A Holistic View of Stream Partitioning Costs

Nikos R. Katsipoulakis, Alexandros Labrinidis, Panos K. Chrysanthis
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
{katsip, labrinid, panos}@cs.pitt.edu

ABSTRACT

Stream processing has become the dominant processing model for monitoring and real-time analytics. Modern Parallel Stream Processing Engines (pSPEs) have made it feasible to increase the performance in both monitoring and analytical queries by parallelizing a query's execution and distributing the load on multiple workers. A determining factor for the performance of a pSPE is the partitioning algorithm used to disseminate tuples to workers. Until now, partitioning methods in pSPEs have been similar to the ones used in parallel databases and only recently *load-aware* algorithms have been employed to improve the effectiveness of parallel execution.

We identify and demonstrate the need to incorporate *aggregation* costs in the partitioning model when executing *stateful* operations in parallel, in order to minimize the overall latency and/or throughput. Towards this, we propose new stream partitioning algorithms, that consider both tuple *imbalance* and *aggregation* cost. We evaluate our proposed algorithms and show that they can achieve up to an order of magnitude better performance, compared to the current state of the art.

1. INTRODUCTION

Monitoring and real-time temporal analytic queries are being widely used in a variety of services, whose quality relies on successfully capturing topic drift or trend fluctuation. Examples of such services include high-frequency algorithmic stock trading, social network analysis, targeted advertising, and click stream analysis. In order to match the speeds of data production, stream processing is deemed as the most promising model. At a high level, it requires (a) one-pass over the data, (b) constant processing time, and (c) continuous execution. After many flavors of single-thread stream processing engines [2, 6, 7, 11], Parallel Stream Processing Engines (pSPEs) have been introduced as a solution for processing high-volume data streams, in both single-node multi-threaded (*scale-up*) [10, 24], and in multiple-node (*scale-out*) [1, 35, 27, 3, 37, 34, 25, 8, 30] environments.

pSPEs have dominated the streaming landscape because of their ability to scale processing capability by dividing load into parallel data-flows, handled by many workers. At their core, pSPEs are critically affected by the partitioning algorithm of the sub data-flows delegated to workers: *the more evenly the load is partitioned, the more scalable a pSPE is*. Therefore, partitioning is paramount, especially in *stateful* operations, which involve windows of computation, complex delivery semantics (i.e., exactly-once), and window synchronization [8].

The focal point of our work is on *stateful* operations, which require the collocation of tuples with similar characteristics and produce an aggregated result, for each user-defined logical window. A window contains tuples based on count, time, or even session. The need for strict delivery semantics (e.g., exactly-once) imposes additional overheads for guaranteeing correctness and timely delivery of results, which are generated by checkpointing, out-of-order window alignment, window barriers, etc.

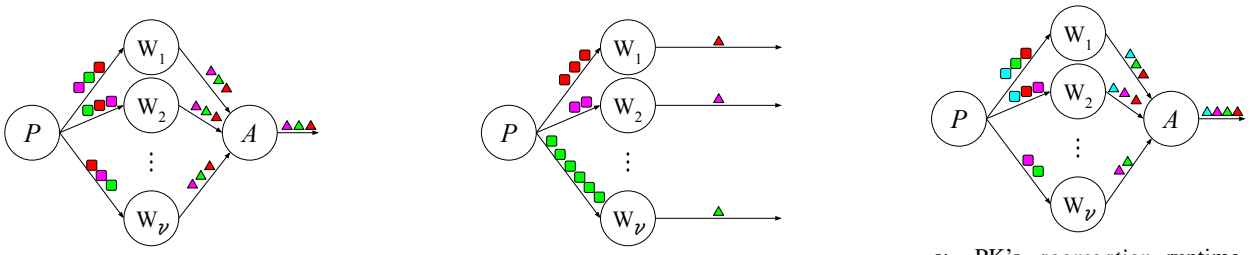
Often, pSPEs rely on dynamic *re-partitioning* of tuples to achieve better load balance [32, 38, 17, 13, 18, 35, 9, 15]. However, re-partitioning comes with the additional burden of state migration, which is a heavyweight task and involves complex synchronization protocols, state integration policies (subject to window semantics), and can potentially lead to delayed tuple delivery. Therefore, in our study we chose to take a step back and focus on the partitioning algorithm to make it more efficient, so that the need for re-partitioning materializes less often. Of course, re-partitioning solutions, such as Flux [32], require partitioning [1], and can complement our work to enhance a pSPE's performance further.

Until recently, pSPEs adopted partition algorithms used in Massively Parallel Processing Database Management Systems. The most popular algorithms among them are "shuffle" (or round-robin) (SH), and "field" (or hash) (FLD) algorithms [8, 34]. The former, blindly sends tuples to workers in a circular fashion akin to shuffling a deck of cards; whereas, the latter exploits a random process, usually a hash function, to disperse tuples to workers. Each partition algorithm has its merits and its drawbacks: SH manages to balance load evenly, but forces a computationally heavy aggregation step (Fig. 1a); FLD underperforms on skewed streams (i.e., when some keys appear more often than others) but does not require an aggregation step (Fig. 1b).

The state of the art partition algorithm is "partial key" grouping (PK) [28]. It focuses on improving performance by keeping track of the number of tuples sent to each worker in an online fashion. PK leverages the idea of *key splitting* [5], which dictates that tuples with the same attribute(s) can be split among two workers for the benefit of overall performance (Fig. 1c). Recently, an extension of PK that uses more than two choices was proposed [29], and was shown to further balance load among workers. The decision about

[1] Flux uses FLD as its partition algorithm.

Proceedings of the VLDB Endowment, Vol. 10, No. 11



a: SH's *aggregation* runtime is proportional to $\mathcal{V}$ times the number of distinct groups.

b: FLD doesn't require *aggregation*, but fails to balance load under skewed input.

c: PK's *aggregation* runtime is proportional to M times the number of distinct groups (M is the number of choices).

Figure 1: Existing stream partitioning algorithms lack a unified model that limits *imbalance* while keeping *aggregation* cost low.

which worker will receive a tuple is determined by the total number of tuples already sent to each one of them at the time of the decision. This way, the merits of FLD and SH are combined by overcoming skewness through the use of multiple choices and, at the same time, reducing aggregation cost.

Partition algorithms like PK (and its multi-choice variant [29]) focus on the aspect of *imbalance*, in terms of tuples sent to each worker on the parallel step of a *stateful* operation. Nevertheless, every *stateful* operation requires a step in which partial results are combined (Figs. 1a and 1c). In our work, we argue that an important factor for performance is the *aggregation* cost required to produce the final result, which is not considered by any other partitioning algorithm. In fact, to the extent of our knowledge, no other stream partitioning algorithm incorporates both *imbalance* and *aggregation* cost.

In this paper we propose that tracking the *aggregation* cost of a *stateful* operation reduces to counting the number of distinct keys sent to each worker on every window. Hence, we introduce a new class of partitioning algorithms, which leverage such information during the decision process. Our **contributions** are:

- Introduce a novel cost model for stream partitioning that considers both load *imbalance* and *aggregation* cost on every window of a *stateful* operation.
- Propose novel stream partitioning algorithms that incorporate our cost model to improve performance.
- Demonstrate the benefits of our cost model in real world benchmarks and present an empirical rule for choosing a partitioning algorithm for a *stateful* query.

Section 2 presents our model and existing partitioning algorithms. Section 3 shows mechanisms for keeping track of cardinality, and Section 4 presents our proposed algorithms. Section 5 and 6 demonstrate the details of our experiments, followed by Section 7, which offers a discussion on picking a partition algorithm. Finally, Section 8 presents related work, and our work concludes in Section 9.

2. PROPOSED MODEL

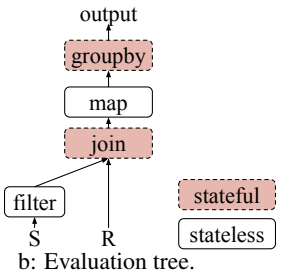
We focus on pSPEs for either *scale-up* or *scale-out* architectures. A *scale-up* architecture is a single multi-core machine, in which multiple cores are used to accommodate concurrent threads. A *scale-out* architecture is a multi-node environment in which a cluster of machines is at the disposal of a central managing authority of the pSPE.

2.1 Preliminaries

A query $\mathcal{Q}$ is submitted to the pSPE in either declarative or imperative form. For the rest of this section we are going to use the

```
SELECT R.a, COUNT(*)
FROM R JOIN S
     on R.a = S.b
[ Range 5 minutes
  slide 30 seconds
]
WHERE S.c < 100
GROUP BY R.a
```

a: Input Query $\mathcal{Q}$.



b: Evaluation tree.

Figure 2: Query submission and evaluation on a pSPE.

Table 1: Model Symbol Overview

Model Symbol Overview	
$\mathcal{V}$	# of workers
S_i	streams $1 \leq i \leq N$
$\mathcal{X}_i$	schema of S_i
$e_{\mathcal{X}_i}$	tuple of S_i
$\mathcal{W} : S_i \rightarrow \{S_i^1, \ldots, S_i^w\}$	window for S_i
$P : S_i^w \rightarrow \{L_{S_i^w}^1, \ldots, L_{S_i^w}^{\mathcal{P}}\}$	*partition* function
$L_{S_i^w}^{\mathcal{P}}$	window load of worker i
$f : L_{S_i^w}^{\mathcal{P}} \rightarrow \{\ldots, (k_x, v_x), \ldots\}$	*partial evaluation* function
$\Gamma : \{\ldots, f(L_{S_i^w}^j), \ldots\} \rightarrow R$	*aggregation* function

example query depicted in Fig. 2a, using CQL [4]. The pSPE transforms $\mathcal{Q}$ into a logical plan, which is often modeled as an evaluation tree (Fig. 2b). The root of the tree represents output, which can either be an external system consuming the result or external storage. The leaves of the tree are streams, each one represented by S_i, where $1 \leq i \leq N$ (N is the number of input streams). Each S_i is abstracted as a sequence of tuples $e_{\mathcal{X}_i}$ with a predefined schema $\mathcal{X}_i$. From this point on we are going to describe our model based on a single input stream S_i. However, without any loss of generality this model is capable of accommodating multiple streams as well.

An $e_{\mathcal{X}_i}$'s attributes can be represented as a triplet $(\tau_{\mathcal{X}_i}, k_{\mathcal{X}_i}, p_{\mathcal{X}_i})$. $\tau_{\mathcal{X}_i}$ is the attribute responsible for ordering tuples in S_i and is used to assign each $e_{\mathcal{X}_i}$ to a logical window (either time- or count-based). A logical window is abstracted as a function $\mathcal{W} : S_i \rightarrow \{S_i^1, ..., S_i^w\}$, where $w \rightarrow \infty$. Each S_i^w represents the tuples of S_i that belong to window w according to $\mathcal{W}$. $k_{\mathcal{X}_i} \subset \{\mathcal{X}_i - \tau_{\mathcal{X}_i}\}$ are the attributes, which identify a tuple, and $p_{\mathcal{X}_i} \subset \{\mathcal{X}_i - (\tau_{\mathcal{X}_i} + k_{\mathcal{X}_i})\}$ are the remaining attributes, which comprise $e_{\mathcal{X}_i}$'s payload. Often, those appear in predicates, projection lists, or are used by aggregate functions. In our example query, $S_1 = R$ and $S_2 = S$.

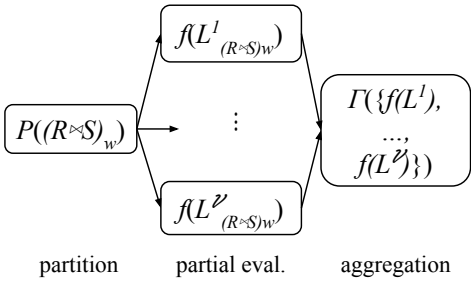


Figure 3: The windowed *group-by* count of the sample query (Fig. 2) as a 3 stage process.

$\mathcal{X}_1 = (t, a)$ and $\mathcal{X}_2 = (t, b, c)$. Each tuple from R is modeled as a triplet where $\tau_{\mathcal{X}_1} = \{R.t\}$, $k_{\mathcal{X}_1} = \{R.a\}$, and $p_{\mathcal{X}_1} = \emptyset$. Similarly, $\tau_{\mathcal{X}_2} = \{S.t\}$, $k_{\mathcal{X}_2} = \{S.b\}$, and $p_{\mathcal{X}_2} = \{S.c\}$.

Turning to the evaluation tree, internal nodes represent algebraic operations, which work as transformations of an input stream S_i to another S'_i. Each operation can be either *stateless* or *stateful*. The former are pure functions (as defined by functional programming principles) and are easily parallelized by arbitrarily partitioning their input stream. The latter can be either a relational algebra operation or any user-defined function that produces a result on every window S^w_i. Our work focuses only on partitioning tuples for *stateful* operations. In the tree illustrated on Fig. 2b, *map* and *filter* are *stateless*, whereas, *join* and *group-by* are *stateful* operations.

2.2 A new formulation for Parallel Stateful Operations

By the time a *stateful* operation is scheduled to execute in parallel, it transforms into a 3-stage process for each window S^w_i. Its input consists of $S^w_1, \ldots, S^w_N$ and the 3 stages are in order: (i) *partition*, (ii) *partial evaluation*, and (iii) *aggregation*. Figure 3 depicts the windowed *group-by* count between streams R and S of the sample query as the 3 stage process.

Partition can be modeled as a function that takes a subsequence S^w_i and produces another sequence of equal length that indicates the worker to which each $e_{\mathcal{X}^w_i}$ is going to be sent. In other words, *partition* is a function $P : S^w_i \rightarrow \{o_1, \ldots, o_{|S^w_i|}\}$, where $1 \leq o_l \leq \mathcal{V}$ ($|x|$ represents the length of a stream/sequence x). The resulting sequence consists of elements o_l, where $1 \leq l \leq |S^w_i|$, each one mapping $e_{\mathcal{X}^w_i}$ indexed by l to a number in $[1, \mathcal{V}]$. $\mathcal{V}$ represents the pSPE's parallelism degree for a particular *stateful* operation and is materialized by $\mathcal{V}$ workers, which are responsible for processing the partial result in window w. Each worker is either a thread or a process. $L^o_{S^w_i} = \{e \in e_{\mathcal{X}^w_i} | P_w(S^w_i)[e] = o\}$ denotes the sequence of tuples from S^w_i that will be sent to worker o, by the *partition* process. *Partial evaluation* is executed by $\mathcal{V}$ workers in parallel. Each worker receives its corresponding $L^o_{S^w_i}$ sequence and applies the user-defined transformation f. f produces a set of key-value pairs: $f : L^o_{S^w_i} \rightarrow \{(k_1, v_1), \ldots, (k_m, v_m)\}$ of arbitrary size m. m is naturally bounded by the cardinality of S^w_i, which is defined as the number of distinct values k_{S_i} in S^w_i. For the rest of this paper, the cardinality of a stream/sequence x will be represented by $\|x\|$, which is used to refer to the number of distinct keys (groups) held by a worker. Finally, *aggregation* combines all the key value pairs $f(L^o_{S^w_i})$ produced by each worker $o, \forall o \in [1, \mathcal{V}]$, into a final result, using an aggregation function $\Gamma(\{f(L^1_{S^w_i}), \ldots, f(L^{\mathcal{V}}_{S^w_i})\})$.

Going back to the query shown in Fig. 3, *partition* would be a function $P(\{R \bowtie S_w\})$ that partitions each windowed stream based on $R.a$. *Partial evaluation* would be the partial count of the *group-by* operator and the result of each worker would produce a sequence of *key value* pairs, in which the *keys* would consist of distinct $R.a$ values and *values* would be the number of tuples for each corresponding $R.a$ *key*. Finally, the *aggregation* stage would combine partial results by adding partial counts for every matching $R.a$ key. In essence, if 2 workers are used with worker $\#1$ producing $\{(x, 12), (y, 123)\}$ and worker $\#2$ producing $\{(x, 43), (y, 1), (z, 4)\}$, then *aggregation* ($\Gamma$) produces the result $\{(x, 55), (y, 124), (z, 4)\}$ (similar to the processing model of [33]).

2.3 Proposed partitioning cost model

Partition aims to: (i) divide S^w_i as evenly as possible among $\mathcal{V}$ workers, while (ii) aggregation load (Γ) remains low. This way, execution can benefit from employing multiple workers: *the more* $\max_{\mathcal{V}}(L^{\mathcal{V}}_{S^w_i})$ *gets reduced, the faster the partial evaluation step is going to progress*. In our work, we adopt the assumption that there exists a monotonic relation between the number of tuples and load increase (similar to previous work on stream partitioning [28]). This entails that when a tuple is assigned to a worker, its load will either increase or stay the same.

[28] introduced tuple *imbalance* as a metric for quantifying a *partitioner's* efficiency in terms of balancing load among workers. However, [28] expressed *imbalance* on the entire stream (i.e., counting from the beginning of time), which we believe is limited, given the dynamic nature and characteristics of data streams. In this work, we extend *imbalance* to cover the window aspect of a streaming query:

$$I(P(S^w_i)) = |\max_j(L^j_{S^w_i}) - \underset{j}{avg}(L^j_{S^w_i})|,\ j = 1, \ldots, \mathcal{V} \quad (1)$$

Equation 1 defines tuple *imbalance* as the difference of the maximum $L_{S^w_i}$ minus the average $L_{S^w_i}$, as partitioned by a partition algorithm P. The less the tuple *imbalance* achieved by P is, the less the maximum runtime of each worker will be.

We propose a new model for measuring the effectiveness of a *partitioner* by incorporating *aggregation* cost, which has been ignored in the past. As we discussed in Sec. 2.2, the *aggregation* stage will have to ingest all $f(L^{\mathcal{V}}_{S^w_i})$ and combine every pair of (k, v) tuples with a matching key k. Hence, the number of operations for processing partial results is proportional to the sum of the sizes of all partial aggregations $|f(L^{\mathcal{V}}_{S^w_i})|$:

$$\Gamma(S^w_i) = O(\sum_{o=1}^{\mathcal{V}} |f(L^o_{S^w_i})|) \quad (2)$$

Equation 2 captures both processing and memory cost of the *aggregation*, since partial results need to be stored until they are processed. In fact, the larger $\Gamma(S^w_i)$ is, the more memory is required to accommodate partial results. Therefore, we model stream partitioning as the following minimization problem:

$$\begin{aligned} &\text{minimize} && I(P(S^w_i)) \\ &\text{while} && \Gamma(S^w_i) \leq \max_{1 \leq j \leq \mathcal{V}}(L^j_{S^w_i}) \end{aligned} \quad (3)$$

The reason $\Gamma(S^w_i)$ should be less or equal than the maximum $L^{\mathcal{V}}_{S^w_i}$ is so that execution benefits from parallelizing the workload and not having *aggregation* become more than the maximum partial processing cost.

Finally, in *scale-out* architectures, workers' load might diverge due to external factors (i.e., communication, multi-tenancy etc.). Our model (Eq. 1) focuses on identifying load generated by the

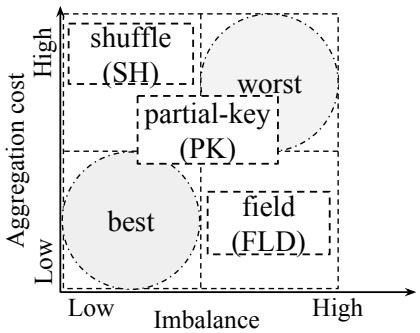


Figure 4: Stream partitioning algorithms expected performance.

stateful operation and act accordingly to balance it. To the extent of our knowledge, any method for broader load monitoring in a pSPE involves architectural interventions, such as monitoring modules and feedback loops [17, 38, 13, 18, 23, 31]. If a pSPE features the aforementioned components to detect load divergence caused by external factors, our cost model (Eq. 3) can be extended to incorporate that information as well [2].

2.4 The pitfall of ignoring aggregation costs

To better understand inherent trade-offs among existing partition algorithms, we present Fig. 4, which illustrates the two dimensions with which each algorithm is measured. The horizontal axis represents the *ability* of an algorithm to balance the load among workers, and the vertical axis represents an algorithm's *ability* to maintain the *aggregation* cost low, based on our model (Eq. 3). In Fig. 4 we have placed previously proposed partition algorithms based on their expected behavior in terms of *imbalance* and *aggregation* cost.

As indicated by Eq. 3, partitioning becomes a trade-off between tuple *imbalance* and *aggregation* cost: *the more tuples are spread, the more aggregation time increases*. Consider S_i to be an input stream with schema $\mathcal{X}_i = (t, a, b)$, where $\tau_{\mathcal{X}_i} = \{t\}$, $k_{\mathcal{X}_i} = \{a\}$ and $p_{\mathcal{X}_i} = \{b\}$. In a *stateful* operation, a partitioning algorithm has to make a choice of where all tuples with a particular key a will be sent. Partitioning algorithms can be categorized based on how many worker options are presented for a given $k_{\mathcal{X}_i}$.

A *1-choice partitioner* offers no mechanisms to balance skewness on input data. As a result, the workers that happen to be assigned the part of the data that appear the most (i.e., most frequent) will always have more work compared to others. That leads to higher *imbalance* (Eq. 1). In addition, when a single option for each $k_{\mathcal{X}_i}$ is presented, *aggregation* cost (Eg. 2) is minimal, because each worker will produce a subset of the full result. On the other hand, a M*-choice partitioner* ($M \leq \mathcal{V}$) presents M candidate workers for each $k_{\mathcal{X}_i}$. Thus, load for $k_{\mathcal{X}_i}$ is divided into M equal parts and handled by M workers. As a result, *imbalance* (Eq. 1) is reduced, and the pSPE takes better advantage of parallelism. Unfortunately, partial results produced by the M workers handling a particular $k_{\mathcal{X}_i}$ have to be gathered and combined. That entails an inflated *aggregation* cost, which is expected to increase by a factor of M. For example, in a single window S_i^w, if tuples with $k_{\mathcal{X}_i} = a_x$ are assigned to 4 workers, then the *aggregation* stage will process 4 partial results (i.e., one from each worker).

Shuffle Partitioning (round-robin) - SH blindly sends tuples to workers, without making any attempt to balance load and collocate keys (Fig. 1a). Therefore, SH is categorized as a M*-choice partitioner* because an *aggregation* stage is required to produce the final result. SH manages to minimize tuple *imbalance* (Eq. 1) since each worker receives the same number of tuples in a given window S_i^w: if $\mathcal{V}$ workers exist, each one will receive $\frac{|S_i^w|}{\mathcal{V}}$ tuples. Turning to *aggregation* cost $\Gamma(S_i^w)$ (Eq. 2), when SH is used it becomes computationally expensive, because tuples are partitioned without an attempt to collocate keys. Therefore, in a worst case scenario, each worker will produce a partial result ($f(L^{\mathcal{V}}_{S_i^w})$) with all the keys that exist in S_i^w (illustrated in Fig. 1a). In that case, $\Gamma(S_i^w)$ will become equal to M times $\|S_i^w\|$. As far as our cost model is concerned (Eq. 3), *SH minimizes* imbalance*, but does not act to limit the* aggregation *cost*.

Hash Partitioning (field) - FLD follows a different approach than SH, by collocating tuples with the same $k_{\mathcal{X}_i}$ on the same worker (Fig. 1b). FLD feeds $k_{\mathcal{X}_i}$ to a hash function and selects a worker based on the result. It guarantees that keys from the same group will be collocated, resulting in minimal *aggregation* cost (Eq. 2). Hence, FLD is characterized as a *1-choice partitioner*. Nevertheless, FLD fails to balance the load effectively when input is skewed and some keys appear more often than others. (i.e., there is tuple *imbalance* - Eq. 1). Matters can get exacerbated if initial expectations (or assumptions) on input load do not hold true overtime. Under such circumstances, "struggling" workers with excess load will hinder the progress of a query and even compromise the correctness of the result. In conclusion, FLD imposes minimal *aggregation* cost but does not act on limiting tuple *imbalance* (based on the cost model - Eq. 3).

Partial Key Partitioning - PK, is the current state of the art algorithm [28]. It adopted the idea of *key splitting* [5] to alleviate the load of processing keys that are part of the skew. PK was first to incorporate load in terms of the number of tuples assigned to each worker (i.e., $L^{\mathcal{V}}_{S_i^w}$). *Key splitting* is materialized by using a pair of independent hash functions (i.e., $\mathcal{H}_1, \mathcal{H}_2$) and feed $k_{\mathcal{X}_i}$ to both. Also, PK maintains an array of size $\mathcal{V}$ with the total tuple count sent to each worker. Every time a tuple arrives, its $k_{\mathcal{X}_i}$ is fed to $\mathcal{H}_1$ and $\mathcal{H}_2$ to identify 2 candidate workers. The partitioner will forward the tuple to the candidate that has received the least number of tuples up to that point. PK was extended to more than two candidates [29], when two are not sufficient to handle skew. Even though PK succeeded in improving *imbalance* (Eq. 1) compared to FLD, it did so by adding an essential *aggregation* step (Eq. 2). Therefore, PK is expected to incur *aggregation cost* proportional to the number of candidates. Turning to our cost model (Eq. 3), PK can potentially violate the *aggregation* cost constraint, when $\Gamma(S_i^w)$ exceeds the maximum workload experienced by each worker.

Summary: Our goal is to propose partitioning algorithms that belong to the *best* quartile (Fig. 4) and use our cost model (Eq. 3). To achieve this, we have to maintain the *aggregation* cost low and achieve better *imbalance*.

3. MINIMIZING IMBALANCE WITH LOW AGGREGATION COST

Designing a partitioning algorithm that achieves low *aggregation* cost entails keeping track of the number of keys produced by each worker on every window S_i^w (i.e., $f(L^j_{S_i^w})$, for $1 \leq j \leq \mathcal{V}$). Equation 2 indicates that if the sum of $f(L^j_{S_i^w})$ is reduced, then the *aggregation* cost gets reduced as well. However, the boundaries of the *aggregation* cost need to be identified first.

PROPOSITION 1. *For a given stream S_i^w, a* stateful *operation f, and $\mathcal{V}$ number of workers, $\Gamma(S_i^w)$ is bounded by:* $\|S_i^w\| \leq \Gamma(S_i^w) \leq \mathcal{V}\|S_i^w\|$.

[2]By changing Eq. 1 to multiply $L^j_{S_i^w}$ with *load-divergence* coefficients, produced periodically by monitoring components.

PROOF. $\Gamma(S_i^w)$ will always be greater or equal to $\|S_i^w\|$ and that happens when the partitioning algorithm sends each key to a single worker only. In this case, $L_{S_i^w}^i \cap L_{S_i^w}^j = \emptyset, \forall\, 1 \leq i \neq j \leq \mathcal{V}$. Hence, $\|L_{S_i^w}^1\| + \ldots + \|L_{S_i^w}^{\mathcal{V}}\| = \|S_i^w\|$. Similarly, if the partition algorithm sends at least one tuple for each key to every worker (i.e., $k \in \|L_{S_i^w}^j\|$, $\forall k \in S_i^w$ and $1 \leq j \leq \mathcal{V}$), then $\|L_{S_i^w}^1\| + \ldots + \|L_{S_i^w}^{\mathcal{V}}\| = \underbrace{\|S_i^w\| + \ldots + \|S_i^w\|}_{\mathcal{V}} = \mathcal{V}\|S_i^w\|$ □

A mechanism for monitoring $\Gamma(S_i^w)$'s value has to be established. Eq. 2 can be expanded to the sum of its operands as:

$$\Gamma(S_i^w) = |f(L_{S_i^w}^1)| + \ldots + |f(L_{S_i^w}^{\mathcal{V}})| \tag{4}$$

Hence, in order to monitor *aggregation* cost, the partition algorithm has to keep track of the number of distinct keys sent to each worker, for each S_i^w.

3.1 Incorporating Cardinality in Partitioning

Assuming a mechanism for keeping track of workers' cardinalities has been established, the cost model (Eq. 3) can be extended to incorporate the knowledge of the number of distinct keys sent to each worker. As indicated by Eq. 3, the information about workers' cardinalities can be used in two places: (a) *imbalance* (Eq. 1), and (b) *aggregation* cost (Eq. 2).

3.1.1 Cardinality in imbalance

The load of each worker has been modeled in terms of number of tuples. In the same manner, a worker's load can be expressed in terms of cardinality using the following formula:

$$CL_{S_i^w}^j = \|L_{S_i^w}^j\|, 1 \leq j \leq \mathcal{V} \tag{5}$$

Equation 5 depicts the load of a worker in terms of the number of distinct keys sent to it. Therefore, cardinality *imbalance* can be expressed as the difference between the maximum and the mean cardinality of all workers for a given window S_i^w, as a result of a partitioning algorithm P:

$$CI(P(S_i^w)) = \max_j(CL_{S_i^w}^j) - \underset{j}{avg}(CL_{S_i^w}^j), 1 \leq j \leq \mathcal{V} \tag{6}$$

At this point, *imbalance* is determined by tuple count and cardinality. However, different *stateful* operations are affected by each metric differently. Hence, there is a need for a more diverse load estimation formula, which combines tuple count and cardinality. In order to avoid one metric dominating the other, the initial values should be scaled accordingly:

$$L_{S_i^w}^j{}' = \frac{L_{S_i^w}^j - \min_{1\leq k\leq \mathcal{V}}(L_{S_i^w}^k)}{\max_{1\leq k\leq \mathcal{V}}(L_{S_i^w}^k) - \min_{1\leq k\leq \mathcal{V}}(L_{S_i^w}^k)} \tag{7}$$

$$CL_{S_i^w}^j{}' = \frac{CL_{S_i^w}^j - \min_{1\leq k\leq \mathcal{V}}(CL_{S_i^w}^k)}{\max_{1\leq k\leq \mathcal{V}}(CL_{S_i^w}^k) - \min_{1\leq k\leq \mathcal{V}}(CL_{S_i^w}^k)} \tag{8}$$

$$H_{S_i^w}^j = pL_{S_i^w}^j{}' + (1-p)CL_{S_i^w}^j{}', \text{ where } 1 \leq j \leq \mathcal{V} \tag{9}$$

Equation 9 combines the normalized loads both in terms of tuples (Eq. 7) and distinct keys (Eq. 8) in a unified score. That score is adjustable based on a user's (or query optimizer's) parameter p, which controls the bias for each score accordingly: *the smaller the p, the less the load in terms of tuples affects Eq. 9; whereas the higher the p, the less the load in terms of distinct keys affects Eq. 9*. Finally, *imbalance* can be expanded to a hybrid form that incorporates load in terms of both tuple count and cardinality:

$$HI(P(S_i^w)) = \max_j(H_{S_i^w}^j) - \underset{j}{avg}(H_{S_i^w}^j),\ 1 \leq j \leq \mathcal{V} \tag{10}$$

3.1.2 Cardinality in aggregation

Aggregation cost is determined by $\Gamma(S_i^w)$ (Eq. 2) and reducing it emanates from reducing the sum of distinct keys sent to each worker. Its minimum value can be $\|S_i^w\|$ when each key is sent to only a single worker. This behavior resembles FLD and it might result in *imbalance* on workers. To avoid this, we employ *key splitting* for keys that have not been sent to a worker before in a particular window S_i^w. By sending each newly encountered key to the worker with either the least keys or the least number of tuples up to that point, the *aggregation* cost remains low. Also, *imbalance* is expected to be lower compared to the one achieved from FLD.

3.2 Cardinality Estimation data structures

The *partitioner* needs to keep track of each worker's cardinality, every time a new tuple arrives. Hence, it should maintain an array of $\mathcal{V}$ cardinality estimation structures (C), which will offer two methods: (i) *update($k_{\mathcal{X}_i}$)*: for updating the count of distinct keys; and (ii) *estimate()*: for returning the count of distinct keys.

3.2.1 Naive

The naive approach for estimating a worker's cardinality involves keeping track of the exact number of distinct keys. Therefore, a partitioner responsible for $\mathcal{V}$ downstream workers, $\mathcal{V}$ unordered set structures are needed. This way, the *update* and the *estimate* methods will offer constant execution time (O(1)).

One caveat of using an unordered set structure for each worker is the memory overhead. Depending on the algorithm used, a key can end up in multiple workers (e.g., SH). This way, the memory required for maintaining the number of keys on each worker can become $O(\mathcal{V}\|S_i^w\|)$, since all unordered sets can end up having each key. The memory cost of a naive cardinality estimation structure is related to the cardinality of S_i^w and the choice of the partitioning policy: If $\|S_i^w\|$ remains low and the partitioner does not send the same keys to multiple workers, the memory requirements for C will remain low. However, if $\|S_i^w\|$ is high and the partitioner tends to send tuples with the same key to multiple workers, then memory load can hinder the partition process.

3.2.2 Hyperloglog

HyperLogLog (HLL) introduced by Flajolet et al. [14] is an algorithm for estimating the number of distinct elements in databases with a bounded error. HLL requires O ($\log_2 \log_2 N$) memory for a relation expected to have N distinct elements. Every time a new tuple arrives, its $k_{\mathcal{X}_i}$ is extracted and fed through a hash function. HLL extracts the m most significant bits of the hash result, and uses them to identify which register (out of 2^m) to update. Each register is $\log_2 \log_2 N$ bits long, and its value is updated depending on the left-most zero of the m most significant bits of the hashed value. HLL has been shown to present an accuracy of $\frac{1.04}{\sqrt{m}}$. Recent work from Heule et al. [20] presented a number of improvements that need to take place so that cardinalities in the orders of billions can be estimated efficiently. For cardinality estimation, a partition algorithm is required to use $\mathcal{V}$ HLL's to measure the number of distinct keys sent to each of the $\mathcal{V}$ worker.

HLL can be used to limit the memory cost but it uses irreversible operations to update its internal buckets. That constitutes it unable to check whether a key has been previously sent to a worker. We introduce an optimistic mechanism for checking if a key has been

Algorithm 1: Partition.

input : $e_{\mathcal{X}_i}$, t_w, C, L
output: worker to which $e_{\mathcal{X}_i}$ will be sent to

```
1 k = GetKey(e_{X_i});
2 t = GetTimestamp(e_{X_i});
3 if t ≥ t_w then
4 |   Reset(C);
5 |   Reset(L);
6 c_1 = H_1(k);
7 c_2 = H_2(k);
8 return decide(C, L, k, c_1, c_2);
```

forwarded to a particular worker before: upon the arrival of a key that hashes to a worker i, its cardinality c_i is estimated (call to *estimate*). A trial update of c_i is performed and the new cardinality c_i' is estimated. If the cardinality estimate difference $\Delta(|c_i - c_i'|) = 0$, then our mechanism optimistically assumes that the key has already been sent to the corresponding worker. HLL is expected to make wrong decisions at the benefit of a constant memory cost.

4. PROPOSED CARDINALITY-AWARE PARTITIONING ALGORITHMS

PK [28, 29] has motivated the merits of *key splitting* [5] for reducing *imbalance* among workers. Therefore, all the variations of our proposed algorithms leverage *key splitting*, which materializes with the use of multiple hash functions for identifying candidate workers. For simplicity, our algorithms are presented with only two candidates, but they can be extended to accommodate more.

The basis of our algorithms is presented in Alg. 1 and is called by the pSPE's *partitioner* when a new tuple arrives. The *partitioner* maintains two arrays of size $\mathcal{V}$: one with cardinality estimation structures (C), and one with tuple counters (L). L is identical to the one used by PK and gets updated the same way in all our proposed algorithms. $e_{\mathcal{X}_i}$'s key is extracted and fed to the two hash functions: $\mathcal{H}_1$ and $\mathcal{H}_2$. If the partitioner uses more than two candidates (i.e., $M > 2$), then an equal number of hash functions are used in the decision process. The resulting choices (c_1 and c_2) along with the arrays C and L are passed to *decide()*.

During query execution, a pSPE might have multiple instances of partitioners running on different machines (especially in a *scale-out* setting, where thousands of threads are involved in a query). The advantage of using hash functions is that no exchange of information is required among different instances of *partitioners*. On top of that, C and L have their counts monotonically increasing on each window. Therefore, if each of the *partitioners* tries to reduce *imbalance* and/or *aggregation* cost, then (through the additive property) the overall *imbalance* and/or *aggregation* cost are reduced. Finally, C and L need to be reset when a window expires (Alg. 1 line 3). This guarantees that decisions reflect the temporal nature of stream processing. Algorithm 1 receives t_w as an argument, which is the expiration timestamp of the current window.

In the following sections we go over our variations for *decide()*: (i) Cardinality *imbalance* Minimization (CM), (ii) Group Affinity with *imbalance* Minimization (AM & cAM), and (iii) Hybrid *imbalance* Minimization (LM).

4.1 Cardinality Imbalance Minimization (CM)

The first partitioning algorithm aims at limiting cardinality *imbalance* (Eq. 6) and the decision is made based on the cardinality estimate retrieved by the C array structure (Eq. 5). The newly arrived tuple $e_{\mathcal{X}_i}$ is sent to the candidate worker that has the least cardinality. Algorithm 2 illustrates the cardinality *imbalance* minimization algorithm (CM), which works as a counterpart to PK. CM can have its cardinality estimation structure be either the Naive (Section 3.2.1) or the HLL with our optimistic mechanism (Section 3.2.2).

Algorithm 2: Cardinality *imbalance* minimization (CM)

input : C, L, k, c_1, c_2
output: worker to which the tuple is going to be sent to

```
1  l_1 = C[c_1].estimate();
2  l_2 = C[c_2].estimate();
3  if l_1 ≤ l_2 then
4  |   C[c_1].update(k);
5  |   L[c_1] += 1;
6  |   return c_1;
7  else
8  |   C[c_2].update(k);
9  |   L[c_2] += 1;
10 |   return c_2;
```

Algorithm 3: Group affinity combined with cardinality *imbalance* minimization (AM)

input : C, L, k, c_1, c_2
output: worker to which the tuple is going to be sent to

```
1  if C[c_1].contains(k) then
2  |   L[c_1] += 1;
3  |   return c_1;
4  else if C[c_2].contains(k) then
5  |   L[c_2] += 1;
6  |   return c_2;
7  else
8  |   l_1 = C[c_1].estimate();
9  |   l_2 = C[c_2].estimate();
10 |   if l_1 ≤ l_2 then
11 |   |   C[c_1].update(k);
12 |   |   L[c_1] += 1;
13 |   |   return c_1;
14 |   else
15 |   |   C[c_2].update(k);
16 |   |   L[c_2] += 1;
17 |   |   return c_2;
```

This algorithm is expected to be used in operations in which processing cost is dominated by the amount of distinct keys. This way, *imbalance* in terms of cardinality will be minimal. However, *imbalance* in terms of tuple counts will be increased, since CM is tuple count agnostic and makes no effort on limiting *aggregation* cost.

4.2 Group Affinity and Imbalance Minimization (AM & cAM)

Group Affinity algorithms try to impose no additional *aggregation* cost, while balancing load with the use of *key splitting*. The name affinity comes from keeping track of whether a key has been encountered before in S_i^w, and if it did, then it is forwarded to the worker that received it previously.

The first variation of affinity based algorithms, is AM and focuses on cardinality *imbalance* (Alg. 3). AM tries to minimize *aggregation* cost by not splitting keys among workers. First, it checks

Algorithm 4: Group affinity with *imbalance* minimization (cAM)

input : C, L, k, c_1, c_2
output: worker to which the tuple is going to be sent to

```
if C[c1].contains(k) then
    L[c1] += 1;
    return c1;
else if C[c2].contains(k) then
    L[c2] += 1;
    return c2;
else
    l1 = L[c1].estimate();
    l2 = L[c2].estimate();
    if l1 ≤ l2 then
        C[c1].update(k);
        L[c1] += 1;
        return c1;
    else
        C[c2].update(k);
        L[c2] += 1;
        return c2;
```

Algorithm 5: Hyblid *imbalance* minimization (LM)

input : C, L, k, c_1, c_2
output: worker to which the tuple is going to be sent to

```
hl1 = p·L^{c1}_{S^w_i} + (1 − p)·CL^{c1}_{S^w_i};
hl2 = p·L^{c2}_{S^w_i} + (1 − p)·CL^{c2}_{S^w_i};
if hl1 ≤ hl2 then
    C[c1].update(k);
    L[c1] += 1;
    return c1;
else
    C[c2].update(k);
    L[c2] += 1;
    return c2;
```

if one of the candidate workers has encountered key k previously. If one of them did, then the tuple is forwarded to that worker; otherwise, it is sent to the worker with the least cardinality up to that point. A different variation of AM, named cAM (Alg. 4) behaves similarly, but it forwards the tuple to the worker with the least tuple count up to that point. This way, both *aggregation* cost and *imbalance* are considered during partitioning. Despite the fact that AM and cAM resemble FLD, they are expected to perform better because of the multiple number of choices that are presented to them.

4.3 Hybrid Imbalance Minimization (LM)

For *stateful* operations equally affected by tuple count and cardinality, we propose the hybrid load *imbalance* minimization algorithm (LM). It combines a worker's tuple count with cardinality and calculates hybrid load as indicated in Eq. 9. A tuple is forwarded to the worker with the least load and LM's main goal is to minimize hybrid load imbalance (Eq. 10). LM is depicted on Algorithm 5.

5. EXPERIMENTAL SETUP

Our experiments were conducted on an AWS c4.8xlarge instance, running Ubuntu v14.04. For all experiments, we used our own multi-threaded stream partitioning library, developed in C++11 and compiled with GCC v4.8.2. Our performance analysis involved a varying numbers of concurrent worker threads (8 up to 32), and data partitions (from 8 to 256). The reason we did not experiment with more threads was because we did not want to pollute results with context-switching overheads. All reported runtimes are the averages of 7 runs, after removing minimum and maximum reported times, to compensate for anomalies related to running concurrent processes.

Table 2: Stream partitioning algorithms. w is the total number of workers.

Symbol	Algorithm	Choices	Cardinality Estimation Structure used
SH-w	shuffle	w	*None*
FLD-1	field	1	*None*
PK-2	partial-key [28]	2	*None*
PK-5	partial-key [29]	5	*None*
CM-2	Alg. 2	2	Naive, Sec. 3.2.1
AM-2	Alg. 3	2	Naive, Sec. 3.2.1
AM-5	Alg. 3	5	Naive, Sec. 3.2.1
cAM-2	Alg. 4	2	Naive, Sec. 3.2.1
cAM-5	Alg. 4	5	Naive, Sec. 3.2.1
LM-2	Alg. 5	2	Naive, Sec. 3.2.1
CM-2-H	Alg. 2	2	HLL, Sec. 3.2.2
AM-2-H	Alg. 3	2	HLL, Sec. 3.2.2
LM-2-H	Alg. 5	2	HLL, Sec. 3.2.2

5.1 Stream Partitioning Algorithms

We evaluated algorithms *shuffle* (SH), *field* (FLD), and *partial key* (PK) [28] (with 2 and 5 candidate workers), along with different variations of our proposed algorithms: *Cardinality Imbalance Minimization* (CM), *Group Affinity with Cardinality Imbalance Minimization* (AM), *Group Affinity with Imbalance Minimization* (cAM) and *Hybrid Imbalance Minimization* (LM). For all variation of LM we set p to 0.5 to achieve unbiased load estimation.

As a reference implementation for SH, FLD and PK we used the ones found in Apache Storm. In addition, we used the open source implementation of Murmur-Hash v3. All our proposed algorithms appear in two versions: one with naive and one with HLL as the cardinality estimation structure. For the former, we used C++ STL's implementation of unordered set, and for HLL, we implemented our version with $4096 (= 2^{12})$ registers and a register size of 5 bits. The choice of the number and size of registers was made to accommodate up to 10^7 distinct keys of 32 bits, with an accuracy lower than 2%, as instructed in [14]. Table 2 explains the algorithm symbols we use in our graphs.

5.2 Data sets and Workloads

Table 3 summarizes the characteristics of each data set/benchmark used in our experiments. Below, we go over each data set and the queries we used in our study.

TPC-H (TPCH): TPCH has been extensively used for throughput oriented streaming scenarios [13, 30, 10, 8, 12]. Out of 22 TPCH queries, 16 of them feature a *grouping* statement: half maintain a constant and half a scaling number of groups that increases when the *scale factor* grows. Due to the fact that our work addresses *stateful* operations, we focused on *grouping* TPCH queries and picked Query 1 (as a constant *grouping* query) and query 3 (as a scaling *grouping* query). Those two differ significantly in the number of resulting groups, and this enabled us to document the performance of different partition algorithms, when the *aggrega-*

Table 3: Summary of data characteristics.

Dataset	Size	Groups	Window	Metric
TPC-H	10GB	4 up to ∼100k	N/A	throughput
DEBS	32GB	62.5K up to 8.1M	sliding	latency
GCM	16GB	4 to ∼670K	sliding	latency

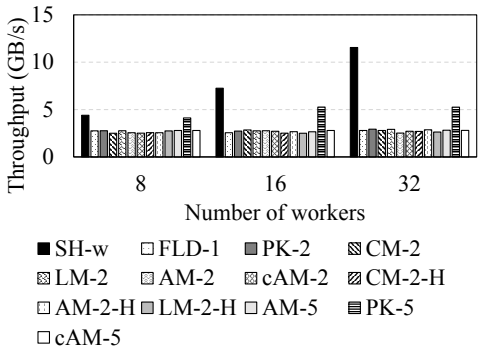


Figure 5: TPCH Query 1 performance (throughput).

tion cost varies in terms of size. As indicated in Table 3, Query 1 presents 4 and Query 3 presents up to 110000 resulting groups. Data were generated using the `dbgen` tool (v2.17) with a *scale factor* of 10.

ACM DEBS 2015 Grand Challenge (DEBS): DEBS totals 32 GBs in raw size, and comes with two sliding window queries [21]: (i) the Top-10 most frequent routes (Query 1), and (ii) the Top-10 most profitable areas (Query 2). DEBS presents a per window latency oriented data set, and its two queries offer group numbers that can potentially range from 62.5 thousand to 8.1 million.

Google Cluster Monitoring (GCM): This data set contains execution traces from one of Google's cluster management software systems, and for it we used two sliding window queries, which, like DEBS, are per window latency oriented. GCM Query 1 was taken from [24], and scans the `task_event` table to calculate every 60 seconds (with a slide of 1 second) the total CPU cores requested by each scheduling class. In addition, we introduced GCM Query 2 that calculates every 45 minutes (with a slide of 1 second) the average CPU cores requested by each job ID. There are more than 600 thousand job IDs in the whole data set.

6. EXPERIMENTAL RESULTS

Our experiments evaluate the impact of a partition algorithm on performance (Sec. 6.1), in terms of throughput (using the TPCH data set) and window latency (using the DEBS data set). Moreover, we evaluate the scalability (Sec. 6.2) of our algorithms compared to the state of the art (using the GCM data set). For all experiments, data were loaded in main memory before execution. The time to load data and write output to storage was not included in the reported times. Finally, for the experiments of Sec. 6.1 and 6.2, the time it takes to partition tuples is not included, because it is analyzed in Sec. 6.3 in terms of both processing and memory costs.

6.1 Performance

In this set of experiments, we used the TPCH and DEBS benchmarks to evaluate performance.

6.1.1 TPCH Query 1 (Fig. 5)

Figure 5 indicates that for TPCH Query 1, SH-w performs the best. This behavior is expected since there are only 4 groups for Query 1. Therefore, *aggregation* cost is negligible and performance is affected only by tuple *imbalance*. SH-w is expected to offer optimal tuple *imbalance* (≤ 1), which is reflected on results shown in Fig. 5. Those agree with our model (Eq. 3), which identifies that SH-w offers minimal *imbalance* with constant *aggregation* runtime of $O(4\mathcal{V})$ ($\mathcal{V}$ is the number of workers). In addition, PK-5 offers the next best throughput, since it reduces tuple *imbalance*, compared to all other algorithms with 2 and 5 alternative choices per group. CM and LM do not scale well with two choices, since they are affected by cardinality *imbalance*. LM is expected to perform similarly to PK, if p takes a value of 1 (as indicated in Eq. 9). Turning to *1-choice partitioners* (i.e., FLD-1, AM and cAM), they present constant performance and do not scale when the number of workers increases. This happens because each group, is presented to a single candidate worker.

Take-away: If the number of groups is constant and much smaller than the size of the *aggregation*, SH performs the best.

6.1.2 TPCH Query 3 (Figs. 6 - 7)

The query plan of TPCH Query 3 consists of a parallel hash join for the *customer* and *orders* tables, followed by a broadcast join with the *lineitem* table. Then, a parallel computation of the *group by* follows, and execution concludes with a final aggregation step to materialize the result.

Figure 6 illustrates the performance of *1-choice partitioners* (i.e., FLD-1, AM, and cAM), SH-w, PK-2, and PK-5. *M-choice partitioners* performed from 2.5x up to an order of magnitude worse (LM and CM offered similar performance to PK-2). As shown on Table 3, TPCH Query 3 involves 110 thousand groups (before applying the *limit* statement), and *aggregation* can take up to 60% of total execution time for *M-choice partitioners*. As a result, *M-choice partitioners* (i.e., SH, PK, CM, and LM) experience a substantial performance overhead on the final *aggregation* step.

Figure 7 illustrates the relative to FLD-1 tuple *imbalance* achieved by different variations of AM and cAM. Even though, AM-2-H achieves better tuple *imbalance* compared to FLD-1, it fails to perform in the same level as AM-2. Tuple *imbalance* results justify the throughput shown on Fig. 6, in which (apart from AM-2-H) all variations of our proposed algorithms perform significantly better than FLD-1. By adopting *key splitting*, throughput increases with the use of multiple candidate workers. AM-5 and cAM-5 offer improved throughput up to 47% compared to FLD-1.

Take-away: For throughput-oriented queries, with a large number of groups, cAM and AM perform the best. They achieve up to an order of magnitude better throughput compared to PK, and outperform FLD by up to 47%.

6.1.3 DEBS Query 1 (Figs. 8a - 8c)

Turning to DEBS, both queries involve window semantics and the performance is measured in window latency. Figure 8 shows the mean and 99 percentile window latency achieved by each partition algorithm. It is clear that in all worker settings, FLD-1, AM-2, cAM-2, AM-2-H, AM-5, and cAM-5 perform the best. This emanates from a lack of *aggregation* overhead, which constitutes those algorithms scalable when the number of workers increases. In fact, *aggregation* cost amounts for more than 70%, 84%, and 88% of total runtime for SH-w, PK-2, PK-5, CM-2, and LM-2. Finally, AM-2-H achieves identical performance with AM-2, which leads us to believe that our optimistic mechanism for cardinality estimation maintains a low error.

Take-away: For latency-oriented *stateful* queries, AM, and cAM perform from 4.5x up to 11.6x better compared to PK.

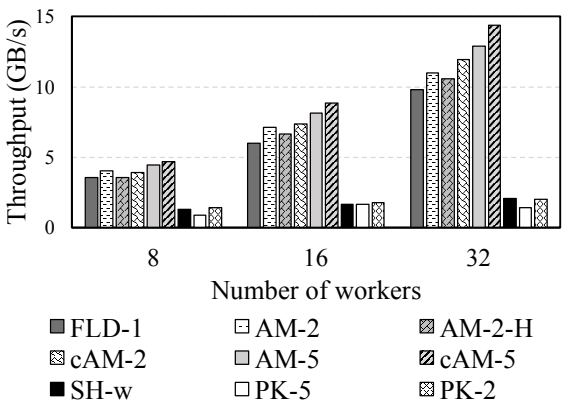


Figure 6: TPCH Query 3 performance (throughput).

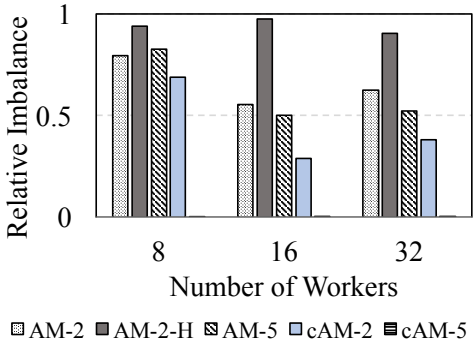


Figure 7: TPCH Query 3 relative imbalance to FLD.

6.1.4 DEBS Query 2 (Figs. 9a - 9c)

The execution plan starts by partitioning incoming tuples based on the medallion of each ride and each worker has to create two local indices: one for accumulating fares for each pickup cell, and one for keeping track of the latest drop-off cell. Then, an *aggregation* step follows, which gathers each pickup cell's fares and determines the latest cell for each medallion. The two resulting streams are partitioned based on pickup and drop-off cell IDs. Next, a gather step is executed, in which the median fare and the number of vacant taxis are processed to calculate the profit of each cell. Finally, partial results are merged and ordered to produce the Top-10 most profitable cells. This query represents a problematic case for our model (Eq. 3), because partitioning does not necessarily affect the workload imposed on each worker (i.e., the partitioning key is the medallion but each worker's state is affected by the number of distinct cells).

Fig. 9 depicts window latency achieved by each algorithm, and it is apparent that FLD-1, AM-2, AM-5, cAM-2, cAM-5, and AM-2-H offer the best performance. AM and cAM in all their variations outperform FLD in both mean (from 1.2x up to 1.5x) and 99 percentile (from 1.3x to 1.9x) latency. This is justified by AM's and cAM's ability to partition data more evenly compared to FLD. *M-choice partitioners* underperform because they do not act on limiting *aggregation* overhead. In comparison with PK (in both PK-2 and PK-5), AM and cAM perform up to 5.7x faster. In order to examine AM's and cAM's scalability, we also ran DEBS Query 2 with 64 and 128 workers. They performed up to 6.2x better than PK and up to 2.3x better than FLD.

Take-away: For latency-oriented complex queries, with more than one *stateful* operations, AM and cAM have window latency between 1.2x and 1.9x lower than FLD, and up to 5.7x lower than PK.

6.2 Scalability

We used the GCM dataset to measure the scalability of AM and cAM compared to SH and PK. The reason for picking GCM for scalability experiments is because it presents a conventional monitoring scenario, in which groups are not significantly more than the number of tuples in a window (like in TPCH and DEBS), and the queries consist of a single *stateful* operation. This way, *M-choice partitioners* would not be impeded by the *aggregation cost*.

6.2.1 GCM Query 1 (Figs. 10 & 11)

GCM Query 1 features up to 4 groups and differs from TPCH Query 1 because the number of tuples in every window is comparable to the number of groups (the average window size is 42 groups). For this query, we measured SH's, AM's and cAM's scalability compared to PK-2, which is the current state of the art and is expected to be scalable due to the small number of groups (as in Sec. 6.1.1).

Fig. 10 presents the percentage improvement in window latency of SH, AM, and cAM compared to PK-2. SH-w has its latency improvement decrease, because *aggregation cost* increases when more workers are employed. In contrast, AM and cAM have their latency decrease when the number of workers increases and they exhibit lower latency than PK-2. As Fig. 11 indicates, AM's and cAM's scalability results from their constant *aggregation* cost while the partial evaluation latency decreases. The former is not the case with SH-w and PK-2, which have the *aggregation cost* percentage increase with the number of workers.

Take-away: AM and cAM are scalable, maintain a constant *aggregation cost*, and outperform PK-2 by up to 1.3x.

6.2.2 GCM Query 2 (Fig. 12)

Fig. 12 illustrates the percentage improvement in window latency of SH-w, AM, and cAM over PK-2. Even though this query contains a large number of groups (Table 3), its average window size is only 181 tuples and group repetition is scarce. Therefore, *M-choice partitioners* will not have their performance deteriorate due to an overwhelming *aggregation cost* (the case in TPCH Query 3-Sec. 6.1.2). However, SH-w is not scalable because when additional workers are employed its *aggregation cost* becomes higher. Turning to PK-2, it manages to be scalable, but it underperforms compared to AM and cAM in all worker settings.

Take-away: AM and cAM are scalable and present more than 1.4x better latency compared to PK.

6.3 Partition algorithm cost

In this set of experiments, we measure overhead imposed by each algorithm in terms of processing and memory cost. To that end, we picked DEBS Q1, because it features the longest group identifier (15 bytes), and the number of groups can reach up to 8.1M.

6.3.1 Partition latency (Figs. 13a - 13c)

To measure processing time, we marshaled DEBS data to each algorithm and measured partition latency on each window. As described in Sec. 3.2, the cardinality estimation structure size relies on the number of workers. Therefore, we measured partition latency for 8, 16, and 32 workers. Figure 13 illustrates the total time spent on each window with each partition algorithm. We included 90 and 99 percentile window latency. Most of the algorithms present constant values for 8, 16, and 32 workers. However, noticeable

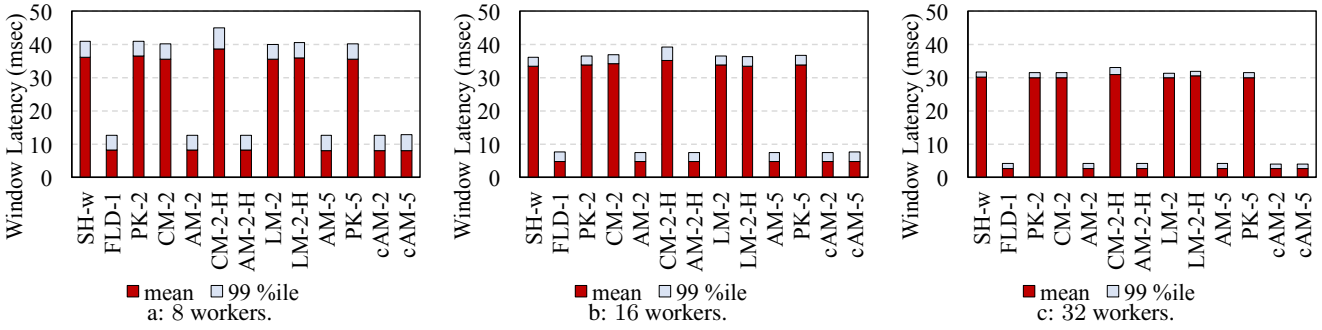


Figure 8: DEBS Query 1 performance (window latency).

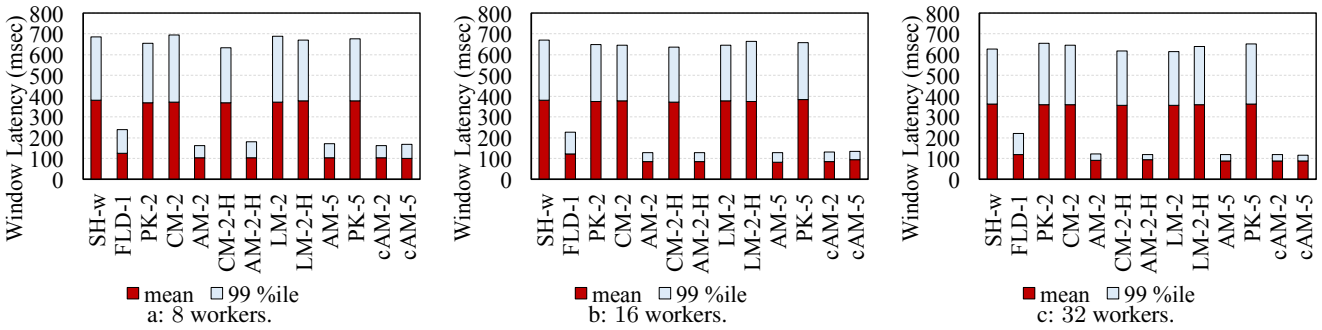


Figure 9: DEBS Query 2 performance (window latency).

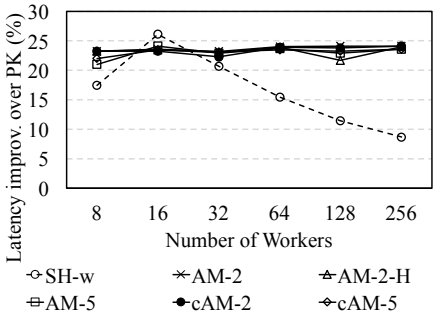


Figure 10: Latency percentage improvement over PK for GCM Query 1.

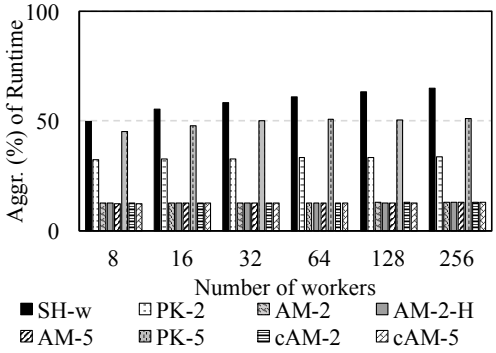


Figure 11: Aggregation percentage of runtime for GCM Query 1.

difference can be seen with 32 workers (Fig. 13c) for the 99 percentile window latency of CM-2, CM-2-H, LM-2, and LM-2-H. The increase is a result of additional processing required for those algorithms.

Take-away: Using our proposed algorithms does not incur any noticeable overhead in latency.

6.3.2 *Partition Memory (Table 4)*

Our proposed algorithms make use of cardinality estimation structures. Hence, we ran a micro-benchmark, in which we produced each possible key and replicated it to both available candidate workers. This experiment aims at examining an extreme scenario, in which all of the 8.1M groups appear in a single window. We measured memory consumed in MBs (Table 4). The naive cardinality estimation structure size quickly increases with the number of keys. Since each key is sent to both of the two candidates, the naive cardinality estimation structure's size increases further. Conversely, when HLL is used, memory consumption increases when the number of workers increases and its size does not get affected by neither the number of keys, nor the number of candidates. However, if the expected cardinality of the input stream is more than 10 million, then each HLL structure needs to double its number of buckets.

Take-away: Memory requirements of the cardinality estimation structure can be significantly limited with the use of HLL.

7. DISCUSSION

In conclusion, a pSPE's performance can be affected by both *imbalance* and *aggregation* cost. According to our experimental results, the state of the art solution (i.e., PK) fails to perform well, when a large number of groups appears, and *1-choice partitioners* like FLD can make use of *key splitting* [5] to achieve better performance.

Maintaining low *imbalance* does not necessarily lead to limiting *aggregation* cost. Even if an "improved" and diverse load metric is used (i.e., CM with Eq. 6 and LM with Eq. 10), performance will degrade when the number of groups increases. In fact, *M-choice partitioners* underperform when a large number of keys appear, because they focus solely on minimizing *imbalance*. After conducting a sensitivity analysis on *M-choice partitioners* and their

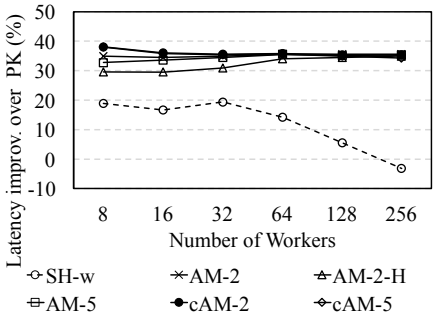


Figure 12: Latency percentage improvement over PK for GCM Query 2.

Table 4: Memory requirements in MBs of proposed algorithms for DEBS Query 1.

DEBS Query 1			
no. of workers	8	16	32
CM-2	243	243	243
AM-2	122	122	122
LM-2	243	243	243
CM-2-H	0.02	0.04	0.08
AM-2-H	0.02	0.04	0.08
LM-2-H	0.02	0.04	0.08

inaction for limiting *aggregation cost*, we found out that only under specific circumstances they achieve minimal *aggregation cost* (i.e., $\Gamma(S_i^w) = O(\|S_i^w\|)$). One such scenario is when tuples appear in an order that prohibits key sharing among workers. For example, S_i^w contains tuples with keys x and y, which map to 2 available workers w_1 and w_2. If all tuples with key x appear in even positions, and all tuples with key y appear in odd positions, then $\Gamma(S_i^w) = O(2)$. Another scenario in which LM achieves minimal *aggregation cost* is when keys are uniformly distributed and each one appears at most once. None of the data sets we used in our experiments demonstrated any of these scenarios and we could not find real-world data sets that behaved as such.

To address the important question of when to use each partition algorithm. As indicated by our experiments, SH is the best option for *stateful* operations, in which the expected number of groups is constant and far less than the window size divided by the number of workers. Our results on TPCH Query 1 (Fig. 5) agree with the previous statement, since SH performed the best and was the only scalable algorithm. Conversely, when a *stateful* operation involves a large number of groups, that constitute a significant percentage of total tuples on each window, AM and cAM have to be used. They offer minimal *aggregation* cost (like FLD), while leveraging the merits of *key splitting* to decrease *imbalance*.
Summary: We codify our results into the following:

Stream partitioner selection Rule: *If the number of groups in a stateful operation are expected to be constant and significantly less than the average number of tuples assigned to a worker on each window, then SH must be used. Otherwise, AM and cAM will offer the best trade-off between imbalance and aggregation cost.*

8. RELATED WORK

Load Balancing: Seminal work on stream re-partitioning is presented in Flux [32], in which the Flux operator is presented for monitoring load on each operator. Compared to our work, Flux employs FLD to distribute tuples and is orthogonal to ours because we do not consider solutions that employ state migration and re-partitioning in the event of *imbalance*. Furthermore, Flux's model does not consider the *aggregation* cost of *stateful* operations. Closer to our work is PK, presented by Nasir et al. [28], which combines *key splitting* when tuple *imbalance* appears. Our proposed algorithms rely on a more complete model that considers both tuple *imbalance* and *aggregation* cost. Thus, as shown in Sec. 6.1, our proposed AM and cAM algorithms achieve better performance. Recent work on partitioning by Rivetti et al. [31] has presented a solution for online adjustment of SH. Their goal is to decrease processing latency by making better decisions of tuple counts. Our work differs from theirs in the sense that we aim to improve stream partitioning in terms of both *imbalance* and *aggregation* cost. The model presented in [31] does not consider *aggregation* cost, and it only focuses on SH, which is expected to underperform in *stateful* operations that involve multiple groups. In addition, THEMIS [22] presented a federated way of controlling load shedding to preserve balanced execution in a pSPE. Our work is orthogonal to THEMIS because our goal is to improve tuple *imbalance* by altering the partitioning algorithm and not by shedding tuples.

Elasticity - Query Migration: Previous work that involves query (or state) migration [32, 36, 38, 16, 9] is orthogonal to our work. Our focus is on the performance of the partition algorithm and its performance in the event that migration is not an option. Nevertheless, our partition algorithms can be integrated as part of a pSPE that allows for state migration, but is beyond the focus of this paper. Work by Gedik [15] performs an extended study on partition algorithms for pSPEs by improving state migration and adaptability by measuring skewness. That work is similar to ours in terms of monitoring input stream cardinalities, but, our model differs because it incorporates the *aggregation* cost in the decision process. Finally, a lot of work focuses on online elasticity of a pSPE and adaptive behavior based on the input. Recent works [13, 26] aim to combine efficient stream partitioning and load migration for distributed stream joins. Additional work has been done on scaling-out [35, 18, 19, 23], where pSPEs are presented with the ability to detect struggling operators, decide on migration policies, and perform online re-configuration of the execution plan. Online re-configuration and state migration are outside the scope of this work.

9. CONCLUSIONS

We presented a new model for stream partitioning which considers tuple *imbalance* and *aggregation* cost for *stateful* operations. Inspired by it, we introduced novel partition algorithms, which adopt our model by keeping track of the cardinalities of each worker. Our experiments indicated that when the number of groups is large, our algorithms are significantly faster compared to both the state of the art (PK) and previously proposed solutions. We conclude that selecting a partitioning algorithm for a *stateful* operation has to be made after considering the expected number of groups, and when that number is large, our proposed algorithms offer the best performance.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful feedback and insightful comments. Furthermore, we thank Cory Thoma, Nicholas Farnan, and Briand Djoko for their feedback on earlier versions of this work. The experimental evaluation was supported by the AWS Cloud Credits for Research program.

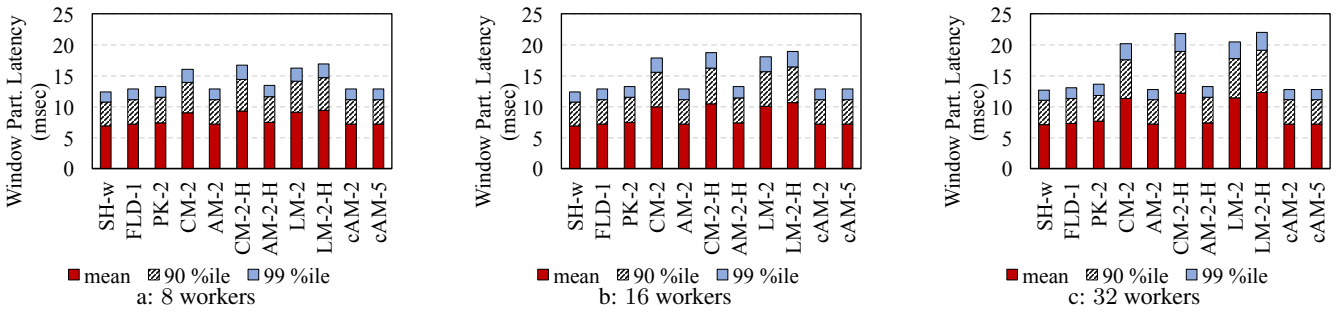


Figure 13: Per window Partition latency for DEBS Query 1 (i.e., processing cost of partitioning algorithm).

11. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, et al. Aurora: A new model and architecture for data stream management. *VLDBJ*, 12(2):120–139, 2003.

[3] T. Akidau, R. Bradshaw, C. Chambers, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *PVLDB*, pages 1792–1803, 2015.

[4] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *VLDBJ*, 15(2):121–142, 2006.

[5] Y. Azar, A. Z. Broder, et al. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[6] B. Babcock, S. Babu, M. Datar, et al. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[7] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[8] P. Carbone, S. Ewen, S. Haridi, et al. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 2015.

[9] R. Castro Fernandez, M. Migliavacca, et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736, 2013.

[10] B. Chandramouli, J. Goldstein, M. Barnett, et al. Trill: A high-performance incre- mental query processor for diverse analytics. In *PVLDB*, pages 401–412, 2015.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[12] G. J. Chen, J. L. Wiener, S. Iyer, et al. Realtime data processing at facebook. In *SIGMOD*, pages 1087–1098, 2016.

[13] M. Elseidy, A. Elguindy, V. A., and C. Koch. Scalable and adaptive online joins. In *PVLDB*, pages 441–452, 2014.

[14] P. Flajolet et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[15] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDBJ*, 23(4):517–539, 2014.

[16] V. Gulisano et al. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 23(12):2351–2365, 2012.

[17] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *DEBS*, pages 318–321, 2014.

[18] T. Heinze, L. Roediger, A. Meister, et al. Online parameter optimization for elastic data stream processing. In *SoCC*, pages 276–287, 2015.

[19] T. Heinze, M. Zia, R. Krahn, et al. An adaptive replication scheme for elastic data stream processing systems. In *DEBS*, pages 150–161, 2015.

[20] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, pages 683–692, 2013.

[21] Z. Jerzak and H. Ziekow. The debs 2015 grand challenge. In *DEBS*, pages 266–268, 2015.

[22] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch. Themis: Fairness in federated stream processing under overload. In *SIGMOD*, pages 541–553, 2016.

[23] N. R. Katsipoulakis, C. Thoma, E. A. Gratta, et al. Ce-storm: Confidential elastic processing of data streams. In *SIGMOD*, pages 859–864, 2015.

[24] A. Koliousis, M. Weidlich, R. Castro Fernandez, et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, pages 555–569, 2016.

[25] S. Kulkarni, N. Bhagat, M. Fu, et al. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.

[26] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, pages 811–825, 2015.

[27] D. G. Murray, F. McSherry, R. Isaacs, et al. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[28] M. A. Nasir, G. Morales, D. Garcia-Sorano, et al. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, pages 137–148, 2015.

[29] M. A. Nasir, G. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *ICDE*, pages 589–600, 2016.

[30] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *SIGMOD*, pages 511–526, 2016.

[31] N. Rivetti, E. Anceaume, Y. Busnel, et al. Online scheduling for shuffle grouping in distributed stream processing systems. In *Middleware*, pages 11:1–11:12, 2016.

[32] M. Shah, M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[33] A. Shein, P. Chrysanthis, and A. Labrinidis. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*, pages 5:1–5:12, 2017.

[34] A. Toshniwal, S. Taneja, A. Shukla, et al. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[35] Y. Wu and K. L. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, pages 723–734, 2015.

[36] Y. Xing, S. Zdonik, and J. H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[37] M. Zaharia, T. Das, H. Li, et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[38] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.