Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems

Research Paper

Nicoló Rivetti
LINA / Université de Nantes, France
DIAG / Sapienza University of Rome, Italy
rivetti@dis.uniroma1.it

Emmanuelle Anceaume
IRISA / CNRS
Rennes, France
emmanuelle.anceaume@irisa.fr

Yann Busnel
Crest (Ensai) / Inria
Rennes, France
yann.busnel@ensai.fr

Leonardo Querzoni
DIAG / Sapienza University of Rome, Italy
querzoni@dis.uniroma1.it

Bruno Sericola
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 almost the same for all the tuples. However, such an 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 Online Shuffle Grouping (OSG), a novel approach to shuffle grouping aimed at reducing the overall tuple completion time. OSG 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.

1. INTRODUCTION

Stream processing systems are today gaining momentum as a tool to perform analytics on continuous data streams. Their ability to produce 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 tuple assignment is constrained). Such grouping functions are often called *shuffle grouping* and represent a fundamental element of a large number of stream processing applications.

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 [16] leverage a simple Round-Robin scheduling 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.

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 en-

Middleware '16 December 12–16, 2016, Trento, Italy

DOI: 10.1145/1235

riches 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.

On the basis of the previous observation the tuple scheduling strategies for shuffle grouping must be re-thought: tuples must be scheduled with the aim of balancing the overall processing time on operators in order to reduce the average tuple execution time. However, tuple processing times are not known in the scheduling phase.

To the best of our knowledge this is the first paper introducing a solution for this problem. In particular, here we introduce *Online Shuffle Grouping* (OSG) a novel approach to shuffle grouping that aims at reducing tuple completion times by carefully scheduling each incoming tuple. Scheduling in OSG is performed by applying a join-shortest-queue policy [7] on queues whose size is defined as the execution time of all the elements of the queue. However, making such idea works in practice in a streaming setting is not trivial. In particular, OSG makes use of sketches to efficiently keep track of tuple execution times at the available operator instances 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, OSG 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 similar, but rather depend on the tuple content.

In summary, we provide the following contributions:

- We introduce OSG the first solution for shuffle grouping that explicitly addresses the problem of parallel operator (non-uniform) instances imbalance under loads characterized by non-uniform tuple execution times; OSG 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 OSG's sensibility to both the load characteristic and its configuration parameters with an extensive simulation-based evaluation that points the scenarios where OSG is expected to provide its best performance;
- We evaluate OSG's performance by integrating a prototype implementation with the Apache Storm stream processing framework on Microsoft Azure platform and running it against a real Twitter trace and a synthetic trace generated with the LDBC Social Network Benchmark [10].

After this introduction, the paper starts by defining a system model and stating the problem we intend to address (Section 2); it then introduces OSG (Section 3) and shows the results of our probabilistic analysis (Section 4); results from our experimental campaign are reported in Section 5 and are followed by a discussion of the related works (Section 6); finally, Section 7 concludes the paper.

2. 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 directed 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 $O_1, \cdots, O_k$ of it and by partitioning its inbound stream O^{in} in k sub-streams $O_1^{in}, \cdots, O_k^{in}$. Each operator instance has a FIFO input queue where tuples are buffered while the instance is busy processing previous tuples. 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 a sequence of tuples whose size (that is the number of tuples) m is unknown. 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[1], we consider a topology with an operator S (*scheduler*) which schedules the tuples of a DS O^{in} consumed by the instances $O_1, \cdots, O_k$ of operator O.

We denote by $w_{t,op}$ the execution time of tuple t on operator instance O_{op} (in the following we will use O_{op} to indicate a generic operator instance). The execution time $w_{t,op}$ is modelled as an unknown function[2] of the content of tuple t and that may be different for each operator instance (*i.e.,* we do not assume that the operator instances are uniform). We simplify the model assuming that $w_{t,op}$ depends on a single fixed and known attribute value of t. The probability distribution of such attribute values, as well as $w_{t,op}$ 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. Abusing the notation, we may omit in $w_{t,op}$ the operator instance identifier subscript.

Let $\ell(t)$ be the completion time of the t-th tuple of the stream, *i.e.,* the time it takes for the t-th tuple from the

[1]The case where operator S is parallelized is discussed in Section 4.1.

[2]In the experimental evaluation we relax the model by taking into account the execution time variance

instant it is inserted in the buffering queue of its operator instance O_{op} up to the instant is has been processed by O_{op}. Then we can define the average completion time as

$$\overline{L} = \frac{1}{m} \sum_{j=1}^{m} \ell(j).$$

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 substreams assignment strategy may introduce additional queuing delays when the execution time of input tuples is not similar. 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 respective 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 $\ell(a_0) + \ell(b_1) + \ell(a_2) = 29$s, where $\ell(a_0) = 10$s, $\ell(b_1) = 1$s and $\ell(a_2) = (8+10)$s, and $\overline{L} = 9.66$s. Note the wasted queuing delay of 8s for tuple a_2. A better schedule would be to assign a_0 to instance 1, while b_1 and a_2 to instance 2, giving a cumulated completion time equals to $10 + 1 + 10 = 21$s (*i.e.,* no queuing delay), and $\overline{L} = 7$s.

3. Online Shuffle Grouping

Online Shuffle Grouping is a shuffle grouping implementation based on a simple, yet effective idea: if we assume to know the execution time $w_{t,op}$ of each tuple t on any of the operator instances, 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 $w_{t,op}$ is generally unknown. A common solution to this problem is to build a cost model for the tuple execution time 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.

To overcome all these issues, OSG takes decisions based on the estimation $\widehat{\mathcal{C}}_{op}$ of the execution time assigned to instance O_{op}, that is $\mathcal{C}_{op} = \sum_{t \in O_{op}^{in}} w_{t,op}$. In order to do so, OSG computes an estimation $\widehat{w}_{t,op}$ of the execution time $w_{t,op}$ of each tuple t on each operator instance O_{op}. Then, OSG can also compute the sum of the estimated execution times of the tuples assigned to an instance O_{op}, *i.e.,* $\widehat{\mathcal{C}}_{op} = \sum_{t \in O_{op}^{in}} \widehat{w}_{t,op}$, which in turn is the estimation of $\mathcal{C}_{op}$. A greedy scheduling algorithm (Section 3.1) is then fed with estimations for all the available operator instances.

To implement 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 processes. 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 then be able to (again) correctly estimate the tuples execution times. This solution does not require any *a priori* knowledge on the stream composition or the 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 *reactive* solution can hardly be applied to this problem, in fact it would schedule input tuples on the basis of a previous, possibly stale, load state of the operator instances. In addition, reactive scheduling typically imposes a periodic overhead even if the load distribution imposed by input tuples does not change over time.

For the sake of clarity, we consider a topology with a single operator S (*i.e.,* a *scheduler*) which schedules the tuples of a DS O^{in} consumed by the k instances of operator O (*cf.,* Figure 1). To encompass topologies where the operator generating DS O^{in} is itself parallelized, we can easily extend the model by taking into account parallel instances of the scheduler S. More precisely, there are s schedulers $S_1, \cdots, S_s$, where scheduler S_i schedules the tuples belonging to the sub-stream $O_{i,1}^{in}, \cdots, O_{i,k}^{in}$. We show (*cf.,* Section 4.1) that also in this setting OSG performances are better than Round-Robin scheduling policy. In other words OSG can be deployed when the operator S is parallelized. Notice that our approach is hop-by-hop, *i.e.,* we consider a single shuffle grouped edge in the topology at a time. However, OSG can be applied to any shuffle grouped stage of the topology.

3.1 Background

Data Streaming model — We present the data stream model [12], 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 $[n] = \{1, \ldots, n\}$, with m the unknown size (or length) of the stream. We denote by p_t the unknown probability of occurrence of item t in the stream and by f_t the unknown frequency[3] 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 $h : [n] \to [c]$ is said to be 2-universal if for every two different items $x, y \in [n]$, for all $h \in \mathcal{H}$, $\mathbb{P}\{h(x) = h(y)\} \leq 1/c$, which is the probability of collision obtained if the hash function assigned truly random values 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 $r = \lceil \log(1/\delta) \rceil$ and $c = \lceil e/\varepsilon \rceil$, with $e \simeq 2.71828$. Each row is associated with a different 2-universal hash function $h_i : [n] \to [c]$. When the `Count Min` algorithm reads item t from the input stream, it updates each row: $\forall i \in$

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

Listing 3.1: 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 r 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
```

$[r], \mathcal{F}[i, h_i(t)] \leftarrow \mathcal{F}[i, h_i(t)] + 1$. 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 $\mathcal{O}(\log[(\log m + \log n)/\delta]/\varepsilon)$ bits, while update and query time complexities are $\mathcal{O}(\log(1/\delta))$. 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 tuples on non-uniform operator instances in order to minimize the average per tuple 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 tuples to the less loaded available operator instance. In Section 4.1 we prove that this algorithm closely approximates an optimal omniscient scheduling algorithm, that is an algorithm that knows in advance all the tuples it will received. Notice that this is a variant of the join-shortest-queue (JSQ) policy [11, 7], where we measure the queue length as the time needed to execute all the buffered tuples, instead of the number of buffered tuples.

3.2 OSG design

Each operator instance *op* maintains two `Count Min` sketch matrices (Figure 1.A): the first one, denoted by $\mathcal{F}_{op}$, tracks the tuple frequencies $f_{t,op}$; the second, denoted by $\mathcal{W}_{op}$, tracks the tuples cumulated execution times $W_{t,op} = w_{t,op} \times f_{t,op}$. Both `Count Min` matrices have the same sizes and hash functions. The latter is the generalized version of the `Count Min` presented in Section 3.1 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 3.1) both matrices after each tuple execution.

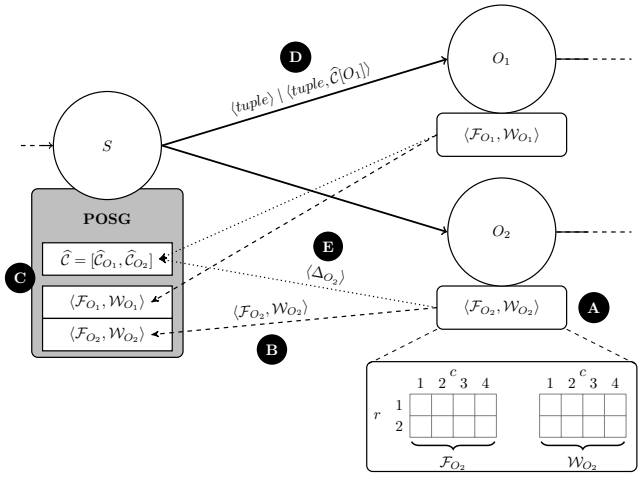

Figure 1: Online Shuffle Grouping design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$) and $k = 2$.

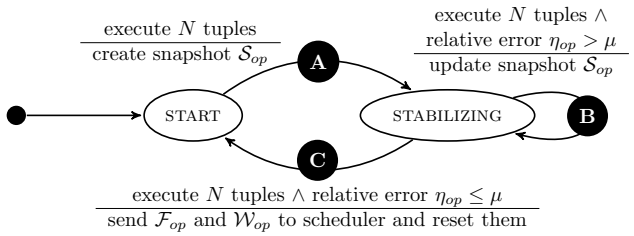

Figure 2: Operator instance finite state machine.

The operator instances are modelled as a finite state machine (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 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 $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices are stable when the relative error η_{op} between the previous snapshot and the current one is smaller than μ, that is if

$$\eta_{op} = \frac{\sum_{i=1}^{r}\sum_{j=1}^{c}\left|\mathcal{S}_{op}[i,j] - \frac{\mathcal{W}_{op}[i,j]}{\mathcal{F}_{op}[i,j]}\right|}{\sum_{i=1}^{r}\sum_{j=1}^{c}\mathcal{S}_{op}[i,j]} \leq \mu \qquad (1)$$

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 a synchronization mechanism that springs 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 opera-

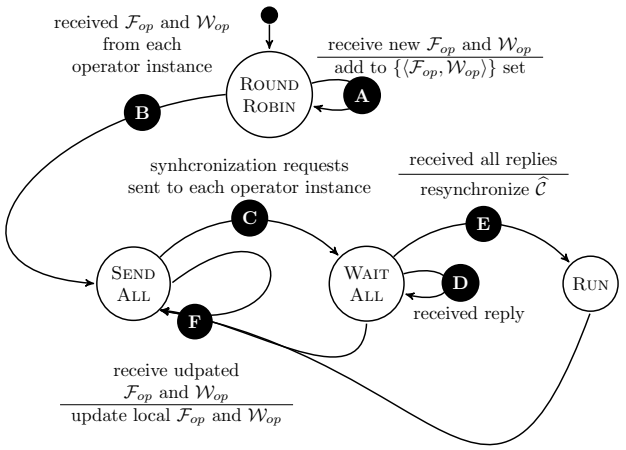

Figure 3: Scheduler finite state machine.

tor 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.

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. When in SEND ALL state, the scheduler sends 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 3.2). 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* 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 3.2). 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 3.2). Then it

Listing 3.2: 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
```

increments the target instance estimated cumulated execution time with the estimated tuple execution time (UPDATE $\widehat{\mathcal{C}}$ function, Listing 3.2). 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 (OSG TIME COMPLEXITY).
For each tuple read from the input stream, the time complexity of OSG for each instance is $\mathcal{O}(\log(1/\delta))$. *For each tuple submitted to the scheduler, OSG time complexity is* $\mathcal{O}(k + \log(1/\delta))$.

PROOF. By Listing 3.1, 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 3.2, for each submitted tuple, the scheduler has to retrieve the index with the smallest 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 $\mathcal{O}(k + \log(1/\delta))$. □

THEOREM 3.2 (OSG SPACE COMPLEXITY).
The space complexity of OSG for the operator instances is $\mathcal{O}(\log[(\log m + \log n)/\delta]/\varepsilon)$, *while the space complexity for the scheduler is* $\mathcal{O}((k \log[(\log m + \log n)/\delta])/\varepsilon)$.

PROOF. Each operator instance stores two matrices, each one requiring $\log(1/\delta) \times (e/\varepsilon) \times \log m$ bits. In addition, it also stores a hash function whose domain size is n. Then the space complexity of OSG on each operator instance is $\mathcal{O}(\log[(\log m + \log n)/\delta]/\varepsilon)$. The scheduler stores the same matrices, one for each instance, as well as a vector of size k. Then the space complexity of OSG on the scheduler is $\mathcal{O}((k \log[(\log m + \log n)/\delta])/\varepsilon)$. □

THEOREM 3.3 (OSG COMMUNICATION COMPLEXITY).
The communication complexity of OSG is of $\mathcal{O}((km)/N)$ *messages and* $\mathcal{O}(m(\log[(\log m + \log n)/\delta]/\varepsilon + k \log m)/N)$ *bits.*

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 $\mathcal{O}(m/(kN))$ messages and $\mathcal{O}(m \log[(\log m + \log n)/\delta]/(kN\varepsilon))$ bits. When the scheduler receives these matrices, the synchronization mechanism springs 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 $\mathcal{O}(m/N)$ messages and $\mathcal{O}((m \log m)/N)$ bits. Since there are

k instances, the communication complexity is $\mathcal{O}((km)/N)$ messages and $\mathcal{O}(m(\log[(\log m + \log n)/\delta]/\varepsilon + k \log m)/N)$ 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 $k \leq 20$).

4. THEORETICAL ANALYSIS

This section provide the analysis of the quality of the scheduling performed by OSG in two steps. First we study the Greedy Online Scheduler algorithm approximation in Section 4.1. Then, in Section 4.2 we provide a probabilistic analysis of the mechanism that OSG uses to estimate the tuple execution times.

4.1 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 is known for each tuple t. 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 $\sigma = \langle 1, \ldots, m \rangle$ 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 3.1 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 O_{op} be the instance on which the last tuple t is executed. By construction of the algorithm, when tuple t starts its execution on instance O_{op}, all the other instances are busy, otherwise t would have been executed on another instance. Thus when tuple t starts its execution on instance O_{op}, each of the k instances must have been allocated a load at least equivalent to $(\sum_{\ell=1}^{m} w_\ell - w_t)/k$. Thus we have,

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

Now, it is easy to see that

$$C^{\sigma}_{OPT} \geq \frac{\sum_{\ell=1}^{m} w_\ell}{k}, \quad (3)$$

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

$$C^{\sigma}_{OPT} \geq \max_{\ell} w_\ell. \quad (4)$$

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

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

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 [8].

Consider the example of $k(k-1)$ tuples with all the same processing time equal to w/k and one tuple with a processing time equal to w. Suppose that the $k(k-1)$ tuples are scheduled first and then the longest one. Then the greedy algorithm will exhibit a makespan equal to $w(k-1)/k + w = w(2-1/k)$ while the OPT scheduling will lead to a makespan equal to w.

If we now consider a parallelized scheduler, it is easy to see that in the worst case the degradation of the makespan obtained with Greedy Online Scheduler algorithm is linear in the parallelization degree. Indeed, in absence of any additional communication between these schedulers, each of them cannot do better than providing a schedule in isolation with respect to the other schedulers. The same degradation applies to Round-Robin scheduling. Consequently our approach efficiently handles a moderate parallelization degree

4.2 Execution Time Estimation

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

From the count Min algorithm, and for any $v \in [n]$, 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 by $\min(w)$ and $\max(w)$ the respectively minimum and maximum execution times of the tuples. We have trivially

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

In the following we respectively write 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 whose elements are the subsets of $\{1, \ldots, n\} \setminus \{v\}$ 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, we have $W_v/C_v \in R(v)$.

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

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

where $X(v, A)$ is the fraction:

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

Note that we have $\sum_{x \in R(v)} 1_{\{x = X(v,A)\}} = 1$, thus

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

THEOREM 4.3. *When all the f_u are equal, we have*

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

where $S = \sum_{i=1}^n w_i$.

PROOF. Since all the f_u are equal, we have $f_u = m/n$. The proof then proceeds by replacing this value of f_u in the above expression of $\mathbb{E}\{W_v/C_v\}$. □

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

5. EXPERIMENTAL EVALUATION

In this section we evaluate the performance obtained by using OSG 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 5.2) and with a prototype of OSG targeting Apache Storm (Section 5.3).

5.1 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 $m = 100,000$ 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 $n(w)$ as the number of distinct execution time values that the tuples can have. These $n(w)$ values are selected at *constant* distance in the interval $[\min_w, \max_w]$. We have also run tests generating the execution time values in the interval $[\min_w, \max_w]$ 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 $n(w) = 64$, $w_{min} = 1$ ms and $\max_w = 64$ 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 $n(w)$ execution time values and the n distinct items: for each of the $n(w)$ execution time values we pick *uniformly* at random $n/n(w)$ 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 build these associations using other distributions, namely geometric and binomial, without noticing unpredictable differences with respect to the results reported in this section. Finally, we have run each stream using 50 different seeds for the hash function generation, yielding a total of $5,000$ executions.

For the two use case we provide in this experimental evaluation, we use two different datasets: MENTION and REACH. The former (MENTION) is 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* (*i.e.,* Twitter users) mentioned in the tweet. 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.
The second dataset (REACH) is generated using the LDBC Social Network Benchmark [10]. Using the default parameters of this benchmark, we obtained a followers graph of $9,960$ nodes and $183,005$ edges (the maximum out degree was 734) as well as a stream of $2,114,269$ tweets where the most frequent author has an empirical probability of occurrence equal to 0.0038.

Algorithms — We present the results for 3 algorithms: Round-Robin, OSG and Full Knowledge. The former is the implementation of the Round-Robin policy and OSG is the implementation of our solution. Full Knowledge instead is an variant of OSG where the estimation is exact, *i.e.,* the scheduling algorithm is fed with the exact execution times for each tuple.

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

(**ii**) the average per tuple completion time speed up Λ^{alg} (simply *speed up* in the following) achieved by OSG or Full Knowledge with respect to Round-Robin

(**iii**) the throughput of the system expressed as tuples processed per second.

Recall that $\ell^{alg}(t)$ is the completion time of the t-th tuple of the stream when using the scheduling algorithm alg. To take into account any overhead introduced by the tested algorithm, we redefine the completion time $\ell^{alg}(t)$ as the time it takes from the injection of the t-th tuple at the source until it has been processed by the operator. Then we can define the average completion time $\overline{L}^{alg}$ and speed up Λ^{alg} as follows:

$$\overline{L}^{alg} = \frac{\sum_{t=1}^{m} \ell^{alg}(t)}{m} \text{ and } \Lambda^{alg} = \frac{\sum_{t=1}^{m} \ell^{\text{Round-Robin}}(t)}{\sum_{t=1}^{m} \ell^{\text{alg}}(t)}$$

Whenever applicable we provide the maximum, mean and minimum figures over the $5,000$ executions.

5.2 Simulation Results

Here we report the results obtained by simulating the behavior of the considered algorithm on an ad-hoc multi-threaded simulator. The simulator streams the input dataset through the scheduler that enqueues it on the available target instances. Each target instance dequeues as-soon-as-possible the first element in its queue and simulates processing through busy-waiting for the corresponding execution time. With this implementation the simulated processing

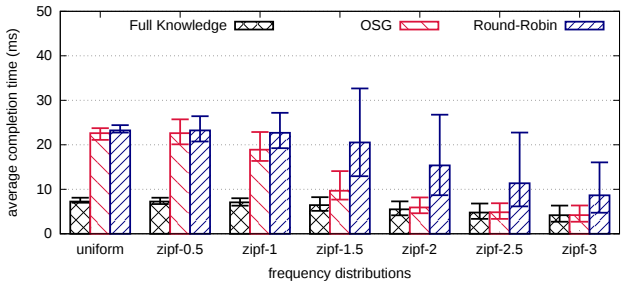

Figure 4: Average per tuple completion time $\overline{L}^{alg}$ with different frequency probability distributions

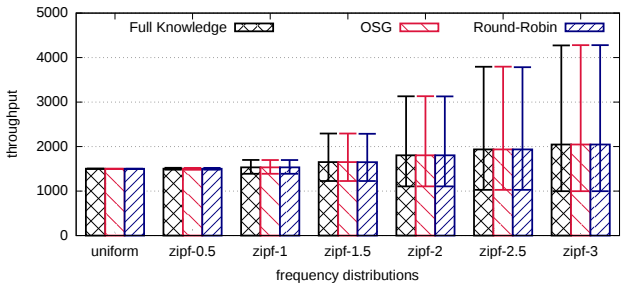

Figure 5: Throughput with different frequency probability distributions

times are characterized by only a slight variance mainly due to the precision in estimating the busy-waiting time, which depends on the operating system clock precision and scheduling.

Frequency Probability Distribution — Figure 4 shows the average completion time $\bar{L}^{alg}$ for OSG, 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 OSG and Round-Robin. With all distributions the Full Knowledge algorithm outperforms OSG 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 OSG is limited (in average 4%). Starting with Zipf-1.0 the gain is much more sizeable (20%) and with Zipf-1.5 we have that the maximum average completion time of OSG is almost smaller than the minimum average completion time of Round-Robin. Finally, with Zipf-2 OSG matches the performance of Full Knowledge. This behavior for OSG stems from the ability of its sketch data structures ($\mathcal{F}$ and $\mathcal{W}$ matrices, see Section 3) to capture more useful information for skewed input distributions.

Figure 5 shows the throughput for the same configurations. Clearly, all the algorithm achieve the same throughput. Each scheduling achieves different average completion

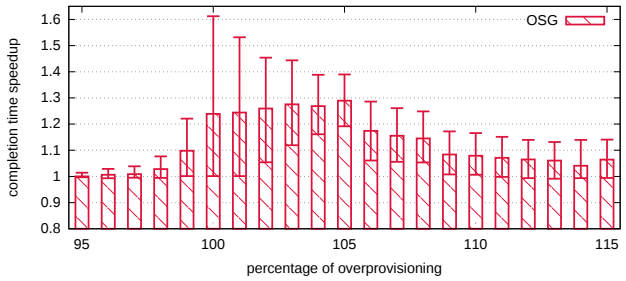

Figure 6: Speed up Λ^{OSG} as a function of the percentage of over-provisioning

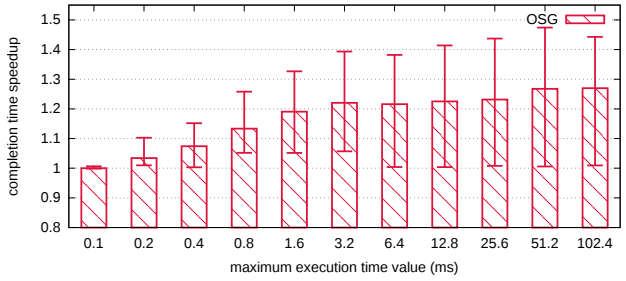

Figure 7: Speed up Λ^{OSG} as a function of the maximum execution time value w_{max}.

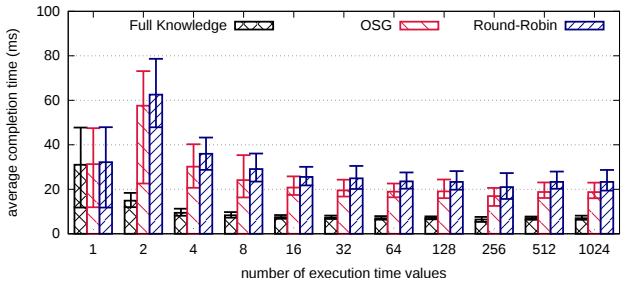

Figure 8: Average per tuple completion time $\bar{L}^{alg}$ as a function of the number of execution time values $n(w)$.

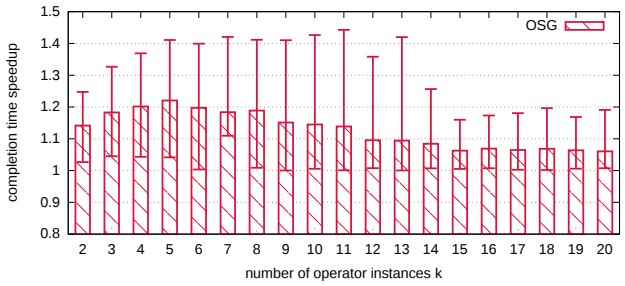

Figure 9: Speed up Λ^{OSG} as a function of the number of operator instances k.

times $\overline{L}^{alg}$ since they affect the queuing time experienced by the tuples. However the throughput results shows that for all three algorithm the total load imposed on each operator instance is balanced. Since this is a quite consistent behavior for all simulation, we omitted the remaining throughput plots.

Input Throughput — Figure 6 shows the speed up Λ^{OSG} 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 OSG. Conversely, when the system is oversized (109% 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 108%), our algorithm introduces a noticeable speed up Λ^{OSG}, in average at least 1.14 with a peak of 1.29 at 105%. Finally, even when the system is largely oversized (115%), we still provide an average speed up of 1.06. We omitted Full Knowledge to improve the readability of the plot. The general trend is the same of OSG, achieving the same speed up when the system is undersized. When the systems is oversized, Full Knowledge hits a peak of 3.6 at 101% and provides a speed up of 1.57 at 115%. In all configurations, all algorithm achieve roughly the same output throughput. In particular it is maximum when the system is undersized (95% to 99%), and decreases accordingly with the percentage of over-provisioning when the system is oversized (100% to 115%)

Maximum Execution Time Value — Figure 7 shows the speed up Λ^{OSG} as a function of the maximum execution time $\max_w$. With $\max_w = 0.1$ms, all the tuples have the same execution time $w_t = 0.1$ms, thus all algorithm achieve the same result. Increasing the value of $\max_w$ increases the gap between the 64 possible execution times, allowing more room to improve the scheduling, then OSG speed up Λ^{OSG} grows for $\max_w \geq 0.2$. However notice that OSG seems to hit an asymptote at 1.25 for $\max_w \geq 102.4$. We omitted Full Knowledge to improve the readability of the plot. The general trend is the same of OSG, however starting with $\max_w \geq 0.4$ Full Knowledge achieves a larger speed up and hits an asymptote at 3.4 for $\max_w \geq 102.4$.

Number of Execution Time Values — Figure 8 shows the average completion time $\bar{L}^{alg}$ for OSG, Round-Robin and Full Knowledge as a function of the number of execution time values $n(w)$. We can notice that for growing values of $n(w)$ both the average completion time values and variance decrease, with only slight changes for $n(w) \geq 16$. Recall that $n(w)$ is the number of completion time values in the interval $[\min_w, \max_w]$ that we assign to the n distinct attribute values. For instance, with $n(w) = 2$, all the tuples have a completion time equal to either 0.1 or 6.4 ms. Then, assigning either of the two values to the most frequent item strongly affects the average completion time. Increasing $n(w)$ reduces the impact that each single execution time has on the average completion time, leading to more stable results. The gain between the maximum, mean and maximum average completion times of OSG and Round-Robin (in average 19%) is mostly unaffected by the value of $n(w)$.

Number of operator instances k — Figure 9 shows the speed up Λ^{OSG} as a function of the number of parallel operator instances k. From $k = 2$ to $k = 5$ the speed up Λ^{OSG} grows, starting with an average value of 1.14 and reaching an

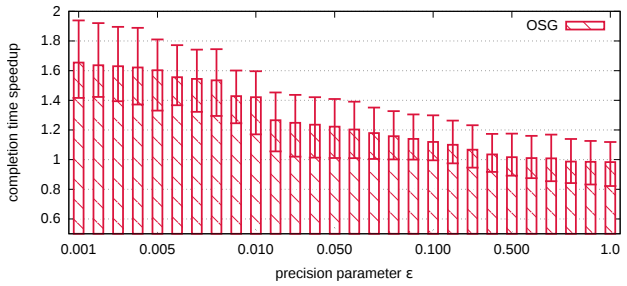

Figure 10: Speed up Λ^{OSG} as a function of the precision parameter ε (*i.e.*, number of columns $c = \lceil e/\varepsilon \rceil$).

average peak of 1.23. Then the speed up Λ^{OSG} decreases to 1.09 ($k = 12$) and reaches what seems to be an asymptote at 1.06 ($k = 20$). In other words, for moderate values of k (*i.e.,* $k \leq 12$), OSG introduces a sizeable improvement in the average completion latency with respect to Round-Robin. On the other hand, for large value of k (*i.e.,* $k > 12$), the impact of OSG is mitigated. As the number of available instances increases, Round-Robin is able to better balance the load, thus limiting OSG effectiveness. We omitted Full Knowledge to improve the readability of the plot. The general trend is the same of OSG, hitting a peak of 4.0 at $k = 11$ and the decreasing toward an asymptote at 2.7 for $k = 20$.

Precision parameter ε — Figure 10 shows the speed up Λ^{OSG} 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 OSG 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 OSG performance: in average a 10 time decrease in ε (thus a 10 time increase in memory) results in a 25% increase in the speed up. Large values of ε do not provide good performance; however, starting with $\varepsilon \leq 0.09$ the minimum average completion time speed up is always larger than 1.

Time Series — Figure 11 shows the completion time as the stream unfolds (the x axis is the number of tuples read from the stream) for both OSG and Round-Robin for a single execution. 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).

With the prototype we could not achieve the same measurement precision as with the simulator. Then, to be able to compare the simulator and prototype time series, we increased by a factor of 10 the execution times.

In the leftmost part of the plot, we can see OSG and Round-Robin provide the same exact results up to $m = 10,690$, where OSG starts to diverge by decreasing the completion time as well as the completion time variance. This behaviour is the result of OSG 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, also reducing the completion time variance.

At $m = 75,000$ we inject the load characteristic change described above. Immediately OSG performance degrades as the content of $\mathcal{F}$ and $\mathcal{W}$ matrices is outdated. At $m = 84,912$ the scheduler receives the updated $\mathcal{F}$ and $\mathcal{W}$ matrices and recovers. This demonstrates OSG ability to adapt at runtime with respect to changes in the load distributions.

5.3 Prototype

To evaluate the impact of OSG on real applications we implemented it as a custom grouping function within the Apache Storm [16] 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. This choice helped us ruling out from tests possible side-effects caused by co-sharing of CPU cores among processes that are not part of the Storm framework. The prototype was first tested with the synthetic dataset and application, as for the previous simulations, and then on two realistic stream processing applications.

Time Series — In this first test the topology was made of a source (*spout*) and operators (*bolts*) S and O. The source generates (reads) the synthetic (real) input stream. Bolt S uses either OSG or the Apache Storm standard shuffle

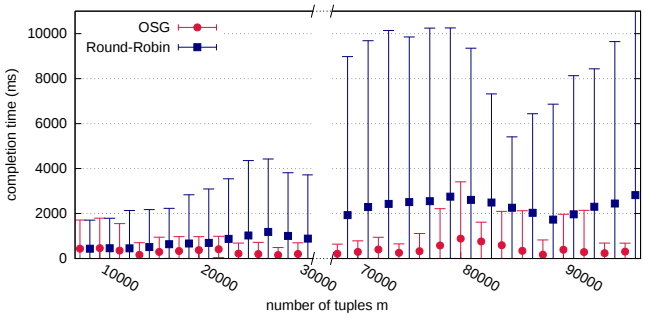

Figure 11: Simulator per tuple completion time time-series.

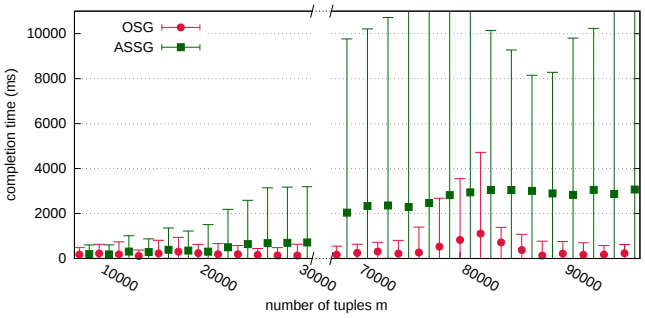

Figure 12: Prototype per tuple completion time time-series.

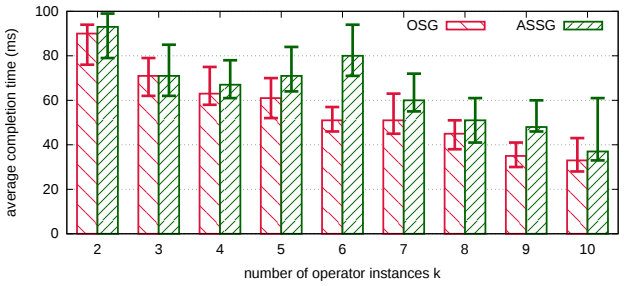

Figure 13: Prototype average per tuple completion time $\overline{L}^{alg}$ as a function of the number of operators k.

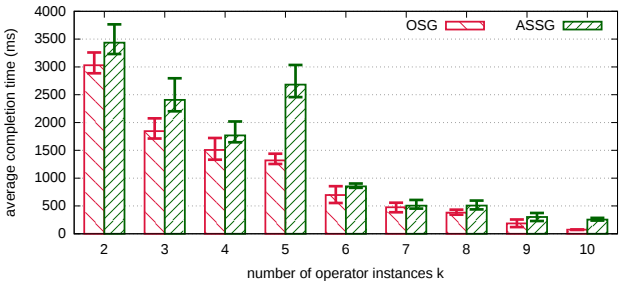

Figure 14: Prototype average per tuple completion time $\overline{L}^{alg}$ as a function of the number of operators k.

grouping implementation (ASSG in the following) to route the tuples toward the k instances (*tasks*) of bolt O.

Figure 12 provides results for the prototype with the same settings of the test whose results were reported in Figure 11. We can notice the same general behavior both in the simulator and in the prototype. In the right part of the plot OSG 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 $m = 75,000$ OSG 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.

Use Case Mentions — In this test we run a simple application using the MENTION (Section 5.1) dataset as input: for each tweet of the stream we want to extract mentions it contains and accordingly decorate the outgoing tuple with additional information on them. In particular, for each mention carried by a tweet the bolt performs a sets of queries (based on the mention itself) on an external database; The larger is the number of mentions contained in the tweet, the more processing time it will take for the bolt to complete the queries and decorate the outgoing tuple. The mention execution times belong to the interval $[0.1, 23]$ ms, each mention adds in average 1 ms to the processing time to execute the corresponding query. The number of mentions is only limited by the number of users registered on Twitter and summed up to a total of $35,000$ unique identities in the MENTION dataset. Globally, the tuple execution times belong to the interval $[1.8, 36]$ ms, while the average per tuple execution time is 7 ms.

Figure 13 shows the mean, maximum and minimum average completion time $\overline{L}^{alg}$ for both OSG and ASSG as a function of the number of instances k over 10 executions. For all values of k OSG provides lower or equal average completion times than ASSG, with a mean, minimum and maximum speed up Λ^{OSG} of 1.27, 1.0 and 2.16. For $k = 5$ and $k = 6$, we can notice an unanticipated behavior of ASSG: adding one more instance increases the completion times. On the other hand, OSG average completion time always decreases with growing values of k. Notice also that, to provide this improvement, OSG exchanged only a few hundred additional messages against a stream of size $m = 500,000$.

Use Case Reach — In this test we want to compute the *reach* of twitted terms using the REACH (Section 5.1) dataset as input. The reach of a term is the total number of estimated unique Twitter users to which were delivered tweets about the search term. Usually, this metric is calculated through a periodic batch process using the followers graph, where edges are enriched with re-tweet probabilities. We propose instead to compute this value in a streaming fashion, for each tweet, restricting the computation to a depth of 3 in the followers graph of $9,960$ nodes. Globally, the tuple execution times belong to the interval $[0.01, 70]$ ms, the most frequent tuple execution time is in average 65 ms, while the average per tuple execution time is 20 ms.

Figure 14 shows the mean, maximum and minimum average completion time $\overline{L}^{alg}$ for both OSG and ASSG as a function of the number of instances k over 10 executions. Except for the unanticipated spike of ASSG for $k = 5$, the completion latency decreases as k increases. For all k OSG has a smaller mean average completion latency than ASSG. In addition, for most values of k, the maximum average completion latency of OSG is smaller or equal to the minimum average completion latency of ASSG. Finally, the average speed up Λ^{OSG} of OSG with respect to ASSG is at least 1.05, at most 3.4 and in average 1.5. To achieve these results, OSG exchanges only a few thousand additional messages, against a stream size of $m = 2,114,269$

6. RELATED WORK

Load balancing in distributed computing is a well known problem that has been extensively studied since the 80s [17, 3]. 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.* [9] 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 last few years there has been new interest on improving load balancing with *key grouping* [6, 13, 14]. 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. In addition, the mentioned works assume that all tuples of a stream have the same execution time.

Sharaf *et al.* [15] propose a comprehensive solution to schedule multiple continuous queries minimizing the response time. Considering *shuffle grouping*, Arapaci *et al.* [2] as well as Amini *et al.* [1], among other contributions, provide solutions to maximize the system efficiency when the execution times of the operator instances are non-uniform, either because the hardware is heterogeneous or due to the fact that 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.

7. CONCLUSIONS

In this paper we have introduced Online Shuffle Grouping, a novel approach to shuffle grouping aiming at reducing the overall tuple completion time by scheduling tuples on operator instances on the basis of their estimated execution time. OSG 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 Greedy Online Scheduler algorithm to assign incoming load.

The analysis of OSG validates the results of the experimental evaluation, proving that the Greedy Online Scheduler algorithm is a $(2 - 1/k)$-approximation of the optimal one (the optimal scheduling algorithm has the full knowledge of all the set of tuples submitted to the system) as well as providing bounds and insights on the accuracy of the estimation of the execution time w_t. Furthermore, we have extensively tested OSG performance both through simulations and with a prototype implementation integrated within the Apache Storm framework. The results show how OSG 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 overall performance. In addition, this work may be applied to [14], lifting the assumption on the tuple execution time.

8. 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] V. Gupta, M. Harchol-balter, K. Sigman, and W. Whitt. Analysis of join-theshortest-queue routing for web server farms. In *Proceedings of the 25th IFIP WG 7.3 International Symposium on Computer Modeling, Measurement and Evaluation*, PERFORMANCE, 2007.

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

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

[10] Linked Data Benchmark Council. Social Network Benchmark. http://ldbcouncil.org/benchmarks/snb.

[11] A. Mukhopadhyay and R. Mazumdar. Analysis of randomized join-the-shortest-queue (jsq) schemes in large heterogeneous processor sharing systems. *IEEE Transactions on Control of Network Systems*, PP(99):1–1, 2015.

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

[13] M. A. U. Nasir, G. D. F. Morales, D. G. Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, ICDE, 2015.

[14] 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.

[15] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems*, 33(1):5:1–5:44, Mar. 2008.

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

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