}_{op}$ matrices to the scheduler. This generates a communication cost of $\mathcal{O}\left(\frac{m}{kN}\right)$ messages and $\mathcal{O}\left(\frac{m}{kN}\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)\right)$ bits. When the scheduler receives these matrices, the synchronization mechanism kicks in and triggers a round trip communication (half of which is piggy backed by the tuples) with each instance. The communication cost of the synchronization mechanism $\mathcal{O}\left(\frac{m}{N}\right)$ messages and $\mathcal{O}\left(\frac{m}{N}\log m\right)$ bits. Since there are k instances, the overall communication complexity is $\mathcal{O}\left(k\frac{m}{N}\right)$ messages and $\mathcal{O}\left(\frac{m}{N}\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}(\log m+\log n)+k\log m\right)\right)$ bits ■

Note that the communication cost is negligible since the window size N should be chosen such that $N\gg k$ (*e.g.,* in our tests we have $N=1024$ and $k\leq 10$).

IV. Theoretical Analysis

This section provide the analysis of the quality of the scheduling performed by POSG in two steps. First we study the Greedy Online Scheduler algorithm approximation. Then, in Section IV-B we provide a probabilistic analysis of the mechanism that POSG uses to estimate the tuple execution times.

A. Online Greedy Scheduler

We suppose that tuples cannot be preempted, that is tuples must be processed in an uninterrupted fashion on the operator instance it has been scheduled on. As mentioned, we assume that the processing time w_{t_j} is known for each tuple t_j. Finally, given our system model, the problem of minimizing the average completion time $\overline{L}$ can be reduced to the following problem (in terms of *makespan*):

Problem *4.1:* Given k identical operator instances, and a sequence of tuples $\sigma=\langle t_1,\ldots,t_m\rangle$ that arrive online from the input stream. Find an online scheduling algorithm that minimizes the makespan of the schedule produced by the online algorithm when fed with σ.

Let OPT be the schedule algorithm that minimizes the makespan over all possible sequences σ, and $\mathcal{C}^{\sigma}_{OPT}$ denote the makespan of the schedule produced by the OPT algorithm fed by sequence σ. Notice that finding $\mathcal{C}^{\sigma}_{OPT}$ is an NP-hard problem. We will show that the Greedy Online Scheduler (GOS) algorithm defined in Section III-A builds a schedule that is within some factor of the lower bound of the quality of the optimal scheduling algorithm OPT. Let us denote by $\mathcal{C}^{\sigma}_{\text{GOS}}$ the makespan of the schedule produced by the greedy algorithm fed with σ.

Theorem 4.2: For any σ, we have $\mathcal{C}^{\sigma}_{\text{GOS}}\leq(2-1/k)C^{\sigma}_{OPT}$.

Proof: Let $i \in [k]$ be the instance on which the last tuple t_j is executed. By construction of the algorithm, when tuple t_j starts its execution on instance i, all the other instances are busy, otherwise t_j would have been executed on another instance. Thus when tuple t_j starts its execution on instance i, each of the k instances must have been allocated a load at least equivalent to $(\sum_{\ell=1}^{m} w_{t_\ell} - w_{t_j})/k$. Thus we have,

$$\begin{aligned} C^{\sigma}_{\text{GOS}} - w_{t_j} &\leq \frac{\sum_{\ell=1}^{m} w_{t_\ell} - w_{t_j}}{k} \\ C^{\sigma}_{\text{GOS}} &\leq \frac{\sum_{\ell=1}^{m} w_{t_\ell}}{k} + w_{t_j}(1 - \frac{1}{k}) \end{aligned} \tag{2}$$

Now, it is easy to see that

$$C^{\sigma}_{\text{OPT}} \geq \frac{\sum_{j=1}^{m} w_{t_j}}{k}, \tag{3}$$

otherwise the total load processed by all the machines in the schedule produced by the OPT algorithm would be strictly less than $\sum_{j=1}^{m} w_{t_j}$, leading to a contradiction. We also trivially have

$$C^{\sigma}_{\text{OPT}} \geq \max_{\ell} w_{t_\ell}. \tag{4}$$

Thus combining relations (2), (3), and (4), we have

$$\begin{aligned} C^{\sigma}_{\text{GOS}} &\leq C^{\sigma}_{\text{OPT}} + C^{\sigma}_{\text{OPT}}(1 - \frac{1}{k}) \\ &= (2 - \frac{1}{k}) C^{\sigma}_{\text{OPT}} \end{aligned} \tag{5}$$

that concludes the proof. ■

This lower bound is tight, that is there are sequences of tuples for which the Greedy Online Scheduler algorithm produces a schedule whose completion time is exactly equal to $(2 - 1/k)$ times the completion time of the optimal scheduling algorithm [7].

Consider the example of $a(a-1)$ tuples whose processing time is equal to $w_{\max}/k$ and one task with a processing time equal to $w_{\max}$. Suppose that the $a(a-1)$ tuples are scheduled first and then the longest one. Then the greedy algorithm will exhibit a makespan equal to $(m-1)w_{\max}/k + w_{\max} = w_{\max}(2 - 1/k)$ while the OPT scheduling will lead to a makespan equal to $w_{\max}$.

B. Execution Time Estimation

POSG uses two matrices, $\mathcal{F}$ and $\mathcal{W}$, to estimate the execution time w_t of each tuple submitted to the scheduler. To simplify the discussion, we consider a single operator instance.

From the count Min algorithm, and for any $v \in \{1, \ldots, n\}$, we have for a given hash function h_i,

$$C_v(m) = \sum_{u=1}^{n} f_u 1_{\{h_i(u)=h_i(v)\}} = f_v + \sum_{u=1, u\neq v}^{n} f_u 1_{\{h_i(u)=h_i(v)\}}.$$

and

$$W_v(m) = f_v w_v + \sum_{u=1, u\neq v}^{n} f_u w_u 1_{\{h_i(u)=h_i(v)\}},$$

where $C_v(m) = \mathcal{F}[v][h_v(m)]$ and $W_v(m) = \mathcal{W}[v][h_v(m)]$. Let us denote respectively by $w_{\min}$ and $w_{\max}$ the minimum and the maximum execution time of the items. We have trivially

$$w_{\min} \leq \frac{W_v(m)}{C_v(m)} \leq w_{\max}.$$

In the following we write respectively C_v and W_v instead of $C_v(m)$ and $W_v(m)$, to simplify the writing. For any $i = 0, \ldots, n-1$, we denote by $U_i(v)$ the set which elements are the subsets $\{1, \ldots, n\} \setminus \{v\}$ whose size is equal to i, that is

$$U_i(v) = \{A \subseteq \{1, \ldots, n\} \setminus \{v\} \mid |A| = i\}.$$

We have $U_0(v) = \{\emptyset\}$.

For any $v = 1, \ldots, n$, $i = 0, \ldots, n-1$ and $A \in U_i(v)$, we introduce the event $B(v, i, A)$ defined by

$$B(v, i, A) = \{h_u = h_v,\ \forall u \in A \text{ and } h_u \neq h_v,\ \forall u \in \{1, \ldots, n\} \setminus (A \cup \{v\})\}.$$

From the independence of the h_u, we have

$$\Pr\{B(v, i, A)\} = \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i}.$$

Let us consider the ratio W_v/C_v. For any $i = 0, \ldots, n$, we define

$$R_i(v) = \left\{ \frac{f_v w_v + \sum_{u \in A} f_u w_u}{f_v + \sum_{u \in A} f_u},\ A \in U_i(v) \right\}.$$

We have $R_0(v) = \{w_v\}$. We introduce the set $R(v)$ defined by

$$R(v) = \bigcup_{i=0}^{n-1} R_i(v).$$

Thus with probability 1,

$$W_v/C_v \in R(v).$$

Let $x \in R(v)$. We have

$$\begin{aligned}
\Pr\{W_v/C_v = x\} &= \sum_{i=0}^{n-1} \sum_{A \in U_i(v)} \Pr\{W_v/C_v = x \mid B(v, i, A)\} \Pr\{B(v, i, A)\} \\
&= \sum_{i=0}^{n-1} \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i} \sum_{A \in U_i(v)} \Pr\{W_v/C_v = x \mid B(v, i, A)\} \\
&= \sum_{i=0}^{n-1} \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i} \sum_{A \in U_i(v)} 1_{\{x = (f_v w_v + \sum_{u \in A} f_u w_u)/(f_v + \sum_{u \in A} f_u)\}}.
\end{aligned}$$

Thus

$$\begin{aligned}
\mathbb{E}\{W_v/C_v\} &= \sum_{i=0}^{n-1} \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i} \sum_{A \in U_i(v)} \sum_{x \in R(v)} x 1_{\{x = (f_v w_v + \sum_{u \in A} f_u w_u)/(f_v + \sum_{u \in A} f_u)\}} \\
&= \sum_{i=0}^{n-1} \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i} \sum_{A \in U_i(v)} \frac{f_v w_v + \sum_{u \in A} f_u w_u}{f_v + \sum_{u \in A} f_u} \sum_{x \in R(v)} 1_{\{x = (f_v w_v + \sum_{u \in A} f_u w_u)/(f_v + \sum_{u \in A} f_u)\}} \\
&= \sum_{i=0}^{n-1} \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i} \sum_{A \in U_i(v)} \frac{f_v w_v + \sum_{u \in A} f_u w_u}{f_v + \sum_{u \in A} f_u}.
\end{aligned}$$

Let us assume that all the f_u are equal, that is for each u, we have $f_u = m/n$. Simulations tend to show that the worst cases scenario of input streams are exhibited when all the items show the same number of occurrences in the input stream. We get

$$\Pr\{W_v/C_v = x\} = \sum_{i=0}^{n-1} \left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{n-1-i} \sum_{A \in U_i(v)} 1_{\{x = (w_v + \sum_{u \in A} w_u)/(i+1)\}}.$$

We define $S = \sum_{i=1}^{n} w_i$. We then have

Theorem 4.3:

$$\mathbb{E}\{W_v/C_v\} = \frac{S - w_v}{n-1} - \frac{k(S - n w_v)}{n(n-1)} \left(1 - \left(1 - \frac{1}{k}\right)^n\right).$$

It important to note that this result does not depend on m.

Let us now consider a numerical application. We take $k = 55$, $n = 4096$ and the distinct values of w_u equal to $1, 2, 3, \ldots, 64$, each item being present 64 times in the input stream, we get for $v = 1, \ldots, 64$, $\mathbb{E}\{W_v/C_v\} \in [32.08, 32.92]$. Note also from above that we have $1 \leq W_v/C_v \leq 64$.

From the Markov inequality we have, for every $x > 0$,

$$\Pr\{W_v/C_v \geq x\} \leq \frac{\mathbb{E}\{W_v/C_v\}}{x}.$$

By taking $x = 64a$, with $a \in [0.6, 1)$, we obtain

$$\Pr\{W_v/C_v \geq 64a\} \leq \frac{\mathbb{E}\{W_v/C_v\}}{64a} \leq \frac{33}{64a}.$$

If r denotes the number of rows of the system, we have by the independence of the h functions,

$$\Pr\{\min_{t=1,\ldots,r}(W_{t,v}/C_{t,v}) \geq 64a\} = (\Pr\{W_v/C_v \geq 64a\})^r \leq \left(\frac{33}{64a}\right)^r.$$

By taking for instance $a = 3/4$ and $r = 10$, we get

$$\Pr\{\min_{t=1,\ldots,r}(W_{t,v}/C_{t,v}) \geq 48\} \leq \left(\frac{11}{16}\right)^{10} \leq 0.024.$$

V. Experimental Evaluation

In this section we evaluate the performance obtained by using POSG to perform shuffle grouping. We will first describe the general setting used to run the tests and will then discuss the results obtained through simulations (Section V-B) and with a prototype of POSG targeting Apache Storm (Section V-C).

A. Setup

Datasets — In our tests we consider both synthetic and real datasets. For synthetic datasets we generate streams of integer values (items) representing the values of the tuple attribute driving the execution time when processed on an operator instance. We consider streams of $m = 32,768$ tuples, each containing a value chosen among $n = 4,096$ distinct items. Synthetic streams have been generated using the Uniform distribution and Zipfian distributions with different values of $\alpha \in \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0\}$, denoted respectively as Zipf-0.5, Zipf-1.0, Zipf-1.5, Zipf-2.0, Zipf-2.5, and Zipf-3.0. We define w_n as the number of distinct execution time values that the tuples can have. These w_n values are selected at *constant* distance in the interval $[w_{min}, w_{max}]$. We have also run tests generating the execution time values in the interval $[w_{min}, w_{max}]$ with geometric steps without noticing unpredictable differences with respect to the results reported in this section. The algorithm parameters are the operator window size N, the tolerance parameter μ, and the parameters of the matrices $\mathcal{F}$ and $\mathcal{W}$: ε and δ.

Unless otherwise specified, the frequency distribution is Zipf-1.0 and the stream parameters are set to $w_n = 64$, $w_{min} = 1$ ms and $w_{max} = 64$ ms, this means that the execution times are picked in the set $\{1, 2, \cdots, 64\}$. The algorithm parameters are set to $N = 1024$, $\mu = 0.05$, $\varepsilon = 0.05$ (*i.e.,* $c = 54$ columns) and $\delta = 0.1$ (*i.e.,* $r = 4$ rows). If not stated otherwise, the operator instances are uniform (*i.e.,* a tuple has the same execution time on any instance) and there are $k = 5$ instances. Let $\overline{W}$ be the average execution time of the stream tuples, then the stream maximum theoretical input throughput sustainable by the setup is equal to $k/\overline{W}$. When fed with an input throughput smaller than $k/\overline{W}$ the system will be over-provisioned (*i.e.,* possible underutilization of computing resources). Conversely, an input throughput larger than $k/\overline{W}$ will result in an undersized system. We refer to the ratio between the maximum theoretical input throughput and the actual input throughput as the percentage of over-provisioning that, unless otherwise stated, was set to 100%.

In order to generate 100 different streams, we randomize the association between the w_n execution time values and the n distinct items: for each of the w_n execution time values we pick *uniformly* at random n/w_n different values in $[n]$ that will be associated to that execution time value. This means that the 100 different streams we use in our tests do not share the same association between execution time and item as well as the association between frequency and execution time (thus each stream has also a different average execution time $\overline{W}$). We have also

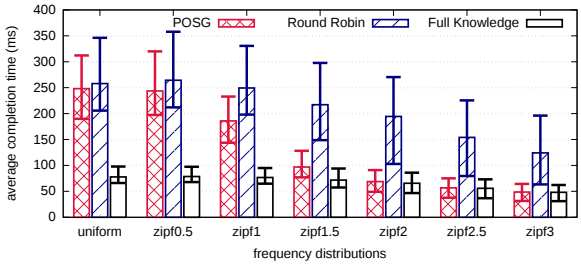

Fig. 4. Average per tuple completion time $\overline{L}$ with different frequency probability distributions

build these associations using other distributions, namely geometric and binomial, without noticing unpredictable differences with respect to the results reported in this section.

We retrieved a dataset containing a stream of preprocessed tweets related to Italian politicians crawled during the 2014 European elections. Among other information, the tweets are enriched with a field *mention* containing the *entities* mentioned in the tweet. These entities can be easily classified into *politicians*, *media* and *others*. We consider the first $500,000$ tweets, mentioning roughly $n = 35,000$ distinct entities and where the most frequent entity ("Beppe Grillo") has an empirical probability of occurrence equal to 0.065.

Evaluation Metrics — The evaluation metrics we provide are (**i**) the average per tuple completion time: $\overline{L}^{alg}$ (simply *average completion time* in the following), where alg is the algorithm used for scheduling, and (**ii**) the average per tuple completion time speed up (simply *speed up* in the following) achieved by POSG with respect to Round-Robin: S_L. Recall that $l^{alg}(i)$ is the completion time of the i-th tuple of the stream when using the scheduling algorithm alg. Then we can define the average completion time $\overline{L}^{alg}$ and speed up S_L as follows:

$$\overline{L}^{alg} = \frac{\sum_{i\in[m]} l^{alg}(i)}{m} \text{ and } S_L = \frac{\sum_{i\in[m]} l^{\text{Round-Robin}}(i)}{\sum_{i\in[m]} l^{\text{POSG}}(i)}$$

Whenever applicable we provide the maximum, mean and minimum figures over the 100 executions.

B. Simulation Results

Frequency Probability Distribution — Figure 4 shows the average completion time $\bar{L}$ for POSG, Round-Robin and Full Knowledge with different frequency probability distributions. The Full Knowledge algorithm represents an ideal execution of the Online Greedy Scheduling algorithm when fed with the exact execution time for each tuple. Increasing the skewness of the distribution reduces the number of distinct tuples that, with high probability, will be fed for scheduling, thus simplifying the scheduling process. This is why all algorithms perform better with highly skewed distributions. On the other hand, uniform or lightly skewed (*i.e.,* Zipf-0.5) distributions seem to be worst cases, in particular for POSG and Round-Robin. With all distributions the Full Knowledge algorithm outperforms POSG which, in turn, always provide better performance than Round-Robin. However, for uniform or lightly skewed distributions (*i.e.,* Zipf-0.5), the gain introduced by POSG is limited (in average 6%). Starting with Zipf-1.0 the gain is much more sizeable (25%) and with Zipf-1.5 we have that the maximum average completion time of POSG is smaller than the minimum average completion time of Round-Robin. Finally, with Zipf-2.5 POSG matches the performance of Full Knowledge. This behavior for POSG stems from the ability of its sketch data structures ($\mathcal{F}$ and $\mathcal{W}$ matrices, see Section III) to capture more useful information for skewed input distributions.

Input Throughput — Figure 5 shows the speed up S_L as a function of the percentage of over-provisioning. When the system is strongly undersized (95% to 98%), queuing delays increase sharply, reducing the advantages offered by POSG. Conversely, when the system is oversized (107 to 115%), queuing delays tend to 0, which in turns also reduces the improvement brought by our algorithm. However, in a correctly sized system (*i.e.,* from 100% to 109%), our algorithm introduces a noticeable speed up, in average at least 1.15 with a peak of 1.26 at 102%. Finally, even when the system is largely oversized (115%), we still provide an average speed up of 1.07.

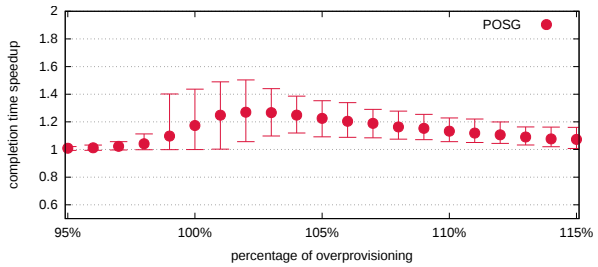

Fig. 5. Speed up S_L as a function of the percentage of over-provisioning

Maximum Execution Time Value — Figure 6 shows the average completion time $\bar{L}$ as a function of the maximum execution time w_{max}. As expected, the average completion time increases with w_{max}. On the other hand, and quite unexpectedly, the gain between the maximum, mean and minimum average completion times of POSG with respect to Round-Robin only slightly improves for growing values of w_{max}, with an average speed up S_L of 1.19.

Number of Execution Time Values — Figure 7 shows the average completion time $\bar{L}$ as a function of the number of execution time values w_n. We can notice that for growing values of w_n both the average completion time values and variance decrease, with only slight changes for $w_n \geq 16$. Recall that w_n is the number of completion time values in the interval $[w_{min}, w_{max}]$ that we assign to the n distinct attribute values. For instance, with $w_n = 2$, all the tuples have a completion time equal to either 1 or 64 ms. Then, assigning either of the two values to the most frequent item strongly affects the average completion time. Increasing w_n reduces the impact that each single execution time has on the average completion time, leading to more stable results. As in the previous plot, the gain between the maximum, mean and maximum average completion times of POSG and Round-Robin (in average 19%) is mostly unaffected by the value of w_n.

Number of operator instances k — Figure 8 shows the speed up S_L as a function of the number of parallel operator instances k. With $k = 1$ the speed up is equal to 1, which is the maximum theoretically achievable speed up with this configuration; this means that our algorithm is efficient as it does not introduce large delays in the average completion time. Increasing the number of instances slightly increases the speed up. However, this growth rapidly converges: there is a 4% increasing from 2 to 3 operators, while from 9 to 10 operators the growth is of only 1%. This result intuitively stems from the fact that moving from $k = 2$ to $k = 3$ we are increasing the available instances by 50%, while moving from $k = 9$ to $k = 10$ only adds $1/9$ more instances.

Precision parameter ε — Figure 9 shows the speed up S_L as a function of the precision parameter ε value that controls the number of columns in the $\mathcal{F}$ and $\mathcal{W}$ matrices. With smaller values of ε POSG is more precise but also uses more memory, *i.e.,* for $\varepsilon = 1.0$ there is a single entry per row, while for $\varepsilon = 0.001$ there are 2781 entries per row. As expected, decreasing ε improves POSG performance: in average a 10 time decrease in ε (thus a 10 time increase in memory) results in a 30% increase in the speed up. Large values of ε do not provide good performance;

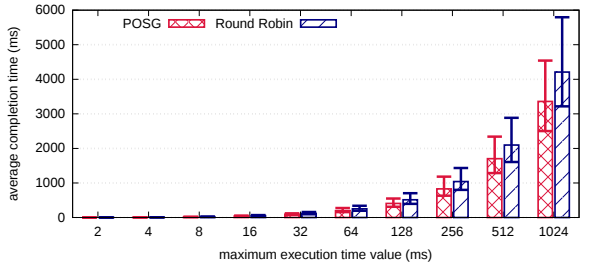

Fig. 6. Average per tuple completion time $\bar{L}$ as a function of the maximum execution time value w_{max}.

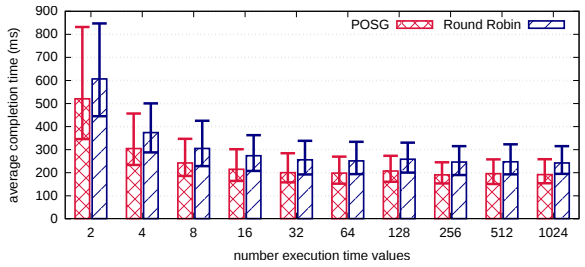

Fig. 7. Average per tuple completion time $\bar{L}$ as a function of the number of execution time values w_n.

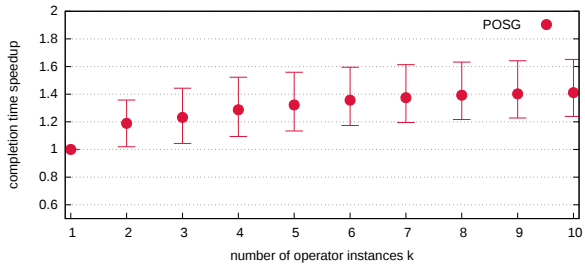

Fig. 8. Speed up S_L as a function of the number of operator instances k.

however, starting with $\varepsilon \leq 0.09$ the minimum average completion time speed up is always larger than 1.

Time Series — Figure 10 shows the completion time as the stream unfolds (the x axis is the number of tuples read from the stream) for both POSG and Round-Robin. Each point on the plot is the maximum, mean and minimum completion time over the previous 2000 tuples. The plot for Round-Robin has been artificially shifted by 1000 tuples to improve readability. In this test the stream is of size $m = 150,000$ split into two periods: the tuple execution times for operator instances 1, 2, 3, 4 and 5 are multiplied by 1.05, 1.025, 1.0, 0.975 and 0.95 respectively for the first $75,000$ tuples, and for the remaining $75,000$ tuples by 0.90, 0.95, 1.0, 1.05 and 1.10 respectively. This setup mimics an abrupt change in the load characteristic of target operator instances (possibly due to exogenous factors).

In the leftmost part of the plot, we can see POSG and Round-Robin provide the same exact results up to $m = 10,690$, where POSG starts to diverge by decreasing the completion time as well as the completion time variance. This behaviour is the result of POSG moving into the RUN state at $m = 10,690$. After this point it starts to schedule using the $\mathcal{F}$ and $\mathcal{W}$ matrices and the Greedy Online Scheduler algorithm, improving its performance with respect to Round-Robin.

At $m = 75,000$ we inject the load characteristic change described above. Immediately POSG performance degrades as the content of $\mathcal{F}$ and $\mathcal{W}$ matrices is outdated. At $m = 84,912$ the scheduler receives the updated $\mathcal{F}$

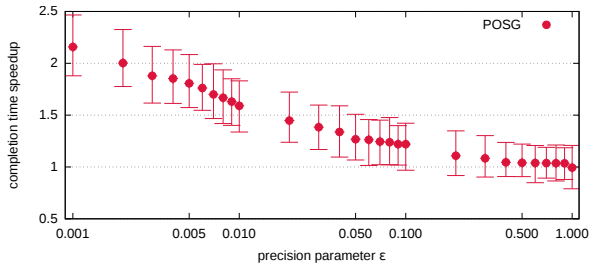

Fig. 9. Speed up S_L as a function of the precision parameter ε (*i.e.,* number of columns $c = e/\varepsilon$).

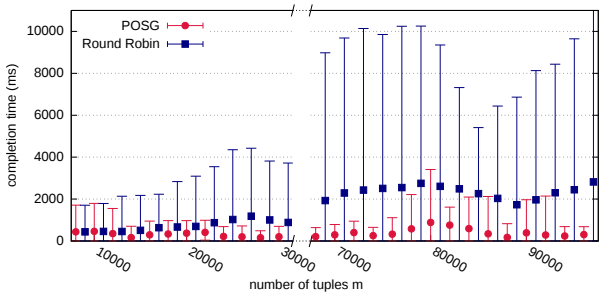

Fig. 10. Simulator per tuple completion time time-series.

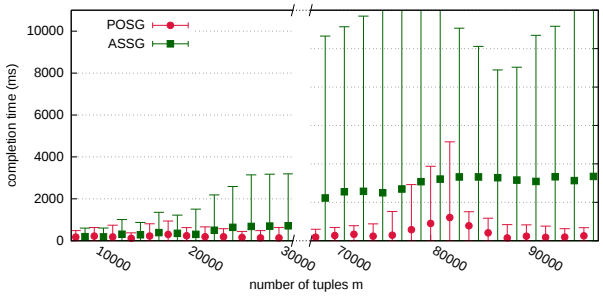

Fig. 11. Prototype per tuple completion time time-series.

and $\mathcal{W}$ matrices and recovers. This demonstrates POSG's ability to adapt at runtime with respect to changes in the load distributions.

C. Prototype

To evaluate the impact of POSG on real applications we implemented it as a custom grouping function within the Apache Storm [11] framework. We have deployed our cluster on Microsoft Azure cloud service, using a Standard Tier A4 VM (4 cores and 7 GB of RAM) for each worker node, each with a single available slot.

The test topology is made of a source (*spout*) and two operators (*bolts*) S and O. The source generates (reads) the synthetic (real) input stream. Bolt S uses either POSG or the Apache Storm standard shuffle grouping implementation (ASSG in the following) to route the tuples toward the k instances (*tasks*) of bolt O.

Time Series — Figure 11 provides results for the prototype with the same settings of the test whose results were reported in Figure 10. We can notice the same general behaviour both in the simulator and in the prototype. In the right part of the plot POSG diverges from ASSG at $m = 20,978$ and decreases the completion time as well as the completion time variance. On the right part of the plot, after $m = 75,000$ POSG performance degrade due to the change in the load distributions. Finally, at $m = 82,311$ the scheduler receives the updated $\mathcal{F}$ and $\mathcal{W}$ matrices and starts to recover. Notice also that during the execution with ASSG, $1,600$ tuples timed out (and where not recovered as the topology was configured disabling Storm fault tolerance mechanisms). This clearly shows how the shuffle grouping scheduling policy can have a large impact on the system performances.

Simple Application with Real Dataset — In this test we pretended to run a simple application on a real dataset: for each tweet of the twitter dataset mentioned in Section V-A we want to gather some statistics and decorate the outgoing tuples with some additional information. However the statistics and additional informations differ depending on the class the entities mentioned in each tweet belong. We assumed that this leads to a long execution time for *media* (e.g. possibly caused by an acces to an external DB to gather historical data), an average execution time for *politicians* and a fast execution time for *others* (e.g. possibly because these tweets are not decorated). We modeled execution times with 25ms, 5 ms and 1ms of busy waiting respectively.

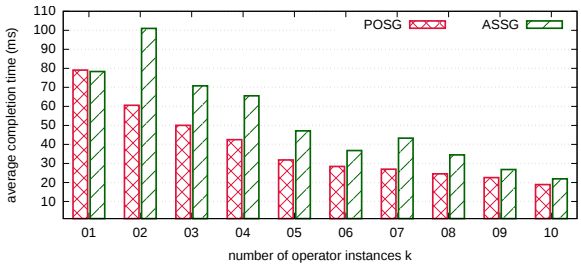

Fig. 12. Prototype average per tuple completion time $\overline{L}$ as a function of the number of operators k.

Figure 12 shows the average completion time $\overline{L}$ for both POSG and ASSG as a function of the number of instances k. For all values of k (except $k = 1$), POSG provides lower average completion times than Round-Robin, with a mean speed up S_L of 1.37. Increasing the value of k decreases the gap between POSG and Round-Robin. However with $k = 10$ POSG still improves the average completion time by 16%. For $k = 2$ and $k = 7$, we can notice an unanticipated behaviour of Round-Robin: adding one more instance increases the completion times (in particular, with $k = 2$ there where 221 timeouted tuples). On the other hand, POSG average completion time always decreases with growing values of k. Notice also that, to provide this improvement, POSG exchanged only 916 additional messages (with respect to a stream of size $m = 500,000$).

VI. Related Work

Load balancing in distributed computing is a well known problem that has been extensively studied since the 80s [3], [12]. Distributed stream processing systems have been designed, since the beginning, by taking into account the fact that load balancing is a critical issue to attain the best performance.

Hirzel *et al.* [8] recently provided an extensive overview of possible optimization strategies for stream processing systems, including load balancing. They identify two ways to perform load balancing in stream processing systems: either when placing the operators on the available machines or when assigning load to operator instances. In this latter case, the load balancing mechanism can be either pull based, *i.e.,* it is the consumers responsibility to acquire the load from the producers, or push based, *i.e.,* the converse.

In the the last year there has been new interest on improving load balancing with *key grouping* [6], [10]. However, key grouping imposes some strict limitations for assigning tuples to operator instances; as such, solutions available for key grouping would underperform if applied with shuffle grouping.

Considering *shuffle grouping*, Arapaci *et al.* [2] as well as Amini *et al.* [1], among other contributions, provide solutions to maximise the system efficiency when the operator instances execution times are non uniform, either because the hardware is etherogeneous or because each instance carries out different computations. On the other hand, at the best of our knowledge, there is no prior work directly addressing load balancing with *shuffle grouping* on non uniform operator instances considering that the tuples execution time depend on the tuple themselves.

VII. Conclusions

In this paper we introduced Proactive Online Shuffle Grouping, a novel approach to shuffle grouping aimed at reducing the overall tuple completion time by scheduling tuples on operator instances on the basis of their estimated execution time. POSG makes use of sketch data structures to keep track of tuple execution time on operator instances in a compact and scalable way. This information is then fed to a greedy scheduling algorithm to assign incoming load.

The analysis of POSG backs up the results of the experimental evaluation, proving that the Greedy Online Scheduler algorithm is a $(2 - 1/k)$-approximation of the optimal one as well as providing bounds and insights on the accuracy of the estimation of the execution time w_t. Furthermore, we extensively tested POSG performance both through simulations and with a prototype implementation integrated within the Apache Storm framework. The results showed how POSG provides important speedups in tuple completion time when the workload is characterized by skewed distributions. Further research will be needed to explore how much the load model affect performance.

For example it would be interesting to include other metrics in the load model, e.g. network latencies, to check how much these may improve the overal performance.

REFERENCES

[1] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS, 2006.

[2] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: Making the fast case common. In *Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems*, IOPADS, 1999.

[3] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2), 2002.

[4] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18, 1979.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55, 2005.

[6] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4), 2014.

[7] D. Gusfield. Bound the naive multiple machine scheduling with release times deadlines. *Journal of Algorithms*, (5):1–6, 1984.

[8] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4), 2014.

[9] Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc., 2005.

[10] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS, 2015.

[11] The Apache Software Foundation. Apache Storm. http://storm.apache.org.

[12] S. Zhou. *Performance Studies of Dynamic Load Balancing in Distributed Systems*. PhD thesis, UC Berkeley, 1987.