Prompt: Dynamic Data-Partitioning for Distributed Micro-batch Stream Processing Systems

Ahmed S. Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, Walid G. Aref
Department of Computer Science, Purdue University
West Lafayette, Indiana
{samy,amahmoo,anas,aref}@purdue.edu

ABSTRACT

Advances in real-world applications require high-throughput processing over large data streams. Micro-batching has been proposed to support the needs of these applications. In micro-batching, the processing and batching of the data are interleaved, where the incoming data tuples are first buffered as data blocks, and then are processed collectively using parallel function constructs (e.g., Map-Reduce). The size of a micro-batch is set to guarantee a certain response-time latency that is to conform to the application's service-level agreement. In contrast to tuple-at-a-time data stream processing, micro-batching has the potential to sustain higher data rates. However, existing micro-batch stream processing systems use basic data-partitioning techniques that do not account for data skew and variable data rates. Load-awareness is necessary to maintain performance and to enhance resource utilization. A new data partitioning scheme, termed *Prompt* is presented that leverages the characteristics of the micro-batch processing model. In the batching phase, a frequency-aware buffering mechanism is introduced that progressively maintains run-time statistics, and provides online *key-based* sorting as data tuples arrive. Because achieving optimal data partitioning is NP-Hard in this context, a workload-aware greedy algorithm is introduced that partitions the buffered data tuples efficiently for the Map stage. In the processing phase, a load-aware distribution mechanism is presented that balances the size of the input to the Reduce stage without incurring inter-task communication overhead. Moreover, Prompt elastically adapts resource consumption according to workload changes. Experimental results using real and synthetic data sets demonstrate that Prompt is robust against fluctuations in data distribution and arrival rates. Furthermore, Prompt achieves up to 200% improvement in system throughput over state-of-the-art techniques without degradation in latency.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389713

CCS CONCEPTS

• **Information systems → Data streams**; **MapReduce-based systems**; **Stream management**; *Record and block layout*; *Query optimization*; *Main memory engines.*

KEYWORDS

Distributed Data Processing, Micro-batch Stream Processing, Data Partitioning, Elastic Stream Processing

ACM Reference Format:
Ahmed S. Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, Walid G. Aref. 2020. Prompt: Dynamic Data-Partitioning for Distributed Micro-batch Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389713

1 INTRODUCTION

The importance of real-time processing of large data streams has resulted in a plethora of Distributed Stream Processing Systems (DSPS, for short). Examples of real-time applications include social-network analysis, ad-targeting, and clickstream analysis. Recently, several DSPSs have adopted a batch-at-a-time processing model to improve the processing throughput (e.g., as in Spark Streaming [43], M3 [4], Comet [21], and Google DataFlow [2]). These DSPSs, often referred to as *micro-batch* stream processing systems, offer several advantages over continuous tuple-at-a-time DSPSs. Advantages include the ability to process data at higher rates [42], efficient fault-tolerance [43], and seamless integration with offline data processing [5]. However, the performance of micro-batch DSPSs is highly susceptible to the dynamic changes in workload characteristics. For example, resource utilization strongly relies on evenly partitioning the workload over the processing units. Moreover, the computational model is inherently subject to performance instability

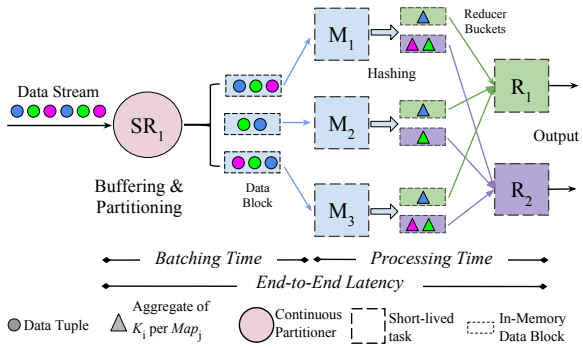


Figure 1: Example of micro-batch stream processing: The computation has 3 Map and 2 Reduce tasks, and a Stream Receiver (SR_1) that provides micro-batches from the input data stream.

with the fluctuations in arrival rates and data distributions. To illustrate, consider a streaming query that counts users' clicks per country for a web advertising-campaign every 30 minutes. When applying the micro-batch processing model to this query, the data flow is divided into two consecutive phases: *batching* and *processing* (See Figure 1). Stream processing is achieved by repeating the *batching* and the *processing* phases for the new data tuples. The two phases are overlapped for any two consecutive batches (Refer to Figure 2). In the *batching* phase, the stream data tuples are accumulated for a predetermined batch interval. Then, the batch content is partitioned, and is emitted in the form of data blocks for parallel processing. In the *processing* phase, the query is executed in memory as a pipeline of Map and Reduce stages. In the Map stage, a user-defined function is applied in parallel to every data block (e.g., a filter over the clickstream tuples). Then, a Reduce stage aggregates the outcome of the Map stage to produce the output of the batch (e.g., sums the clicks for each country). Finally, the query answer is computed by aggregating the output of all batches that are within the query's time-window. The end-to-end latency is defined at the granularity of a batch as the sum of the *batch interval* and the *processing time*. The system is stable as long as $processing\ time \leq batch\ interval$. A stable system prevents the queuing of batches and provides an end-to-end latency guarantee. In the rest of this paper, the terms *micro-batch* and *batch* are used interchangeably to define a buffered set of stream-tuples over a predefined small interval of time (i.e., the batch interval).

In this paper, we address two challenges, namely data partitioning and performance stability:

Data Partitioning: Data partitioning is crucial to the batching and the processing phases. First, the *execution time* of all Map tasks within a micro-batch needs to be nearly equal. A Map task that lags due to extra load can severely impact resource utilization as it blocks the scheduling of the subsequent Reduce stage, e.g., as in Cases II, III and IV of Figure 2.

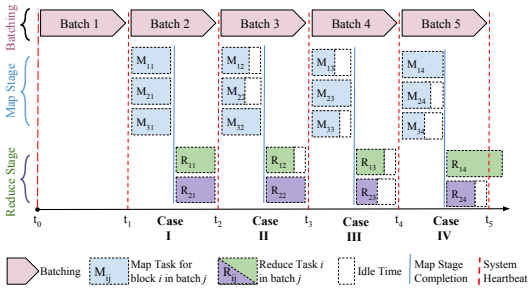


Figure 2: The effect of data partitioning on the pipelined-execution over micro-batches: An example timeline that illustrates three different cases of unbalanced-load execution in $[t_0\text{-}t_5]$. Notice that Batch x is processed concurrently while Batch x+1 is being accumulated.

Similarly, the input to all Reduce tasks should be evenly distributed to finish at the same time. The system may become unstable as batches get queued. The end-to-end latency would also increase. Second, the time to process a batch may exceed the batch interval time due to the lack of even data distribution in the Map and/or the Reduce stages leading to mistakenly requesting additional resources. Case IV in Figure 2 can be avoided by adequately partitioning the data load at the Map and Reduce stages. The data partitioning problem in the micro-batch stream processing model is challenging for the following reasons: **(1)** The partitioning decision needs to happen as fast as the data arrive. The reason is that the processing starts as soon as the batching is completed. Applying a basic partitioning algorithm, e.g., round-robin, leads to an uneven distribution of the workload. **(2)** The data partitioning problem is inherently complex. It entails many optimization factors, e.g., key locality, where tuples with the same key values need to be co-located into the same data blocks. In fact, as will be explained in Section 4.2, this data partitioning problem is NP-hard. **(3)** In a Map-Reduce computation, intermediate results of a key must be sent to the same Reducer. Fixed key-assignment with hashing does not guarantee balanced load at all Reducers. Also, dynamically assigning keys to the Reducers by globally coordinating among Map tasks is time-consuming due to the inter-communication cost. Thus, it is not suitable for streaming applications.

Performance Stability: The healthy relationship between processing time and batch interval is mandatory to keep the system stable and achieve latency guarantees. If the processing time exceeds the batch interval while applying even-data partitioning, then additional resources must be warranted to maintain system performance. However, resource allocation in micro-batch DSPSs is challenging for two reasons: (1) Manually tuning the resources is an error-prone and complicated task. The reason is that, in contrast to offline data processing, data streams are dynamic, and can change data rate or distribution at runtime [35, 36, 41, 43]. To maintain its performance, micro-batch DSPSs need to automatically scale

in and out to react to workload changes. (2) Permanent resource over-provisioning for peak loads is not a cost-effective strategy and leads to a waste in resources.

Previous work on improving the performance of micro-batch DSPSs focuses on resizing the batch interval [12, 45]. The batch interval is *resized* to maintain an equal relationship between the processing and batching times. However, batch resizing does not solve the resource utilization problem targeted in this paper, and may lead to delays in result delivery, e.g., when the resized batch interval violates the application's latency requirements. In this paper, we focus on achieving latency guarantees while minimizing the resource consumption of micro-batch DSPSs.

In Prompt, we introduce a new data partitioning scheme that optimizes the performance of the micro-batch processing model. In the batching phase, Prompt has a new load-aware buffering technique that constructs, at runtime, a sorted list of the frequencies of the occurrences of keys in the current batch. Prompt partitions the batch content in a workload-aware manner for the upcoming Map stage. To prepare partitions for the Reduce stage, Prompt has an effective distribution mechanism that allows the Map tasks to make local decisions about placing intermediate results into the Reduce buckets. This partitioning mechanism balances the input to the Reduce tasks, and avoids expensive global partitioning decisions. To account for workload changes and maintain performance, Prompt employs a threshold-based elasticity technique to swiftly adjust the degree of parallelism at run-time according to workload needs. Prompt does not require workload-specific knowledge, and is agnostic to changes in data distribution and arrival rates. The main contributions of this paper can be summarized as follows:

- We formulate the problem of data partitioning in distributed micro-batch stream processing systems. We show that this problem in both the batching and processing phases is NP-hard. We reduce the data partitioning problems in the batching and processing phases to two new variants of the classical *Bin Packing* problem.
- We introduce Prompt, a data partitioning scheme tailored to distributed micro-batch stream processing systems. Prompt elastically adjusts the degree of execution parallelism according to workload needs. Prompt is robust to fluctuations in data distribution and arrival rates.
- We realize Prompt in Spark Streaming [1], and conduct an extensive experimental evaluation using Amazon EC2. Prompt improves system throughput by up to 2x using real and synthetic datasets over state-of-the-art techniques.

The rest of this paper proceeds as follows. Section 2 presents background on the distributed micro-batch stream processing model and the existing data partitioning techniques. Section 3 highlights the objectives of Prompt along with the problem formalization and the underlying cost model. The partitioning techniques for batching and processing in Prompt are presented in Sections 4 and 5, respectively. Section 6 illustrates resource elasticity in Prompt. Performance evaluation is detailed in Section 7. Related work is in Section 9. Finally, Section 10 contains concluding remarks.

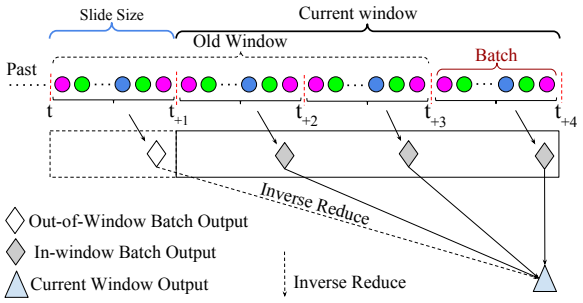


Figure 3: Micro-batch Stream Processing

2 BACKGROUND

2.1 Processing Model

The distributed micro-batch stream processing model executes a continuous query in a series of consecutive, independent, and stateless Map-Reduce computations over small batches of streamed data to resemble continuous processing. System-wide heartbeats, i.e., triggers, are used to define the boundaries of the individual batches. Dedicated processes are responsible for continuously receiving stream data tuples and for emitting a micro-batch at every heartbeat.

Every micro-batch is partitioned according to the supported level of parallelism by the computing resources, i.e., the number of processing cores. We term every partition a *data block*. The input data S is an infinite stream of tuples. Each tuple $t = (t_s, k, v)$ has a three-component schema; a timestamp t_s set by the stream's originating source, a key k, and a value v. Keys are not unique, and are used to partition the tuples for distributed processing purposes. The value v can be single or multiple data fields. We make the following assumptions: (1) Tuples arrive in sorted timestamp order, and (2) The delay between the timestamp of a tuple and its ingestion time cannot exceed a maximum delay. A streaming query Q submitted in a declarative or imperative form is compiled into a Map-Reduce execution graph (see Figure 1). The Map stage is defined over data with (k, v) pairs as $Map(k_i, v_1) \rightarrow (k_i, List(V))$. The Reduce stage follows, and uses the output of the Map stage to provide the final output as $Reduce(k_i, list(v2)) \rightarrow list(V)$.

The execution graph shows the physical details of the execution, e.g., the level of parallelism, i.e., the number of Map and Reduce tasks, data partitioning, the order of task execution, and the data dependency among the tasks. The execution graph applies over each batch to compute a partial output that is preserved as the query state. In contrast to

tuple-at-a-time systems, the query state is decoupled from the execution tasks. The streaming query Q can be defined over a sliding or tumbling window. The query answer is computed by aggregating the output of all batches that reside within the query window (Figure 3). To avoid redundant recalculations, the micro-batches that exit the window are reflected incrementally onto the query answer by applying an inverse Reduce function (e.g., [43]).

2.2 Existing Data Partitioning Techniques

The most popular partitioning techniques are *time-based* [43], *hashing*, and *shuffling*. Then, we describe the state-of-the-art data partitioning in continuous tuple-at-a-time stream processing, namely, a technique referred to as *key-splitting* [6] partitioning, and discuss the feasibility of adopting it in the micro-batch stream processing model.

2.2.1 Time-based Partitioning. Time-based partitioning uses the arrival time of a new tuple to assign the tuple into a *data block* (Figure 4a). Given the target *number* of data blocks, the *batch interval* is split into consecutive, non-overlapping, and equal-length time periods, denoted by *block interval*. All the data tuples received during each period constitute a data block. When the batch interval finishes, all the data blocks are wrapped as a batch, and become available for processing. This simple partitioning technique has the following limitation. Time-based partitioning results in unequally sized data blocks due to dynamic input data rates. Moreover, time-based partitioning does not have any guarantees on key placement. Data tuples that share the same key value can end up in different data blocks. This can increase the cost of the per-key-aggregation step at the Reduce stage.

2.2.2 Shuffle Partitioning. Shuffle partitioning assigns tuples to data blocks in a round-robin fashion based on arrival order without considering other factors (Figure 4b). This technique guarantees that all the data blocks have the same size even with dynamic input data rates. However, this technique has a major drawback. It does not ensure *key locality*, i.e., tuples with the same key are not necessarily co-located into the same data blocks. This leads to further overhead at the Reduce stage to combine all the intermediate results of each key produced by different Map tasks.

2.2.3 Hash Partitioning. Hash partitioning, also termed *Key Grouping* [36], uses one or more particular fields of each tuple, i.e., a *partitioning key*, and uses a hash function to assign the tuple into a *data block* (Figure 4c). Hence, all the tuples with the same keys are assigned to the same data blocks. Applying this technique in the batching phase eliminates the *per key* aggregation at the Reduce stage.

If the input data stream is skewed, then some key values will appear more often than others. Thus, this partitioning technique would result in unequally sized data blocks (See Figure 4c). Moreover, in the processing phase, Map tasks use the hashing technique to ascertain that all the intermediate results for a key are at the same Reduce task. In the case of data skew, this technique will result in uneven input sizes for the various Reduce tasks.

2.2.4 Key-Split Partitioning. The state-of-the-art in stream data partitioning achieves the benefits of both the shuffling and the hashing techniques by splitting the skewed keys over multiple data blocks. The partitioner applies multiple hash functions to the tuple's *partitioning key* to generate multiple candidate assignments. Then, the partitioner selects the data block with the least number of tuples at the time of the decision [25, 35, 36]. The partitioner keeps track of the following parameters in an online fashion; (1) The number of tuples in each data block, and (2) Statistics on the data distribution to detect the skewed keys in order to split them. Due to the continuity of execution in native DSPSs, these techniques are obliged to make a per-tuple decision upon tuple arrival. Otherwise, stream processing will be interrupted. Micro-batch stream processing systems offer the opportunity to optimize the partitioning of the entire batch. Instead of relying on approximate data statistics to detect key-skewness, it can be accurately evaluated for the entire batch.

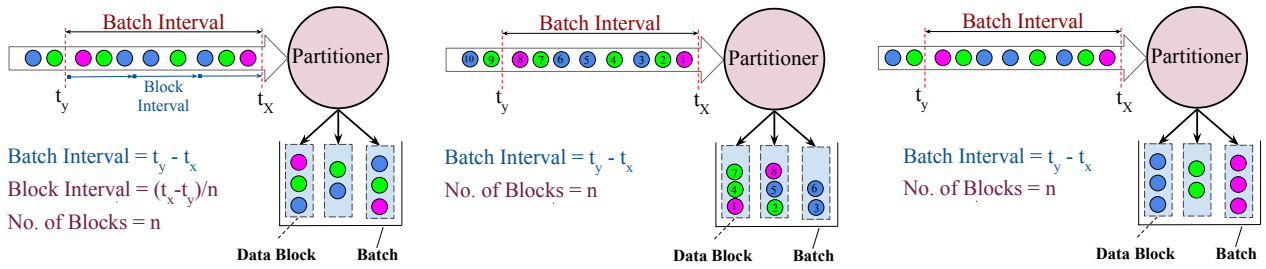
3 DATA PARTITIONING

The performance of micro-batch DSPSs depends heavily on the data partitioning technique adopted. In this section, we describe the desirable properties of efficient data partitioning techniques to improve system performance. We formalize the data partitioning problem, and devise a cost model that captures the processing of a micro-batch.

3.1 Design Goals

The main goal of Prompt's data partitioning scheme is to maximize the overall system throughput under the following constraints: (1) The *batch interval* is fixed, and is set as a system parameter to meet an end-to-end latency requirement of the user's application. (2) The computing resources are available on-demand, i.e., the number of nodes and cores available for processing can be adjusted during processing. The highest throughput is defined as the maximum data ingestion rate the system can sustain using the allocated computing resources without increasing the end-to-end latency, i.e., having batches waiting in a queue. The latency is maintained by keeping the processing time bounded by the batch interval. To illustrate, the processing time of n Map and m Reduce tasks can be modeled using the following equation that represents the sum of the maximum duration of any Map task and the maximum duration of any Reduce task:

$$\max_{1 \le i \le n} MapTaskTime_i + \max_{1 \le j \le m} ReduceTaskTime_j \quad (1)$$



(a) Time-based Partitioning **(b) Shuffle Partitioning** **(c) Hash Partitioning**

Figure 4: Existing Data partitioning Techniques

As the maximum *MapTask* time decreases, the Map stage completes faster. The same applies to the Reduce stage. Prompt adaptively balances the load as evenly as possible at the *Map* and *Reduce* stages without increasing the aggregation costs. Utilizing all available resources at processing time is key to maximizing the data ingestion rate, e.g., Case I in Figure 2. Similarly, resource usage should be adequate to workload needs to minimize cost. Extra resources should be relinquished when possible, e.g., Case III in Figure 2 could have been executed with less resources. The opposite is also true. When all the current resources are saturated and run at maximum capacity, if the workload increases further, then we need to elastically increase the resources during runtime. Moreover, the proposed scheme should incur minimal architectural intervention to the stream processing engine, and should not disrupt the developer programming interfaces APIs so that no modifications are required to the users' existing programs.

3.2 Problem Formulation

In the batching phase, the input tuples received during a specified time interval constitute a batch. This batch is partitioned into several data blocks to serve as input to the Map stage. The partitioning algorithm aims to balance the load at the Map stage by providing equal-load data blocks for the Map tasks. The partitioning algorithm uses 3 key aspects to guide the partitioning process that are defined as follows.

Problem I: Map-Input Partitioning: Given a finite set of data tuples with known schema $<k, t, v>$, with k being the partitioning key, and a fixed number of output partitions p, i.e., blocks, it is required to assign each data tuple to one partition while satisfying the following objectives: **(1) Block-size equality**: The execution time of a Map task increases monotonically with its input block size. Having equal data block sizes to all Mappers decreases the variance in execution time for the Map tasks. **(2) Cardinality balance**: Each data block is assigned an equal number of distinct keys. This requirement serves two purposes. First, it enables the Map tasks to generate equal-sized Reduce buckets. Second, it balances the computation overheads among the Map tasks. **(3) Key locality**: Each key is either assigned to one block, or splits over a minimal number of blocks. This requirement limits the *per-key aggregation* overhead at the Reduce stage.

Once a Map task completes, it assigns its output to a number of Reduce buckets (that correspond to the Reduce tasks). The input for each Reduce task is the union of its designated buckets from all Map tasks. At this point, the partitioning algorithm aims to provide an even-load input for each Reduce task. We define this problem as follows.

Problem II: Reduce-Input Partitioning: Given a finite number of data elements in the form of $<k,list(v)>$, and a defined number of Reduce buckets, it is required to assign the data elements to buckets such that: **(1) Bucket-size equality**: Buckets need to be equal in size. The execution time of a Reduce task increases monotonically with the bucket size. Equal-sized input to all Reducers minimizes the variation in execution latencies among all reducers. **(2) Key Locality**: Data tuples having the same key must be in the same Reduce bucket by all Map tasks. This is vital to maintain the correct computational behavior, i.e., each key is aggregated by a single Reduce task.

3.3 Cost Model

We introduce the cost model for data partitioning that captures the problem formulation in Section 3.2. Notice the positive correlation between the size of a partition and the execution time of the task responsible to process it. This applies to both the Map and Reduce stages. Inspired by the work in [35], we define the *Block Size-Imbalance* metric(*BSI*, for short) over a micro-batch at the granularity of a data block or a Reduce bucket:

$$BSI(Blocks) = max_i|Block_i| - avg_i|Block_i| \quad i \in p \qquad (2)$$

Eqn. 2 defines the size imbalance metric as the difference between the maximum block size and the average size of all data blocks, where p is the number of data blocks. Similarly, Eqn. 3 models the size imbalance at the Reduce stage for the buckets, where r is the number of Reduce buckets:

$$BSI(Buckets) = max_j|Bucket_j| - avg_j|Bucket_j| \quad j \in r \qquad (3)$$

Eqn. 4 defines the *Block Cardinality-Imbalance* (BCI, for short) as the difference between the block with maximum key cardinality and the average key cardinality of all blocks [25].

$$BCI(Blocks) = max_i||Block|| - avg_i||Block)|| \quad i \in p \qquad (4)$$

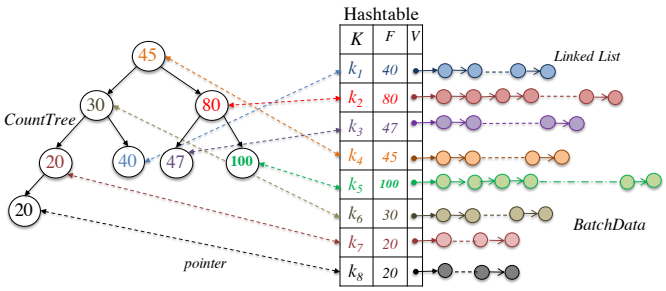


Figure 5: Frequency-aware Micro-batch Buffering: Fully-updated *CountTree* and *HTable* after receiving 385 tuples with 8 distinct keys.

Let a *key fragment* be a collection of tuples that share the same key value. Eqn. 5 defines the *Key Split Ratio* metric (KSR, for short) as the ratio between the total number of distinct keys in a batch and the number of key fragments on all data blocks. If no keys are split, then KSR=1.

$$KSR(Blocks) = \frac{Sum_k|Fragments|}{Sum_k|Keys|} \; k \in K \tag{5}$$

Finally, we define the overall *Partitioning-Imbalance* metric [25] over a *Micro-batch* (MPI, for short) using a combination of the above metrics as follows:

$$MPI(Blocks) = p_1 * BSI + p_2 * BCI + p_3 * KSR \tag{6}$$

Notice that the objective is to minimize the three metrics which lead to minimize MPI. We provide a mathematical formulation for the problem in Section 4.2. The parameters p_1, p_2, p_3 are adjustable to control the contribution of each metric (i.e., $p_1 + p_2 + p_3 = 1$). In our experiments, we set $p_1 = p_2 = p_3 = 1/3$ to achieve unbiased and equal contribution from all the metrics (i.e., avoiding one metric dominating the others). Setting $p_1 = 1$ represents the shuffling partitioning behavior, while setting $p_3 = 1$ represents the hashing partitioning behavior.

4 MICRO-BATCH DATA PARTITIONING

We introduce Prompt's partitioning technique for the batching phase in the micro-batch stream processing model. This partitioning technique has 2 main steps: (1) The input data tuples are buffered while statistics are collected as the tuples arrive. (2) The partitioning algorithm is applied over the micro-batch to generate data blocks for the processing phase. The following subsections explain these 2 steps.

4.1 Frequency-aware Buffering

As in Section 2.1, the micro-batch is to be processed at the end of the batch interval, i.e., at the system heartbeat. To minimize the time required to prepare the micro-batch for partitioning, two data structures are used to maintain runtime statistics of the data tuples as they arrive. We utilize a *hash table* and a *Balanced Binary Search Tree (BBST)* as follows: The partitioning key of the incoming data tuples is used to store the tuples into the hash table $HTable <K, V>$, where the value part is a pointer to the list of tuples for every key. Also, *HTable* stores auxiliary statistics for each key, e.g., frequency count and other parameters that are utilized in the following update mechanism. In addition, approximate frequency counts of the keys are kept in a balanced binary search tree *CountTree*. Every key in *HTable* has a bi-directional pointer to a designated counting node in *CountTree*. This pointer allows to directly update the count node of a key. For illustration, Figure 5 gives a simple example for *HTable* and a fully-updated *CountTree* after receiving 385 tuples with 8 distinct keys. Notice that the typical size of a micro-batch can grow up to millions of tuples with 10-100ks of distinct keys. To handle high data rates, a coarse-grained approach to update *CountTree* is used. Instead of updating the *CountTree* for each incoming tuple, each key is allowed to update the *CountTree* periodically for a maximum of *budget* times in a batch interval. A control parameter, $f.step$, is defined, where a node is updated once for every $f.step$ new tuples received of its key. The $f.step$ parameter is estimated adaptively for each key based on the proportional of the current key frequency to the total number of tuples received since the beginning of the current batch interval (i.e., keys with high frequency will require more data tuples to trigger an update). Furthermore, to ensure that nodes for tuples with low frequency get updated, a time-based $t.step$ is also used. Similarly, $t.step$ is estimated based on how much of the key's *budget* updates are consumed and the remaining duration of the batch interval. An update is triggered when an incoming tuple satisfies the time or frequency step for its key. Initially, $f.step$ is set to some constant f that reflects the best step value if the data is assumed to be uniformly distributed. $f \longleftarrow \frac{N_{Est}}{K_{Avg} * Budget}$, where N_{Est} is the estimated number of tuples given the average data rate and batch interval, and K_{avg} is the average number of distinct keys over the past few batches. Notice that $f.step$ quickly converges to the proper value that suits the current batch. Algorithm 1 lists the buffering technique used in the batching phase. This updating mechanism avoids thrashing *CountTree* with re-balancing operations, and bounds the complexity of all updates to $Klog(K)$, where K is the total number of distinct keys received in a micro-batch. This is comparable to the complexity of sorting keys after the batch interval has ended. However, this updating mechanism takes place during the batching phase, and hence does not require explicit sorting before the start of the processing phase. A dedicated sorting step would have consumed a portion of the time available for processing the batch. At the end of every batch interval, an in-order traversal of the *CountTree* generates a quasi-sorted list of the keys with their associated frequencies. The sorted list: $<k_i, count_i, tupleList_i>$ is used

as the input to the partitioning algorithm. The *HTable* and the *CountTree* data structures are both cleared at the end of every batch interval, i.e., system heartbeat.

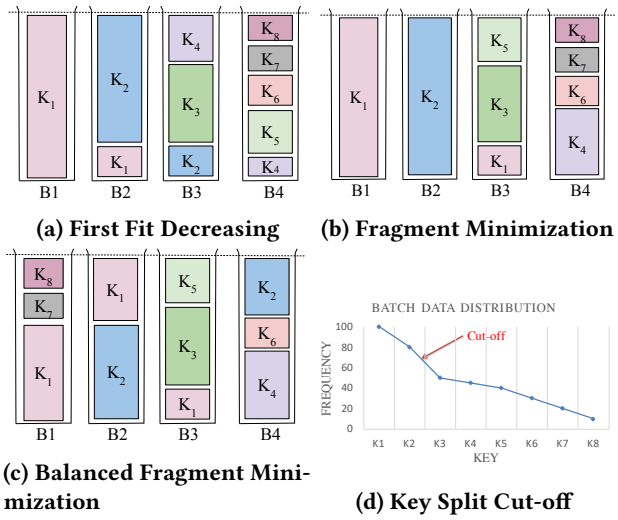
Algorithm 1: Micro-batch Accumulator

Input: S: Input Stream, $[t_{start}\text{-}t_{end}]$: Batch Interval, *budget*: Update Allowance, f: Initial Frequency Step
Output: *Stat*: Batch Statistics, e.g., NC: Number of data tuples, $|K|$: Number of keys, *Batch*: SortedList<k_i,$count_i$,$tupleList_i$>

```
1  Reset HTable, and CountTree;
2  while tuple_i.ts ∈ Batch Interval do
3      Increment number of tuples Count: N_C;
4      if tuple_i.k ∈ HTable then
5          Insert tuple_i into HTable_k chain;
6          Update k.Freq_Current;
7          Delta_freq = k.Freq_Current - k.Freq_Updated;
8          Delta_time = Time_Now - k_LastUpdateTime;
9          if k_f.step == Delta_freq then
10             Update k_freq in CountTree;
11             Update k.budget = k.budget -1 ;
12             Update k.Freq_Updated;
13             Set k_f.step =
                 (N_EST/budget)*k.Freq_Current/N_C;
14         else
15             if k_t.step == Delta_time then
16                 Update k_freq in CountTree;
17                 Update k.budget = k.budget -1 ;
18                 Update k.Freq_Updated;
19                 Set k_t.step = (t_end-Now_Time)/k.budget;
20             else
21                 k is not eligible for an update yet;
22             end
23         end
24     else
25         Increment Unique Keys Count: |K|;
26         Insert tuple_i into HTable;
27         Insert tuple_i.k as new node into CountTree;
28         Initialize k.Freq_Current and k.Freq_Updated to 1;
29         Initialize k_t.step = (t_end - Time_Now)/budget;
30         Initialize k_f.step = f;
31     end
32 end
```

4.2 Load-Balanced Batch Partitioning

In this section, we discuss how Prompt handles the *Batch Partitioning* problem. The problem is reduced to a new variant of the classical bin-packing problem. All data tuples that shares the same key value are modeled as a single item, whereas each data block is modeled as a bin. Each bin has a capacity that corresponds to the expected size of the data block. In contrast, each item has a distinct size equal to the number of tuples that shares same key value. We refer to this problem as the *Balanced Bin Packing with Fragmentable Items (B-BPFI)*. An item is *fragmented* if it is split into two or more sub-items such that the sum of the sizes of all sub-items is equal to the initial size of the item before splitting. In this case, the newly split items with the same key value can be stored in different bins (i.e., in different data blocks). In this instance of the bin packing problem, the number of bins is known a priori, and all bins have equal capacities. In addition, the items are allowed to be fragmented such that the number of distinct items per bins are equal. Hence, the solution to the problem is to find a good assignment of the items to the bins that satisfies the three objectives captured by the cost model in Eq 6, mainly, (1) Limit the fragmentation of the items, (2) Minimize the cardinality variance among the bins, and (3) Maintain the size balanced among the bins. Notice that achieving the three objectives is challenging. For instance, Figures 6a and 6b give two possible assignments into four data blocks ($B1$, $B2$, $B3$ and $B4$) for the batched data in Figure 5. In both assignments, the data blocks are of equal size. However, the well known First-Fit-Decreasing technique [33], illustrated in Figure 6a, does not minimize item fragmentation. This results in fragmenting 3 out of the 8 keys, namely, K_1, K_2, and K_4. In contrast, in Figure 6b, the use of the Fragmentation Minimization technique [24] limits the fragmentation to only one key (K_1). Both assignments fail to meet Objective 2; the number of items in $B4$ is twice the number of items in the other blocks. The *B-BPFI* problem can be formally defined as follows:



(a) First Fit Decreasing (b) Fragment Minimization

(c) Balanced Fragment Minimization (d) Key Split Cut-off

Figure 6: Assignment Trade-offs for the Bin Packing with Fragmentable Items problem

Definition 1. *Balanced Bin Packing with Fragmentable Items. Given a set of K distinct items: k_1, k_2, $\cdots$, k_K; each with Item Size s_i, where $1 \leq i \leq K$, and B bins, each with Bin Capacity C, the Balanced Bin Packing with Fragmentable*

Items (B-BPFI) is to generate item assignments to bins: b_1, b_2, $\cdots$, b_B that satisfy the following 3 requirements: (1) $|b_j|$=C $\forall j \in [1, B]$, where $|b_j|$ denotes the number of tuples in b_j; (2) $||b_j|| \geq$ K/B, where $||b_i||$ denotes the number of unique items in b_j $\forall j \in [1, B]$; (3) For any item ∈ K, it is split over the minimum number of bins.

The problem can be formulated mathematically as follows.

$$Minimize \quad (\sum_{i=1}^{K}\sum_{j=1}^{B} y_{ij}) \tag{7}$$

so that:

$$\sum_{j=1}^{B} x_{ij} = s_i \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{8}$$

$$\sum_{i=1}^{K} x_{ij} \leq c_j \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{9}$$

$$\frac{x_{ij}}{s_i} \leq y_{ij} \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{10}$$

$$\frac{K}{B} \leq \sum_{i=1}^{K}\sum_{j=1}^{K} y_{ij} \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{11}$$

$$x_{ij} \in N^+, y_{ij} \in 0, 1 \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{12}$$

The variable x_{ij} represents the size of Item i's fragment that is placed in Bin j. x_{ij} must be an integer (included between 0 and s_i) (see Eqn. 8). Eqn. 8 implies that the sum of the sizes of the fragments of any item i must be equal to its total size si. Eqn. 9 restricts the sum of the sizes of the fragments put in any bin j to not exceed the capacity c_j of this bin. y_{ij} is a bivalent variable equal to 0 or 1 (see Eqn. 12) to mark the presence of Item i in Bin j. As soon as a part of Item i is present in Bin j (even a very small part), y_{ij} is equal to 1. Otherwise, y_{ij} is equal to 0. Eqn 10 forces Variable y_{ij} to be 1 whenever Variable x_{ij} is strictly greater than 0. The sum of all the variables y_{ij} corresponds to the total number of fragments of items. It is this quantity that we want to minimize (Eqn. 7). Without loss of granularity, we assume that the data tuples are of the same size for simplicity. However, our problem formulation can be easily extended to variable tuple sizes. Moreover, the total capacity of the bins is defined to be larger than or equal to the total size of the items.

$$\sum_{i=1}^{K} s_i \leq \sum_{j=1}^{B} c_j \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{13}$$

Theorem 1. *The Balanced Bin Packing with Fragmentable Items problem is NP-Complete.*

Proof. The classical Bin Packing problem is a special case of B-BPFI in which all the bins have the same capacities, the maximum number of fragments per bin is equal to $K - B - 1$ (or 1 in case $B > K$), and the maximum number of fragments allowed is K (i.e., no fragmentation is allowed), and the items can be assigned arbitrarily. Since the bin packing problem is strongly NP-complete, hence the B-BPFI is strongly NP-complete, and the optimization form is at least as hard as the classical bin packing problem. Proof by restriction. □

The classical bin packing problem is a well-known combinatorial optimization problem [40] and has been studied for decades. Some of its forms have even dealt with fragmentable items [9, 10, 16, 23, 24, 29, 33, 37–39]. The available solutions for the bin packing problem fall into two categories: First, they are very customized to the classical bin packing optimization problem, where the objective is to minimize the number of bins required, and hence yields unsatisfactory results in the case of the *B-BPFI* problem (e.g., First Fit Decreasing). The reason is that filling a bin nearly completely is a good result for BP, because minimizing wasted space results in fewer required bins, but it is generally a bad strategy for B-BPFI as it results in plenty of fragmentation and cardinality imbalance. Second, due to the hardness of the problem, the available computational solution algorithms do not scale well. They involve problem instances with no more than 100 items, and may require several minutes or even hours to solve [29]. Our focus here is on finding a heuristic algorithm that produces high-quality partitioning for thousands of items in milliseconds. However, to the best of our knowledge, no algorithms or other heuristic approaches for *B-BPFI* exist in the literature.

Algorithm 2 lists the proposed heuristic to partition the batch of tuples into data blocks, while achieving the three objectives highlighted above. First, the high-frequency keys are detected and are fragmented (See Figure 6d). Any key that has a frequency larger than the ratio of $\frac{Block\ Size}{Block\ Cardinality}$ is split into two fragments. One fragment is placed in a data block, and the other fragment is maintained temporarily in a list. Then, the rest of the keys are assigned to the data blocks in a *zigzag-style*, i.e., the blocks' order is reversed after each pass. Notice that since the keys are quasi-sorted, this has the effect of **BestFitDecreasing** without the need and cost to maintain the block sizes. Finally, the residuals of the high-frequency keys are assigned using **BestFit** while giving priority to key-locality. Figure 6c gives the assignments for the batched data example of Figure 5 into four blocks using Algorithm 2. Notice that this assignment strikes a balance between fragmentation and cardinality while maintaining equal-sized blocks, i.e., only two items are fragmented (K_1 and K_2) and the key-cardinality is almost identical.

To avoid contributing to the processing time, Prompt applies a simple latency-hiding mechanism, termed *Early Batch Release*. The objective of this mechanism is to ensure that the batch is partitioned and ready-for-processing at the heartbeat signal. To achieve this objective, the batching cut-off is separated from the processing cut-off, i.e., the system's heartbeat. The batch content is to be released for partitioning before the expected system heartbeat that originally signals the end of the batch interval (See Figure 7). This allows the partitioner a slack time to execute the partitioning algorithm

Algorithm 2: Micro-Batch Partitioner

Input: $SortedList <k, k_{count}, k_{TupleList}>$: Input Batch, N_c: Number of data tuples, K: Number of distinct keys P: Number of required data partitions.
Output: *Plan*: Optimized keys-to-partitions assignments

```
Define Partition-Size: P_Size=N_C/P;
Define Partition-Cardinality: P_|k| = K/P;
Define Key-Split-CutOff: S_Cut = P_Size/P_|k|;
Set b_i = b_1 (b_1 ∈ Partitions);
while ∃k ∈ List and |k| > S_Cut do
    Put S_Cut fragment → b_i and add residual to RList;
    Update lookupLargePos(k ↔b_i);
    Set b_i to b_{i++%P};
end
while ∃k ∈ List do
    foreach b ∈ Partitions do
        Put one key in b;
        Move to next b;
    end
    Reverse Order of Partitions loop;
end
while ∃k ∈ RList do
    b = lookupLargePos(k);
    if k fits in b then
        Add k to b;
    else
        Fill b from k;
        Add rest of k to partition with lowest remaining
          capacity that can hold it;
    end
end
```

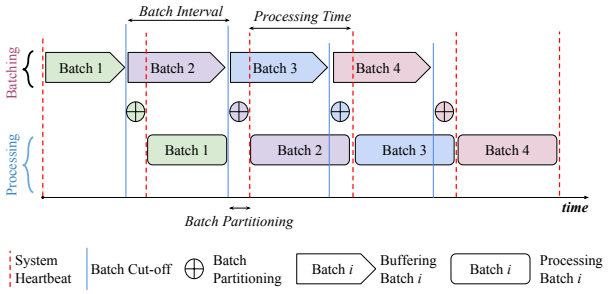
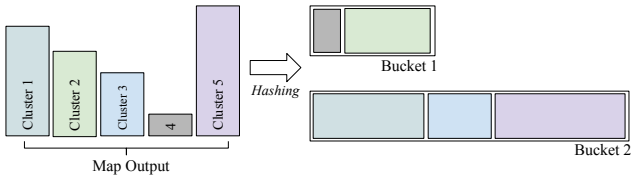
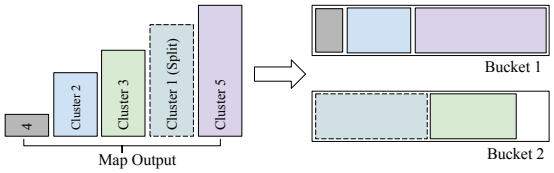


Figure 7: Early Batch Release

on the collected input data, and make batch data ready at the heartbeat pulse. The mechanism is implemented within the batching module, and hence the normal execution of the processing engine is not influenced. In our experiments, we have observed that a maximum of 5% of the batch-interval is sufficient to achieve this objective.



(a) Hashing Assignment



(b) Balanced Assignment

Figure 8: Reduce Replacement Strategies

5 PROCESSING-PHASE PARTITIONING

In this section, we introduce a partitioning technique for the processing phase of the micro-batch stream processing model. In the batching stage, each data *block* is equipped with a reference table. In this table, keys that exist in the data block are labeled to indicate if they are split over other data blocks. Each Map task leverages this information to guide the assignment of the key clusters to its Reduce buckets. Figure 8a gives an example of the default assignment of a Map task output to its Reduce buckets using conventional hashing approach. This method does not consider the key cluster sizes, and that leads to un-balanced input to the Reduce stage. The Map output is key-value pairs grouped into clusters. Each key cluster has all data values with the same key, and can be represented as: $C_k = \{(k, v_i) \mid v_i \in k\}$.

Key clusters can have different sizes. Assume that there are K key clusters in the output of the Map task to be assigned to r Reduce buckets. Let I be the output (i.e., referred to as intermediate results) of the Map task: $I = \{C_k \mid k \in K\}$. To provide a balanced load for the Reduce tasks, an equal assignment to each Reduce bucket should be warranted. The expected size of a bucket can be estimated as: $Bucket\ Size = \frac{|I|}{r}$. We reduce this problem to a new variant of the bin packing problem. All key clusters are items, and the Reduce buckets are bins. However, in contrast to the batch partitioning problem, the bins are of variable capacities, and the items are not fragmentable. Each key cluster is a non-fragmentable item as values of the same key must be at the same Reduce bucket. The bins are of variable capacities because keys split over multiple data blocks are assigned to the Reduce buckets using the hashing method. The Map task has the freedom to assign the non-split keys only. The capacity of each bucket is the residual of the value estimated by *Bucket Size* after assigning the split key clusters using the hashing method. The problem is defined and is proved NP as follows:

Definition 2. *Balanced Bin Packing with Variable Capacity (B-BPVC). Given a set of items, K, and B bins, each with*

Algorithm 3: Reduce Bucket Allocator

Input: C: Key Clusters - Map intermediate results,
Ref: Block Reference Table (Split/NonSplit keys),
R: Set of Reduce buckets.
Output: $Cluster - Bucket$ Assignments

```
Define Bucket_size = |C|/|R|;
Assign SplitKeys to R using Hashing;
Let C= C - SplitKeys;
Sort NonSplit key Clusters in descending order;
while ∃k ∈ C do
    Assign k into Bucket r= Worst-Fit(R) ;
    R = R-r ;
    if !∃r ∈ R then
        Reset R = All Reduce Bucket ;
    else
    end
end
```

associated capacity C_i, the Variable balanced bin packing is to generate item assignments to bins: b_1,b_2,...,b_B, that satisfy three requirements: (1) $|b_i| \leq C_i$ for any $i \in [1,B]$, where $|b_i|$ denotes the number of tuples in b_i ; (2) $||b_i|| \leq K$, where $||b_i||$ denotes the number of unique items in b_i;

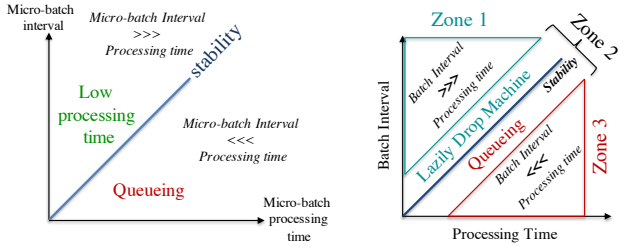
Theorem 2. *The Balanced Bin Packing with Variable Capacity (B-BPVC) problem is NP-Complete.*

Proof. The Bin Packing problem is a special case of B-BPVC. Since the bin packing problem is strongly NP-complete, hence the B-BPVC is strongly NP-complete and the optimization form is at least as hard as the classical bin packing problem. Proof by restriction. □

Alg. 3 lists the proposed technique used by each Map task to assign the key clusters to the Map task's Reduce buckets. Each Map task assigns the split keys using hashing, and sorts its non-split key clusters based on size. Next, the Map task evaluates the capacity of its Reduce buckets, as explained earlier. The Map task uses **WorstFit** to assign bigger key clusters as early as possible to buckets with maximum available capacity. Notice that the selected bucket is removed from the candidate list until all other buckets receive an item. This limits bucket overflow while promoting a balanced number of key clusters per Reduce bucket. Also, no share of information is necessary among the Map tasks. Thus, as each Map task tries to minimize size imbalance, through the additive property, the overall imbalance is reduced.

6 DYNAMIC RESOURCE MANAGEMENT

We introduce Prompt's technique to adaptively adjust the degree of execution parallelism according to workload needs. The objective of this technique is to enforce latency requirements of the users' applications, while maximizing resource utilization. As explained earlier, the execution graph of a streaming query includes the physical details of the execution including the level of parallelism, i.e., the number of Map and Reduce tasks and the data dependency among the tasks (see Figure 1). To enforce latency, Prompt continuously monitors the relationship between the *batch interval* and the *processing time* for the running micro-batches. Figure 9a depicts the possible relationships between the processing time and the batch interval in micro-batch DSPSs. The stability line represents the ideal scenario, when the processing time and the batch interval are equal. For Prompt, the stability line means that the system is meeting the latency requirement with the currently utilized resources, i.e., the degree of parallelism is sufficient to meet the workload. Otherwise, the system is either overloaded or under-utilized. In the former case, overloading leads to queuing of micro-batches that await processing, and the system experiences an increase in latency time. On the latter case, the system meets its latency requirements, but the resources are under-utilized, i.e., the system is idle and is waiting for the next batch to process.



(a) Workload Behavior **(b) Prompt's Zones**

Figure 9: Prompt's Elasticity Zones

Prompt seeks to meet the latency requirement using minimum resources. Figure 9b illustrates how Prompt defines 3 elasticity zones to guide when auto-scale actions should take place. The purpose of Zone 2 is to shield Prompt from sudden workload changes. It can be viewed as an expansion of the stability line. It queues the delayed batches briefly in case of load spikes, and lazily reduces the executing-tasks when the load is reduced. In Zone 1, Prompt can remove some of the Map or Reduce tasks without affecting the latency guarantees. In Zone 3, Prompt must add more resources to restore stability. The objective is to maintain Prompt in Zone 2.

Prompt uses a threshold-based technique to change the level of parallelism at runtime (see Alg. 4). When the ratio $W = \frac{Processing\ Time}{Batch\ Interval}$ exceeds a system-defined threshold (termed $thres$) for d consecutive batches, a scale-out is triggered. Prompt adds Map and/or Reduce tasks to the execution graph according to workload changes. It uses the two statistics *data rate* and *data distribution* in the past d batches to guide the process. The two metrics are computed as part of the Frequency-aware buffering technique (Section 4.1). If both metrics increase, then Map and Reduce tasks are added.

If only the data rate (i.e., the total number of tuples) increases, then Mappers are added. If only the data distribution (i.e., the number of keys) increases, then Reducers are added. The process repeats until $W \leq thres$. When $W \leq thres - step$ is true for d consecutive batches, scale-in is triggered. Prompt removes Map or Reduce tasks from the execution graph by the same criteria for scaling out. A grace period of d batches is used after completing a scale-in or scale-out, where no reverse decision is made.

Algorithm 4: Latency-aware Auto-Scale

Input: $Stats_d$: Processing Time/Batch Interval for previous d batches, K: Current number of keys and $Size$: Current data rate, p: Current number of Map tasks, r: Current number of Reduce tasks, L_{step}: increments of W (10%), L_{thres}: Upper Load Threshold (90%)

Output: $New\ Execution\ Plan: p\ and\ r$

```
Define W_i = |ProcessingTime_i|/|BatchInterval_i|;
Append W_i to Stats_d;
Update data Rate / data distribution into Stats_d;
if W_i > thres then
    if count = d then
        Increment p if data rate increased;
        Increment r if data distribution increased;
        reset count;
else
    increment count ;
end
return Execution Plan;
```

7 EVALUATION

We conduct experiments on 20 nodes in Amazon EC2. Each node has 16 cores and 32GB of RAM. The nodes are connected by 10 Gbps Ethernet and are synchronized with local NTP servers. We realize Prompt's partitioning technique in Apache Spark v2.0.0. The same concepts are applicable to other micro-batch streaming systems that have a similar design principle of block and batch, e.g., M3 [4], Comet [21], and Google Dataflow [2]. Prompt is realized by modifying four components in Spark Streaming [43]. Algorithm 1 in is implemented in a customized receiver. The receiver layer is responsible to ingest data tuples and maintain the two data structures of Algorithm1. Algorithm2 is implemented in the batch module. The batching module is responsible to seal and serialize the data blocks and place them on the memory of the cluster nodes. Algorithm 3 is implemented in the shuffle phase of the mappers. Algorithm 4 is implemented within the scheduler (i.e., Spark Driver). It is responsible for deciding the number of mappers and reducers for each micro-batch computation. The same JVM heap size and garbage collection flags are applied to launch all Spark executor instances.

Datasets' Statistics		
Name.	Size	Cardinality
Tweets	50GB	790k
SynD	40GB	500k-1M
DEBS	32GB	8M
GCM	16GB	600K
TPC-H	100GB	1M

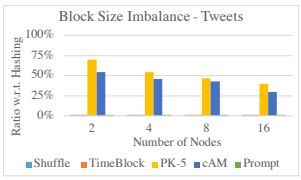
Table 1: Datasets Properties

The queries of the used benchmarks are written as a map-reduce computation. Figure 1 shows the execution graph (i.e., the topology of the computation). The execution graph illustrates the data ingestion within the receiver and processing by mappers and reducers. The window operations are defined over the batch computations similar to Figure 3.

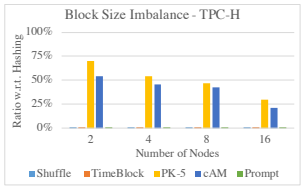
To alleviate the effects of CPU, network, and disk I/O bottlenecks on performance, the following measures are taken: (1) Inverse Reduce functions are implemented for all window queries to account for the expired batches leaving the window span, and hence avoid re-evaluations. (2) Previous in-window batch results are cached in memory to be used in future computations. We ensure that all the window length can fit in memory to avoid spilling to disk. (3) The number of data blocks is bounded by the number of CPU cores on Spark executor instances to avoid any Map task queuing. (4) The system is allowed some time to warm up and stabilize before measuring performance results. Spark Streaming back-pressure is used to indicate when the maximum ingestion rate is reached for every experiment. (5) All the techniques under comparison are assigned the same resources. For PK2 [36],PK5 [35] and cAM [25], the number of candidates per key refers to the maximum number of partitions a key can be assigned to (i.e., the number of hash functions per key). This is different from the total number of workers assigned to the system. For PK2 [36],PK5 [35], the number of candidates per key are fixed at 2 and 5, respectively. For cAM [25], we always report the best performance achieved from several runs with various candidates. For each workload, we increase the number of candidates until performance is stable (i.e., does not improve) or degrades.

7.1 Datasets and Workloads

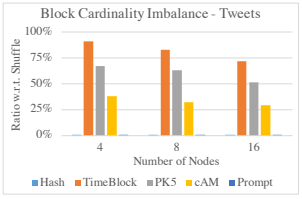
We test the performance of the proposed techniques using various workloads of increasing complexity (refer to Table 1): *WordCount* performs a sliding window count over 30 seconds, and *TopKCount* finds the k most-frequent words over the past 30 seconds. We use the following two datasets, namely Tweet and SynD. Tweet is a real sample of tweets collected in 2015. Each tweet is split into words that are used as the key for the tuple. SynD is a synthetic dataset generated using keys drawn from the Zipf distribution with exponent values $z \in \{0.1, ..., 2.0\}$ and distinct keys up to 10^7. Also, we use the following 3 real and synthetic workloads:



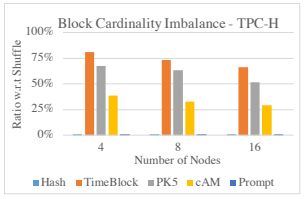
(a) Relative BSI: Tweets



(b) Relative BSI: TPC-H

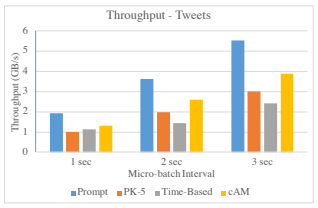


(c) Relative BCI: Tweets

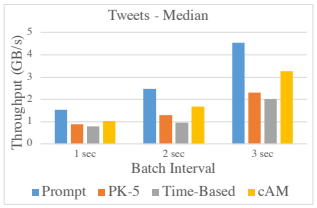


(d) Relative BCI: TPC-H

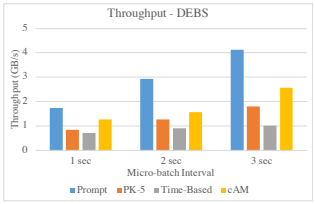
Figure 10: Data Partitioning Metrics



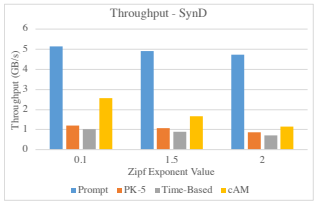
(a) Tweets (Counting)



(b) Tweets (Median)



(c) DEBS



(d) SysD

Figure 11: Effect of Variable Data Rate and Data Skew on Throughput

1. ACM DEBS 2015 Grand Challenge (DEBS): This dataset contains details of taxi trips in New York City. Data are reported at the end of each trip, i.e., upon arriving in the order of the drop-off timestamps. We define 2 sliding-window queries: *DEBS Query 1*: Total fare of each taxi over 2 hrs windows with a 5-min slide. *DEBS Query 2*: Total distanceper taxi over 45-min window with a slide of 1 min.

2. Google Cluster Monitoring (GCM): represents the execution details of a Google data cluster. The GCM queries used are similar to the ones used in [25].

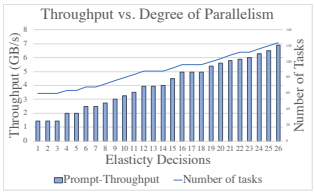
3. TPC-H Benchmark: Table *LineItem* tracks recent orders, and *TPCH Queries 1* and *6* are to generate Order Summary Reports, e.g., Query 1: Get the quantity of each Part-ID ordered over the past 1 hr. with a slide-window of 1 min.

7.2 Experimental Results

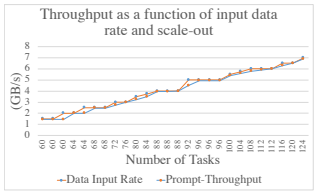
Data Partitioning. We assess the effectiveness of the proposed batch partitioning scheme using two metrics: *Block Size Imbalance - BSI* and *Block Cardinality Imbalance - BCI*. For this purpose, we use two datasets, namely Tweets and TPC-H in this experiment. We compare with existing and state-of-the-art techniques: *Shuffle*, *Hashing*, *PK-2* [36], *PK-5* [35] and *cAM* [25]. Figures 10a and 10b compare the *BSI* metric achieved for all the techniques relative to the hashing technique, i.e., as in [25]. Results for all the techniques are shown relative to the hashing technique since hashing provides no guarantees on size balancing. As the relative value approaches 0, this means the technique is providing a balanced load under this metric. In this experiment, Shuffle, Time-based, and Prompt achieve the best performance. Shuffle and Time-based partitioning are expected to achieve that as they assign equal number of tuples to the data blocks. However, they perform badly when it comes to balancing block cardinality. Also, notice that Time-based partitioning performs well on *BSI* because the data rate is fixed. Figures 10c and 10d compare the *BCI* achieved for all the techniques relative to the shuffle technique using the two datasets. In this case, the shuffle technique is used as the relative measure because it provides no guarantees on key assignment. Hashing and Prompt performs significantly better than all the other techniques. In these two experiments, **Prompt outperforms state-of-the art techniques by striking a balanced optimization for both block size and block cardinality**. This contributes to the throughput reported in Figure 11. GCM and DEBS have shown similar results but are omitted due to space limitation.

Effect of Variable Input Data Rate. We study the robustness of Prompt against sinusoidal changes to the input data rate. This simulates variable spikes in the workload. The provided resources are fixed, otherwise. Also, the target end-to-end latency is bounded by the batch interval (1,2,3 secs). The triggering of Spark Streaming's back-pressure is used to report the maximum throughput achieved. Back-pressure stabilizes the system to avoid data loss by signaling the data source to lower the input data rate. Figure 11 gives the maximum throughput achieved using the various partitioning techniques. All the techniques perform better when increasing the batch interval. However, Prompt maintains up to 2x-4x better throughput than those of cAM and Time-based partitioning. Time-based partitioning shows the worst throughput as it is sensitive to changing the data rate. While PK5 and cAM exhibit back-pressure sooner, **Prompt allows the system to achieve up to 2x throughout compared to existing techniques, before activating back-pressure.**

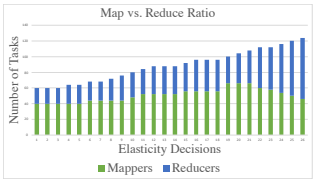
Effect of Variable Data Distribution. We use the synthetic dataset, SynD, to evaluate the performance of the partitioning techniques under skewed data distribution. In this experiment, the batch interval is set to 3 seconds. We report the highest throughput achieved for each technique before



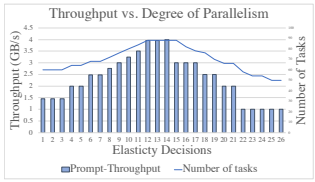
(a) Prompt Scale-out



(b) Prompt Throughput



(c) Prompt Maps/Reducers



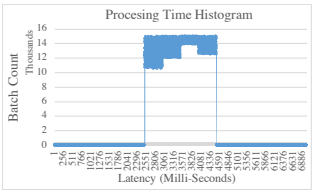
(d) Prompt Scale-In

Figure 12: Prompt Elasticity

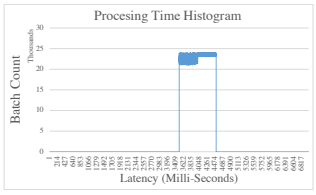
back-pressure is triggered. Figure 11d gives the performance results of all partitioning techniques under different Zipf exponent values. **In contrast to the existing techniques, Prompt consistently maintains the highest throughput even when the input data is highly skewed (between 2x to 5x better throughput).**

Latency Distribution. In this experiment, we report the processing details for thousands of batches under the default Spark Streaming's partitioner (i.e., Time-based) and when using Prompt. For each batch, we report the average completing time of the reduce tasks. In Figure 13a, the average processing time of the reduce tasks is highly variable when using Time-based data partitioning, and hence the higher distribution of latency. Figure 13b illustrates how Prompt reduce the distribution of execution time among the reduce tasks, and hence there is low variance between the latency's upper and lower bounds. The variance of reduce tasks execution depends on the the partitioning quality. The ultimate objective for micro-batch DSPSs to maintain the latency guarantee, while maximizing throughput. **By applying Prompt in Spark Streaming, both the average and maximum latencies increase because of decreasing the partitioning imbalance. This contributes to the increase in overall system throughput, while maintaining upper bound of latency**.

Resource Elasticity. We assess Prompt's ability to adjust the degree of parallelism in response to changes in workload. In the experiment, Prompt has a pool of Spark executors, each set with 4 cores. Back pressure is disabled to allow for Prompt's elasticity technique to be triggered. Figure 12a illustrates the effect of increasing the number of tasks on Prompt's throughput. We continuously increase the number of input data tuples and data distribution (i.e., number of unique keys) over time. Figure 12b illustrates how Prompt responds swiftly to the increase in workload by adding more execution tasks. Notice that when Prompt's

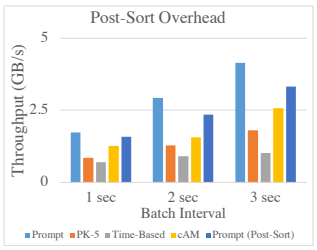


(a) Time-based

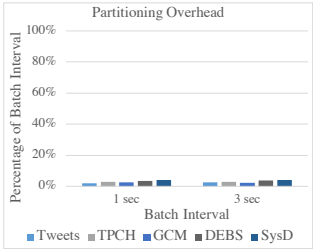


(b) Prompt

Figure 13: Latency Distribution: (b) Latency when using Time-based partitioning, (b) Latency when using Prompt.



(a) Post-Sort Throughput



(b) Prompt Overhead

Figure 14: Post-Sort cost and Partitioning Overhead: (a) Throughput of Prompt with Post-Sort, (b) Partitioning overhead for Prompt.

throughput matches the input rates, it maintains its stability and provide latency guarantees (Figure 9a). Figures 12c and 12d show the behavior when data rate is decreased and how Prompt can adapt the ratio of map/reduce tasks according to changes in data rate or data distribution. **Prompt's ability to match its throughput to that of the input workload is crucial to maintain latency guarantees**.

Partitioning Overhead. We study the overhead of applying Prompt. Figure 14a shows the effect on throughput when using post-sorting instead of Prompt Algorithm1. Figure 14b gives the time required to apply Prompt as percentage of the micro-batch interval. Observe that the cost is bounded by 5% of the micro-batch interval. Notice that this cost does not contribute to the processing time due to the use of the early batch release mechanism explained in Section 4.2. **Prompt is able to utilize the complete statistics of the batched-data while matching latency guarantees.**

8 CONSISTENCY IN PROMPT

Prompt relies on the micro-batching computational engine (i.e., [43]) to maintain consistency: (1) The isolation of state is natural as each batch is defined by system-wide heartbeats. The execution tasks are decoupled from the state that is preserved in-memory and is immutable. Window operations are defined over the states of the batches within the window's time predicate (Figure 3). (2) Exactly-once semantics is guaranteed by initially replicating the input batch. Once the batch output is produced and the batch expires from the query window, this batch can be removed. Exactly-once semantics is guaranteed at the batch level. In case of

losing a batch's state due to hardware failure, this state is recomputed using the replicated batched data. (3) The ordering of the tuples is guaranteed at a coarse-granularity as in [43], where a maximum delay (i.e., a small percentage of the batch interval) can be defined to all delayed tuples from the source to be included in the correct batch. Cases where the data tuples are expected to be delayed more than the batch-interval are to be handled outside of Prompt's execution engine, e.g., via revision tuples [15]. In addition, the optimization of join queries in map-reduce computations include several factors besides data partitioning, e.g., communication and scheduling overheads, which are out of the scope of this paper.

9 RELATED WORK

(1) Data Partitioning in Tuple-at-a-time Stream Processing Systems: Early work in parallel stream processing focus on the efficient partitioning of the incoming data stream tuples to workers. Cagari et al. [7] exploits the potential overlap of sliding-window queries to guide data partitioning. The partitioning decision is applied to each data stream tuple through a split-merge model. The number of splits, query replicas, and merge nodes are dynamically set to match the changes in workload. Cagri et al. [8] relies on forecasting the future behavior and known metrics of the workload (e.g., peak-rate). Prompt differs in that it exploits exact statistics about the data to make proper partitioning decisions. Zeitler et al. [44] propose a spitting operator where the input data stream is split into multiple sub-streams based on query semantics. Liroz-Gistau et al. [32] propose DynPart to adaptively re-partition continuously growing databases based on the workload. DynPart co-locates the newly appended data with the queries to maintain fixed execution time. Gedik et al. [18] provide a formal definition of the desired properties for stream data partitioning for System S [22]. It enables compactness by applying lossy-counting to maintain key frequencies, and uses consistent hashing for state-ful operators. Recently, the concept of key-splitting [6] has been proposed to improve load-balanced stream processing. It allows tuples with skewed keys to be sent to two or more workers [35, 36]. In addition, Nikos et al. [25] propose an enhancement to the key-splitting technique by accounting for tuple imbalance and the aggregation cost. These approaches are optimized for the tuple-at-a time stream processing. Prompt differs from these approaches in the sense that provide a formalization for the partitioning in micro-batching setting.

(2) Data Partitioning For Map-Reduce Framework: The Map-Reduce framework [13] has received criticism due to its load balancing and skewness issues [14]. Previous effort to handle these issues focus on three dimensions [11, 20, 27, 28, 30]: (1) The use of statistics from a sample of the input data to devise a balanced partitioning plan for the whole input data [17, 19]. (2) Performing a partial re-partitioning of the input data based on changing workload characteristics, i.e., query and data distribution [3, 31]. (3) Repartitioning the input of the Reduce stage dynamically based on Mappers output statistics [26, 34]. Although the micro-batch stream processing model has adopted the Map-Reduce processing model, it is different in many aspects. For example, the input data is coming online and a series of the batch jobs are launched against new data. This allows computing complete statistics as the batches build up, and devise a partitioning plan that is independently customized for every batch. Moreover, latency expectations are different in streaming workloads. Hence, the use of global statistics from all Mappers to guide the partitioning for the Reduce stage is not suitable.

(3) Adaptive Batch Resizing in Micro-Batch Stream Processing Systems: Das et al. [12] propose a control algorithm to set the batch interval based on runtime statistics. The processing time of previous batches is used to estimate the expected processing time of the next batch, and hence set the batch interval accordingly. The batch interval is set such that it matches the processing time. Similarly, Zhang et al. [45] use statistical regression techniques to estimate both batch and block sizes under variable input data rates. These techniques are orthogonal to Prompt. Batch resizing techniques treat the micro-batch stream processing engine as a black box, and focus on stabilizing the relationship between the batch interval and the processing time. However, Prompt delves into the data partitioning aspect of the micro-batch stream processing model. Prompt uses online statistics within each batch to guide its partitioning decisions. The objective is to increase the throughput of the system and maximize resource utilization for a micro-batch interval.

10 CONCLUSION

In this paper, we investigate and formulate the problem of data partitioning in the micro-batch stream processing model. We show how it is imperative to optimize the data partitioning for both the batching and processing phases of the micro-batch stream processing model to reach the peak throughput. We show that finding the optimal data partitioning is an NP-hard problem. We introduce Prompt with two heuristics for partitioning the Map and the Reduce input. We introduce elasticity to Prompt to adjust the degree of execution parallelism according to workload needs. This allows Prompt to be robust to fluctuations in data distribution and arrival rates. Finally, we present an extensive experimental evaluation of the performance of Prompt that shows that our proposed algorithms achieve up to 2x improvement in the overall system throughput compared to the state-of-the-art techniques.

ACKNOWLEDGMENTS

Walid G. Aref acknowledges the support of the U.S. NSF under Grant Numbers: IIS-1910216 and III-1815796.

REFERENCES

[1] https://spark.apache.org/.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. F. Indez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *VLDB*, 2015.

[3] A. M. Aly, A. S. Abdelhamid, A. R. Mahmood, W. G. Aref, M. S. Hassan, H. Elmeleegy, and M. Ouzzani. A demonstration of aqwa: Adaptive query-workload-aware partitioning of big spatial data. In *VLDB*, 2015.

[4] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor. M3: Stream processing on main-memory mapreduce. In *ICDE*, 2012.

[5] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Sigmod*, 2018.

[6] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *SIAMJ.Comput.*, 1999.

[7] C. Balkesen and N. Tatbul. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *8th International Workshop on Data Management for Sensor Networks (DMSN)*, 2011.

[8] C. Balkesen, N. Tatbul, and M. T. Ozsu. Adaptive input admission and management for parallel stream processing. In *DEBS*, 2013.

[9] B. Byholm and I. Porres. Fast algorithms for fragmentable items bin packing. In *TUCS Technical Report, No 1181*, 2017.

[10] C.A.Mandal, P.P.Chakrabarti, and S.Ghose. Complexity of fragmentable object bin packing and an application. In *Computers and Mathematics with Applications*. ELSEVIER, 1998.

[11] Y. Chen, Z. Liu, T. Wang, , and L. Wang. Load balancing in mapreduce based on data locality. In *ICA3PP*. Springer, 2014.

[12] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *SoCC*, 2014.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[14] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. In *Database Column*, 2008.

[15] M. C. E. Ryvkina, A. S. Maskey and S. Zdonik. Revision processing in a stream processing engine: A high-level design. In *ICDE*, 2006.

[16] L. Epstein, L. M. Favrholdt, and J. S. Kohrt. Comparing online algorithms for bin packing problems. In *Journal of Scheduling*, 2012.

[17] Y. Gao, Y. Zhou, B. Zhou, L. Shi, and J. Zhang. Handling data skew in mapreduce cluster by using partition tuning. In *Journal of Healthcare Engineering*. Hindawi, 2017.

[18] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. In *VLDB Journal*, volume 23,4, pages 75–87, 2014.

[19] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in mapreduce. In *International Conference on Cloud Computing and Services Science*, 2011.

[20] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *ICDE*, 2012.

[21] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched stream processing for data intensive distributed computing. In *SoCC*, 2010.

[22] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD*, 2006.

[23] K. Jansen, S. Kratsch, D. Marx, and I. Schlotter. Bin packing with fixed number of bins revisited. In *Journal of Computer and System Sciences*. Academic Press, 2013.

[24] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Gareyi, and R. L. Grahamii. Worst-case performance bounds for simple one-dimensional packing algorithms. In *Journal of Computing*. SIAM, 1974.

[25] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. In *VLDB*, 2017.

[26] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *IEEE*, 2012.

[27] Y. Kwon, K. Ren, M. Balazinska, and B. Howe. Managing skew in hadoop. In *TCDE*, 2013.

[28] Y. Le, J. Liu, F. Ergun, and D. Wang. Online load balancing for mapreduce with skewed data input. In *INFOCOM*, 2014.

[29] B. LeCun, T. Mautor, F. Quessette, and M.-A. Weisser. Bin packing with fragmentable items: Presentation and approximations. In *Theoretical Computer Science*. ELSEVIER, 2015.

[30] J. Li, Y. Liu, J. Pan, P. Zhang, W. Chen, and L. Wang. Map-balance-reduce: An improved parallel programming model for load balancing of mapreduce. In *FGCS*. ELSEVIER, 2017.

[31] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, and P. Valduriez. Data partitioning for minimizing transferred data in mapreduce. In *Globe*, 2013.

[32] M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez. Dynamic workload-based partitioning for large-scale databases. In *DEXA*, pages 183–190, 2012.

[33] N. Menakerman and R. Rom. Bin packing with item fragmentation. In *WADS*. Springer, 2001.

[34] J. Myung, J. Shim, J. Yeon, and Sang-goo. Handling data skew in join algorithms using mapreduce. In *Expert Systems with Applications*, 2016.

[35] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *ICDE*, 2016.

[36] 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 *ICDE*, 2015.

[37] K. Pienkosz. Bin packing with restricted item fragmentation. In *Operations and Systems Research Conference*, 2014.

[38] H. Shachnai, T. Tamir, and O. Yehezkely. Approximation schemes for packing with item fragmentation. In *Theory of Computing Systems*, 2008.

[39] H. Shachnai and O. Yehezkely. Fast asymptotic fptas for packing fragmentable items with costs. In *FCT*, 2007.

[40] D. S.Johnson. Fast algorithms for bin packing. In *Journal of Computer and System Sciences*. ELSEVIER, 1974.

[41] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, 2014.

[42] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[43] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[44] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. In *VLDB*, 2011.

[45] Q. Zhang, Y. Song, R. R. Routray, and W. Shi. Adaptive block and batch sizing for batched stream processing system. In *IEEE International Conference on Autonomic Computing*, 2016.