EFFICIENT DISTRIBUTED PROCESSING OVER MICRO-BATCHED DATA STREAMS

by

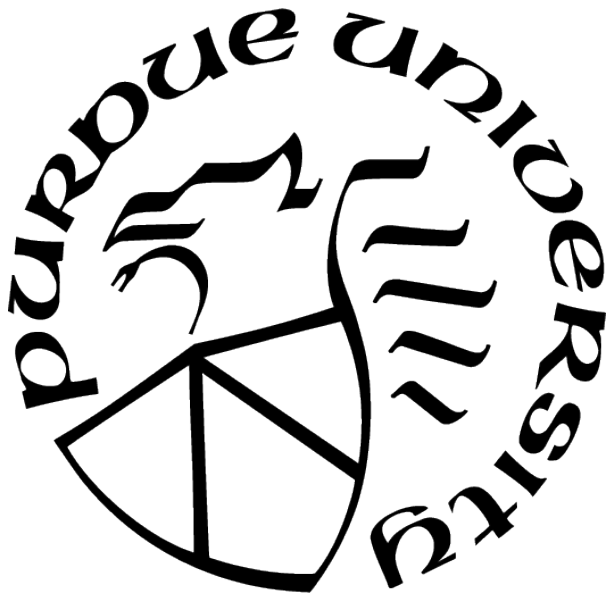
Ahmed S. Abdelhamid

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



Department of Computer Science

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Prof. Walid G. Aref, Chair

Department of Computer Science

Prof. Elisa Bertino

Department of Computer Science

Prof. Sunil Prabhakar

Department of Computer Science

Prof. Sonia Fahmy

Department of Computer Science

Approved by:

Prof. Kihong Park

To my family

ACKNOWLEDGMENTS

I would like to begin by thanking the Almighty Allah for the blessings and favors He has bestowed upon me, and for the endless opportunities, He has granted me to pursue my degree. This dissertation would not have been possible without the help and support of many people who touched different aspects of my life.

I thank my parents who stood by me and supported me all my life. I am grateful to my father for teaching me the importance of always being patient, and to my mother for her care and warmth. I am also thankful to my brothers for their support and encouragement. They taught me the importance of education, persistence, perseverance, and never give up. This has been one of the main ingredients for me to pursue my Ph.D.

I would like to express my gratitude to Professor Walid G. Aref. His wealth of experience in advising research projects and his utmost support during my stay at Purdue have helped me the most in producing this dissertation. He has been extremely patient with me helping me improve my research and problem-solving skills. I always found his advice incredibly useful in developing my technical and research skills. I am thankful to him for helping me develop a researcher's mindset, and for teaching me how to explore new ideas for research.

I thank Prof. Gustavo Rivera for mentoring my teaching fellowship. He helped to enrich my teaching skills.

I thank my mentors in Microsoft Research; Jonathan Goldstein, Badrish Chandramouli, and Umar Farooq Minhas for their kind advice and support.

Finally, I would like to thank CS Department for having me in their Teaching Assistant group. Their Funding has covered most of my Ph.D. tuition and offered me a great experience of teaching. Also, I thank Dr. Gorman and Monica Shively for their hospitality and their invaluable input, support, and explanations.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Advances in real-world applications require high-throughput processing over large data streams. Micro-batching is a promising computational model 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. Compared to native tuple-at-a-time data stream processing, micro-batching can sustain higher data rates. However, existing micro-batch stream processing systems lack *Load-awareness* optimizations that are necessary to maintain performance and enhance resource utilization. In this thesis, we investigate the micro-batching paradigm and pinpoint some of its design principles that can benefit from further optimization. A new data partitioning scheme termed *Prompt* is presented that leverages the characteristics of the micro-batch processing model. Prompt enables a balanced input to the batching and processing cycles of the micro-batching model. Prompt achieves higher throughput processing with an increase in resource utilization. Moreover, *Prompt+* is proposed to enforce latency by elastically adapting resource consumption according to workload changes. More specifically, *Prompt+* employs a scheduling strategy that supports elasticity in response to workload changes while avoiding rescheduling bottlenecks. Moreover, we envision the use of deep reinforcement learning to efficiently partition data in distributed streaming systems. *PartLy* demonstrates the use of artificial neural networks to facilitate the learning of efficient partitioning policies that match the dynamic nature of streaming workloads. Finally, all the proposed techniques are abstracted and generalized over three widely used stream processing engines. Experimental results using real and synthetic data sets demonstrate that the proposed techniques are robust against fluctuations in data distribution and arrival rates. Furthermore, it achieves up to 5x improvement in system throughput over state-of-the-art techniques without degradation in latency.

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 click-stream analysis. Recently, several DSPSs have adopted a batch-at-a-time processing model to improve the processing throughput (e.g., as in Spark Streaming [61], M3 [5], Comet [26], and Google DataFlow [3]). 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 [56], efficient fault-tolerance [61], and seamless integration with offline data processing [6]. 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 with the fluctuations in arrival rates and data distributions.

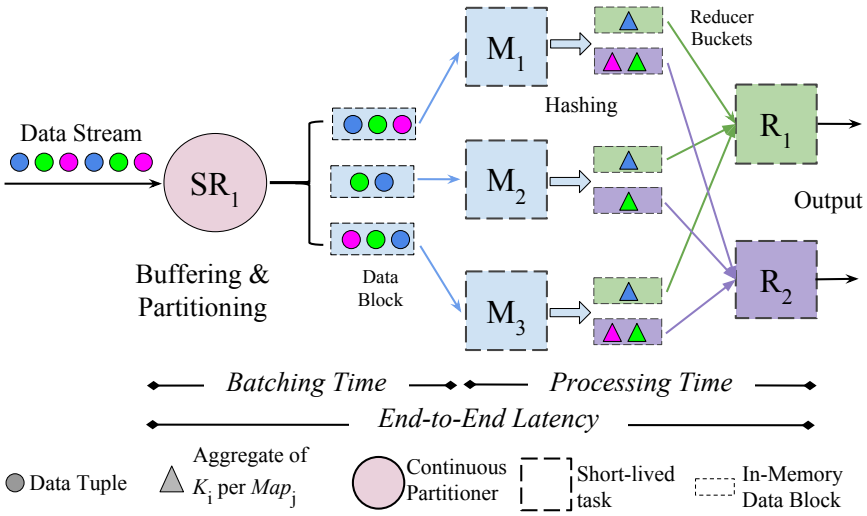


Figure 1.1. Example of micro-batch stream processing: The computation has three Map and two Reduce tasks, and a Stream Receiver (SR_1) that provides micro-batches from the input data stream.

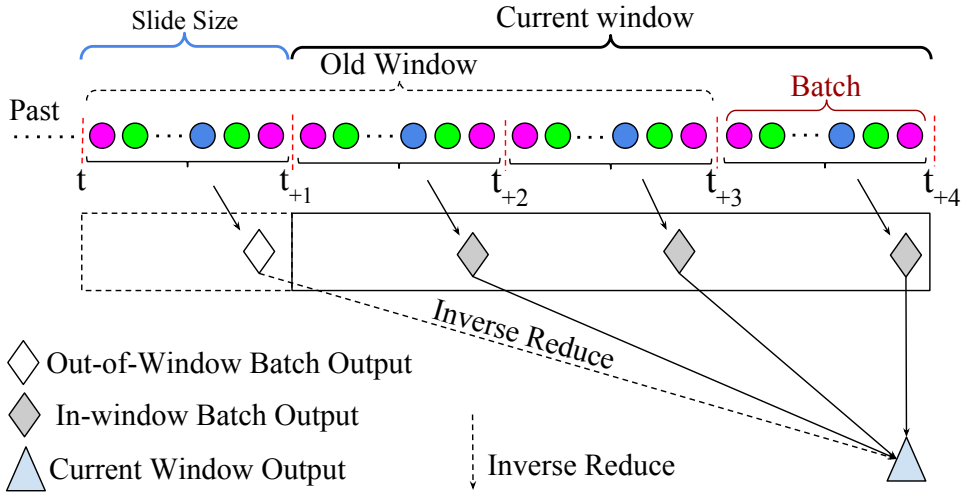


Figure 1.2. Micro-batch Stream Processing

1.1 Micro-batch Stream Processing

The micro-batching computational model can be best explained using an example query. 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 1.1). 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 1.3). In the *batching* phase, the stream data tuples are accumulated for a predetermined batch interval. The interval size is set according to the target application latency. 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 parallel task-oriented stages (e.g., Map and Reduce stages). A centralized scheduler is informed of the whereabouts of the buffered data blocks and schedule the Map stage. In the Map stage, a user-defined function is applied in parallel to every data block (e.g., a filter over the click-stream tuples). Upon completion, the centralized scheduler is informed of the whereabouts of the intermediate results and makes a schedule decision to start the Reduce stage. The Reduce stage aggregates the outcome of the Map stage to produce the output of the batch (e.g., sums the clicks for each country). Figure 1.4a explains the activities during

the processing phase. Finally, the query answer is computed by aggregating the output of all batches that are within the query's time-window (See Figure 1.2). The end-to-end latency is defined at the granularity of a batch and is equal to 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 thesis, 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). Also, *MapReduce* refers to a computation that consists of a parallel function followed by an aggregation.

1.2 Research Challenges

In this thesis, we address three challenges, namely data partitioning, task-scheduling overhead, and performance stability:

1.2.1 Data Partitioning

Data partitioning is crucial to the batching and 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 1.3. 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 1.3 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 arrives. 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

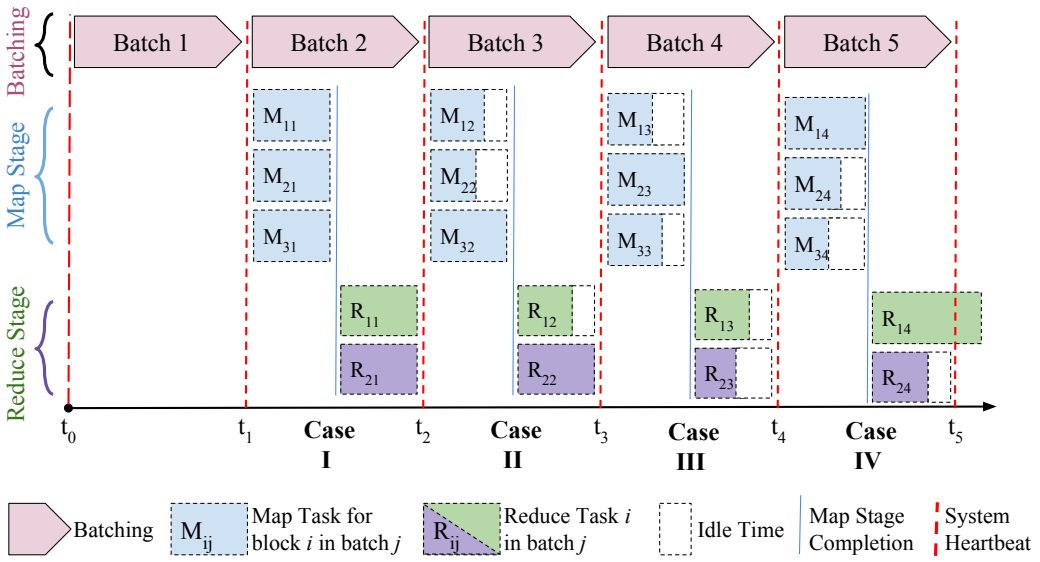
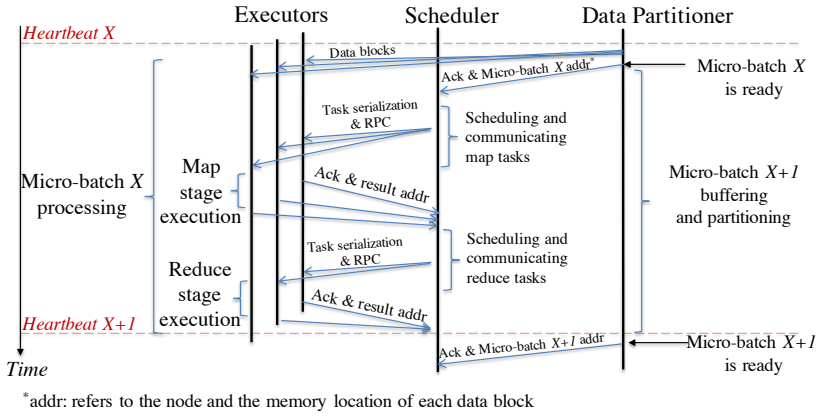


Figure 1.3. The effect of data partitioning on the pipelined-execution over micro-batches: An example timeline that illustrates three different cases of unbalanced-load execution in $[t_0$-$t_5]$. Notice that Batch x is processed concurrently while Batch x+1 is being accumulated

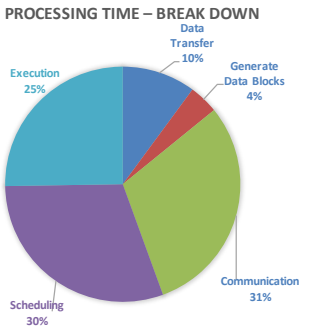
complex. It entails many optimization factors, e.g., key locality, where tuples with the same key values need to be co-located into the same data blocks. In fact, as will be explained in Chapter 2, this data partitioning problem is NP-hard. **(3)** In a Map-Reduce computation, intermediate results of 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.

1.2.2 Scheduling Bottleneck

Another hurdle for micro-batched systems is that of communication and centralized scheduling. The coordination and communication with a centralized scheduler to orchestrate the computation is very costly. Figure 1.4a shows the centralized role of the scheduler during the micro-batch computation. The scheduler is responsible for keeping track of the accumulating data blocks and the intermediate results. The scheduler leverages this information when initiating the execution tasks at the processing stages. Clearly, the centralized



(a) Activity Breakdown of One Micro-batch Computation



(b) Breakdown of Micro-batch Processing Time (e.g., Spark Streaming)

Figure 1.4. Typical activities during the Processing Time of a micro-batch computation

scheduler is a bottleneck. Figure 1.4b gives an example of the percentages of the different activities that are typically carried out during the processing time of a micro-batch. Due to coordination with a centralized scheduler, the actual data processing is only a fraction of the time available to process the micro-batch. Scheduling and communication consume a significant fraction of the time (e.g., at least 60%, as seen in Figure 1.4b).

1.2.3 Performance Stability

A healthy relationship between processing time and the 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 (e.g., by dynamically increasing the number of mappers and reducers to meet the changes in the workload). 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 [46,47,53,61]. To maintain their 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 of resources.

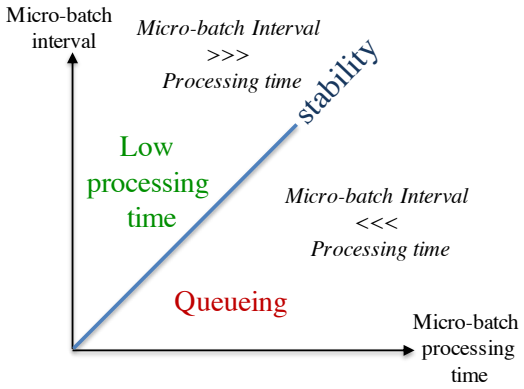


Figure 1.5. System Stability: Workload Behavior in Micro-batch Processing Model

1.3 Research Contributions

This dissertation addresses the challenges associated with efficient data stream processing over the micro-batch processing model. The research contributions of this dissertation are as follows:

• We analyze the micro-batching design principles. We provide motivating break-down analysis of the micro-batching computational model and highlight the need for efficient data partitioning and task-scheduling techniques. We demonstrate that micro-batching is a promising solution for today's data-intensive streaming workloads when the three performance bottlenecks are mitigated. These challenges are data partitioning, centralized-scheduling, and performance stability.

• To support efficient data-partitioning for the micro-batch processing model, 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 leverages a look-ahead data partitioning strategy that optimizes the performance of the micro-batch processing model [2]. In the batching phase, Prompt has a 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. Prompt improves system throughput by up to 2x using real and synthetic datasets over state-of-the-art techniques.

• To mitigate the centralized-scheduling bottleneck and the performance stability concerns of the micro-batching computational model, we introduce Prompt+ with an elastic-scheduling technique for the micro-batching model. Prompt+ aims to overcome the overheads of task scheduling (i.e., serialization and communication) and maintaining resource utilization, while enforcing latency. Prompt+ schedules multiple batch-computations at once. This serves to amortize the scheduling cost over a group of batches. In addition, Prompt+ uses three elasticity zones to monitor system stability w.r.t. workload changes. Prompt+ updates the degree of task-parallelism, without interrupting the execution, to meet workload needs. Prompt+ is robust to fluctuations in data distribution and arrival rates. Experiments on the evaluation of the proposed elastic-scheduling technique and the combined effect, when applied along with the data partitioning techniques, show up to 5x improvement in throughput while enforcing latency guarantees.

• To further support efficient data stream processing over the micro-batch processing model, we introduce PartLy, a learning-based technique to provide efficient data-partitioning of micro-batched data over execution tasks. PartLy leverages the deep reinforcement learning techniques to learn a data partitioning policy using the run-time statistics of the execution tasks. PartLy enhances over Prompt through reducing the variance of execution-tasks. We formulate the data partitioning of the micro-batching model in the context of deep reinforcement learning. PartLy opens the possibility of autonomous execution and reduces the overheads of tuning heuristics by humans. We provide directions on how to extend this strategy to the elasticity problem.

• We study the realization of the proposed techniques in Prompt, Prompt+, and PartLy within three widely-adopted micro-batched stream processing systems. We demonstrate how the ideas proposed in this thesis are general and applicable in Spark Streaming [61], Apache Flink [12] and Apache Storm [48].

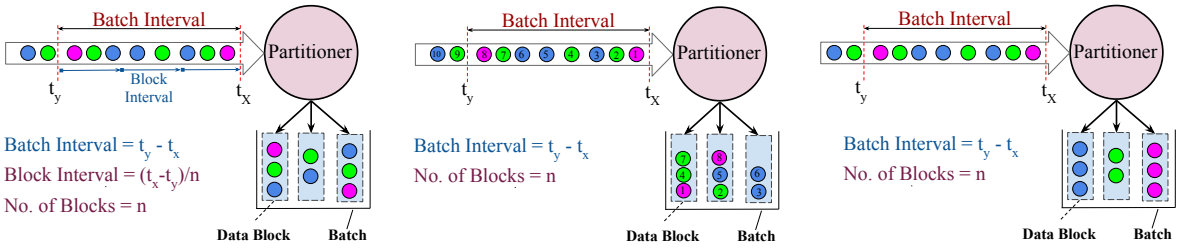
1.4 Dissertation Outline

This dissertation is organized as follows. Chapter 2 introduces the data partitioning scheme for maximizing throughput in micro-batched stream processing systems. Chapter 3 introduces an elastic-scheduling technique for enforcing latency in micro-batched stream processing systems. Chapter 4 introduces a learning-based model for data partitioning and envision a learned multi-agent scheme for achieving load-awareness in micro-batched stream processing systems. Chapter 5 illustrates the adoption of the presented techniques in Apache Spark, Apache Flink, and Apache Storm. Finally, Chapter 6 presents concluding remarks for the dissertation.

2. PROMPT: HIGH-THROUGHPUT PROCESSING WITH BALANCED DATA PARTITIONING

2.1 Problem Statement

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 most popular data-partitioning techniques are *time-based* [61], *hashing*, and *shuffling*.



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

Figure 2.1. Existing Data partitioning Techniques

Time-based partitioning uses the arrival time of a new tuple to assign the tuple into a *data block* (Figure 2.1a). 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. On the other hand, Shuffle partitioning assigns tuples to data blocks in a round-robin fashion based on arrival order without considering other factors (Figure 2.1b). 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. Hash partitioning, also termed *Key Grouping* [47], 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 2.1c). 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 2.1c). 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.

In this chapter, we address the problem of data partitioning in the micro-batching computational model.

2.2 Related Work

Early work in parallel stream processing focus on the efficient partitioning of the incoming data stream tuples to workers. Cagari et al. [8] 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. [9] rely 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. [62] propose a spitting operator where the input data stream is split into multiple sub-streams based on query semantics. Liroz-Gistau et al. [40] 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. [23] provide a formal definition of the desired properties for stream data partitioning for System S [29]. It enables compactness by applying lossy-counting to maintain key frequencies, and uses consistent hashing for stateful operators. Recently, the concept of key-splitting [7] has been proposed to improve load-balanced stream processing. It allows tuples with skewed keys to be sent to two or more workers [46, 47]. In addition, Nikos et al. [32] 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 providing a formalization and a solution for the partitioning in micro-batch setting.

In addition, the Map-Reduce framework [15] has received criticism due to its load balancing and skewness issues [16]. Previous effort to handle these issues focus on three dimensions [13, 25, 35, 36, 38]: (1) The use of statistics from a sample of the input data to devise a balanced partitioning plan for the whole input data [22, 24]. (2) Performing a partial repartitioning of the input data based on changing workload characteristics, i.e., query and data distribution [4, 39]. (3) Repartitioning the input of the Reduce stage dynamically based on Mappers output statistics [33, 45]. 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 arrives 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.

2.3 Balanced Data Partitioning

2.3.1 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:

$$\max_{1\leq i\leq n} MapTaskTime_i + \max_{1\leq j\leq m} ReduceTaskTime_j \tag{2.1}$$

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., refer to Case I of Figure 1.3. 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 1.3 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 run-time. 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.

2.3.2 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 three 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.

2.3.3 Cost Model

We introduce the cost model for data partitioning that captures the problem formulation in Section 2.3.2. 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 [46], 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:

$$BSI(Blocks) = max_{\mathrm{i}}|Block_{\mathrm{i}}| - avg_{\mathrm{i}}|Block_{\mathrm{i}}|\mathrm{i} \in p \tag{2.2}$$

Eqn. 2.2 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. 2.3 models the size imbalance at the Reduce stage for the buckets, where r is the number of Reduce buckets:

$$BSI(Buckets) = max_{\mathrm{j}}|Bucket_{\mathrm{j}}| - avg_{\mathrm{j}}|Bucket_{\mathrm{j}}|\mathrm{j} \in r \tag{2.3}$$

Eqn. 2.4 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 [32].

$$BCI(Blocks) = max_{\mathrm{i}}||Block|| - avg_{\mathrm{i}}||Block)|| \quad \mathrm{i} \in p \tag{2.4}$$

Let a *key fragment* be a collection of tuples that share the same key value. Eqn. 2.5 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.

$$KSR(Blocks) = \frac{Sum_k|Fragments|}{Sum_k|Keys|} \quad k \in K \tag{2.5}$$

Finally, we define the overall *Partitioning-Imbalance* metric [32] over a *Micro-batch* (MPI, for short) using a combination of the above metrics as follows:

$$MPI(Blocks) = p_1 * BSI + p_2 * BCI + p_3 * KSR \tag{2.6}$$

Notice that the objective is to minimize the three metrics that lead to minimize MPI. We provide a mathematical formulation for the problem in Section 2.3.2. 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.

2.4 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 two 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 two steps.

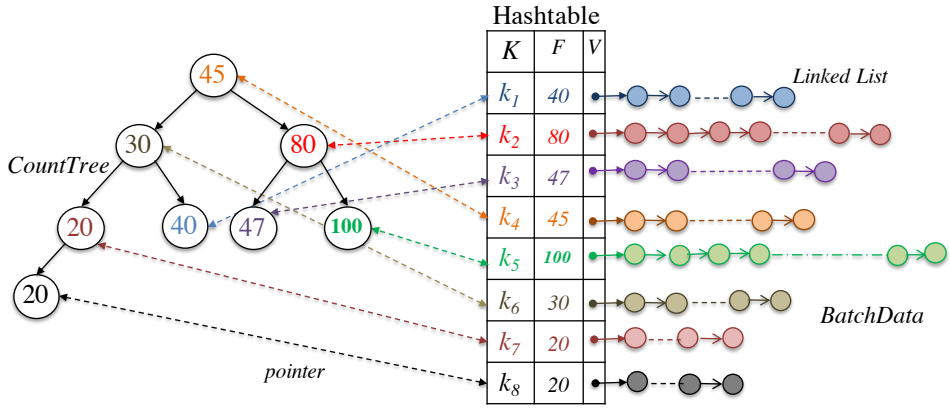


Figure 2.2. Frequency-aware Micro-batch Buffering: Fully-updated *CountTree* and *HTable* after receiving 385 tuples with eight distinct keys.

2.4.1 Frequency Aware Buffering

As explained in Chapter 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 run-time 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 2.2 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 10-100k 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 proportion 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 update 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.

Algorithm 1: Micro-batch Accumulator

Input: S: Input Stream, [t_{start}-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},$count_{\mathrm{i}}$,$tupleList_{\mathrm{i}}$>

Reset $HTable$, and $CountTree$;
while $tuple_{\mathrm{i}}.ts \in$ *Batch Interval* **do**
  Increment number of tuples Count: N_C;
  if $tuple_{\mathrm{i}}.k \in HTable$ **then**
    Insert $tuple_{\mathrm{i}}$ into $HTable_k$ chain;
    Update $k.Freq_{Current}$;
    $Delta_{freq} = k.Freq_{Current}$ - $k.Freq_{Updated}$;
    $Delta_{time} = Time_{Now}$ - $k_{LastUpdateTime}$;
    if $k_{f.step} == Delta_{freq}$ **then**
      Update k_{freq} in $CountTree$;
      Update $k.budget = k.budget$ -1 ;
      Update $k.Freq_{Updated}$;
      Set $k_{f.step} = (N_{EST}/budget)$*$k.Freq_{Current}/N_C$;
    else
      if $k_{t.step} == Delta_{time}$ **then**
        Update k_{freq} in $CountTree$;
        Update $k.budget = k.budget$ -1 ;
        Update $k.Freq_{Updated}$;
        Set $k_{t.step} = (t_{end}$-$Now_{Time})/k.budget$;
      else
        k is not eligible for an update yet;
      end
    end
  else
    Increment Unique Keys Count: $|K|$;
    Insert $tuple_{\mathrm{i}}$ into $HTable$;
    Insert $tuple_{\mathrm{i}}.k$ as new node into $CountTree$;
    Initialize $k.Freq_{Current}$ and $k.Freq_{Updated}$ to 1;
    Initialize $k_{t.step} = (t_{end}$ - $Time_{Now})/budget$;
    Initialize $k_{f.step} = f$;
  end
end

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.

2.4.2 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 share 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 share same key value. We refer to this problem as *Balanced Bin Packing with Fragmentable Items (B-BPFI, for short)*. 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 bin is 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 2.6, 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 2.3a and 2.3b give two possible assignments into four data blocks ($B1$, $B2$, $B3$ and $B4$) for the batched data in Figure 2.2. In both assignments, the data blocks are of equal size. However, the well-known First-Fit-Decreasing technique [44], illustrated in Figure 2.3a, does not minimize item fragmentation. This results in fragmenting three out of the eight keys, namely, K_1, K_2, and K_4. In contrast, in Figure 2.3b, the use of the Fragmentation Minimization technique [31] 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:

Definition 2.4.1 *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 three 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) Every item* $\in K$ *is split over a minimum number of bins.*

The problem can be formulated mathematically as follow:

$$Minimize \qquad (\sum_{i=1}^{K}\sum_{j=1}^{B} y_{ij}) \tag{2.7}$$

so that:

$$\sum_{j=1}^{B} x_{ij} = s_i \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{2.8}$$

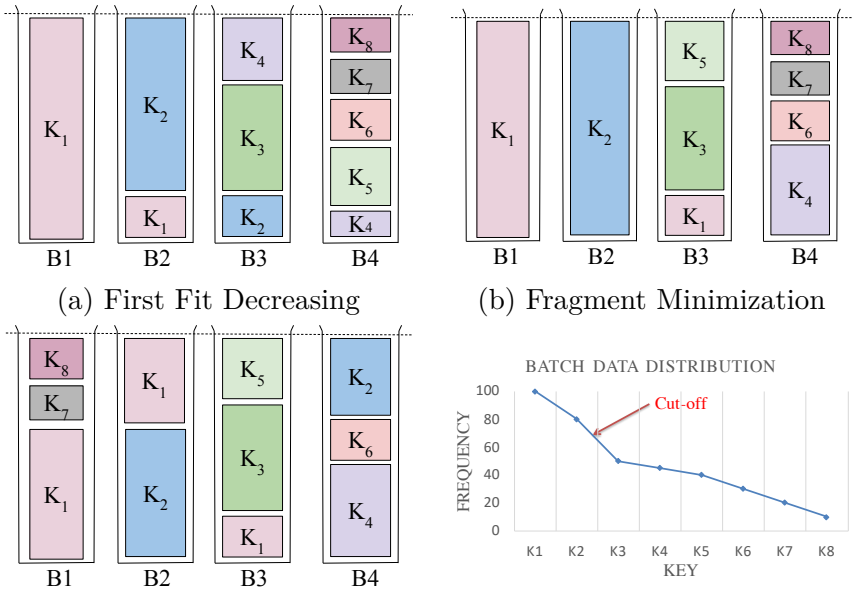
$$\sum_{i=1}^{K} x_{ij} \leq c_j \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{2.9}$$

$$\frac{x_{ij}}{s_i} \leq y_{ij} \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{2.10}$$

$$\frac{K}{B} \leq \sum_{i=1}^{K}\sum_{j=1}^{K} y_{ij} \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{2.11}$$

$$x_{ij} \in N^+, y_{ij} \in 0, 1 \qquad \forall i \in [1...K], \forall j \in [1...B] \tag{2.12}$$

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. 2.8). Eqn. 2.8 implies that the sum of the sizes of the fragments of any item i must be equal to its total size si. Eqn. 2.9 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. 2.12) 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 2.10 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. 2.7). Without loss of granularity, we assume that the data tuples are of the same size for simplicity. However,



(a) First Fit Decreasing (b) Fragment Minimization

(c) Balanced Fragment-Min. (d) Key Split Cut-off

Figure 2.3. Assignment Trade-offs for the Problem of Bin Packing with Fragmentable Items

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.

$$\sum_{\mathrm{i}=1}^{K} s_{\mathrm{i}} \leq \sum_{\mathrm{j}=1}^{B} c_{\mathrm{j}} \qquad \forall \mathrm{i} \in [1...K], \forall \mathrm{j} \in [1...B] \tag{2.13}$$

Theorem 2.4.1 *The Balanced Bin Packing with Fragmentable Items problem is NP-Complete.*

Proof 1 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 [52] and has been studied for decades. Some of its forms have even dealt with fragmentable items [10, 11, 17, 30, 31, 37, 44, 49–51]. 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 [37]. 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.

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 2.4). This allows the partitioner a slack time to execute the partitioning algorithm 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.

2.5 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 2.5a gives an example of the default assignment of a Map task output to its Reduce buckets using conventional

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

```
1  Define Partition-Size: P_Size=N_C/P;
2  Define Partition-Cardinality: P_|k| = K/P;
3  Define Key-Split-CutOff: S_Cut = P_Size/P_|k|;
4  Set b_i = b_1 (b_1 ∈ Partitions);
5  while ∃k ∈ List and |k| > S_Cut do
6  |   Put S_Cut fragment → b_i and add residual to RList;
7  |   Update lookupLargePos(k ↔b_i);
8  |   Set b_i to b_i++%P;
9  end
10 while ∃k ∈ List do
11 |   foreach b ∈ Partitions do
12 |   |   Put one key in b;
13 |   |   Move to next b;
14 |   end
15 |   Reverse Order of Partitions loop;
16 end
17 while ∃k ∈ RList do
18 |   b = lookupLargePos(k);
19 |   if k fits in b then
20 |   |   Add k to b;
21 |   else
22 |   |   Fill b from k;
23 |   |   Add rest of k to partition with lowest remaining capacity that can hold it;
24 |   end
25 end
```

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

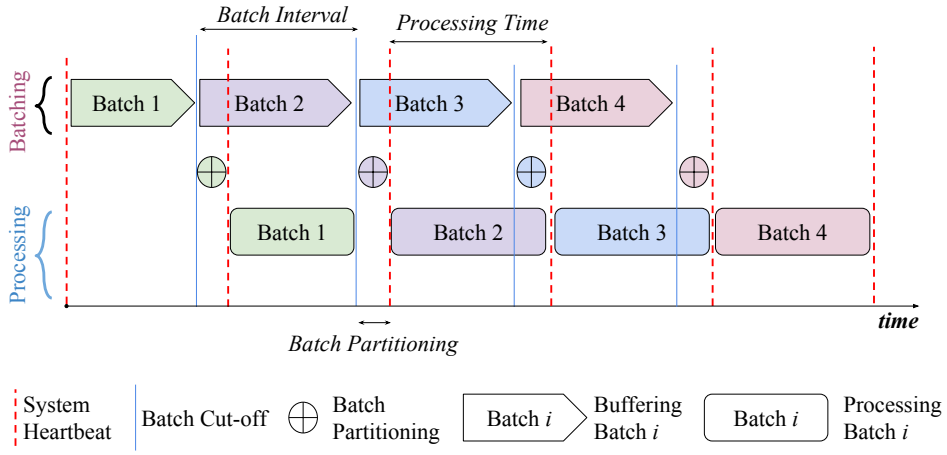
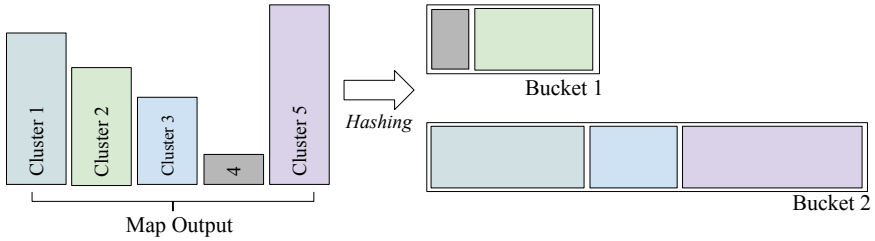
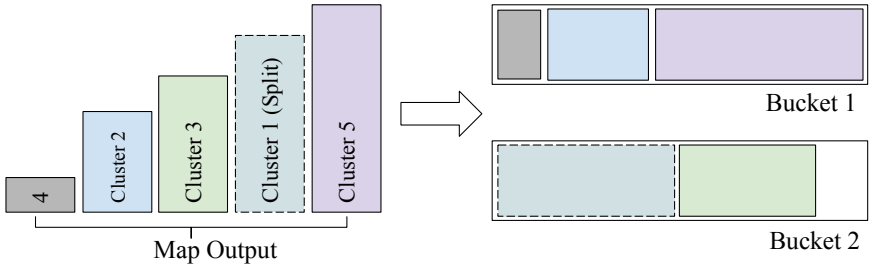


Figure 2.4. Early Batch Release



(a) Hashing Assignment



(b) Balanced Assignment

Figure 2.5. Reduce Replacement Strategies

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:

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

```
1  Define Bucket_size = |C|/|R|;
2  Assign SplitKeys to R using Hashing;
3  Let C= C - SplitKeys;
4  Sort NonSplit key Clusters in descending order;
5  while ∃k ∈ C do
6  |   Assign k into Bucket r= Worst-Fit(R) ;
7  |   R = R-r ;
8  |   if !∃r ∈ R then
9  |   |   Reset R = All Reduce Bucket ;
10 |   else
11 |   end
12 end
```

Definition 2.5.1 *Balanced Bin Packing with Variable Capacity (B-BPVC). Given a set of items, K, and B bins, each with 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 2.5.1 *The Balanced Bin Packing with Variable Capacity (B-BPVC) problem is NP-Complete.*

Proof 2 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. ■

Algorithm 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.

2.6 Experimental Evaluations

2.6.1 Experimental Setup

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 [5], Comet [26], and Google Dataflow [3]. Prompt is realized by modifying four components in Spark Streaming [61] as described in Section 5.1.2. The same JVM heap size and garbage collection flags are applied to launch all Spark executor instances.

The queries of the used benchmarks are written as a map-reduce computation. Figure 1.1 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 1.2. 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 [47],PK5 [46] and cAM [32], 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 [47],PK5 [46], the number of candidates per key are fixed at 2 and 5, respectively. For cAM [32], 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.

Datasets and Workloads

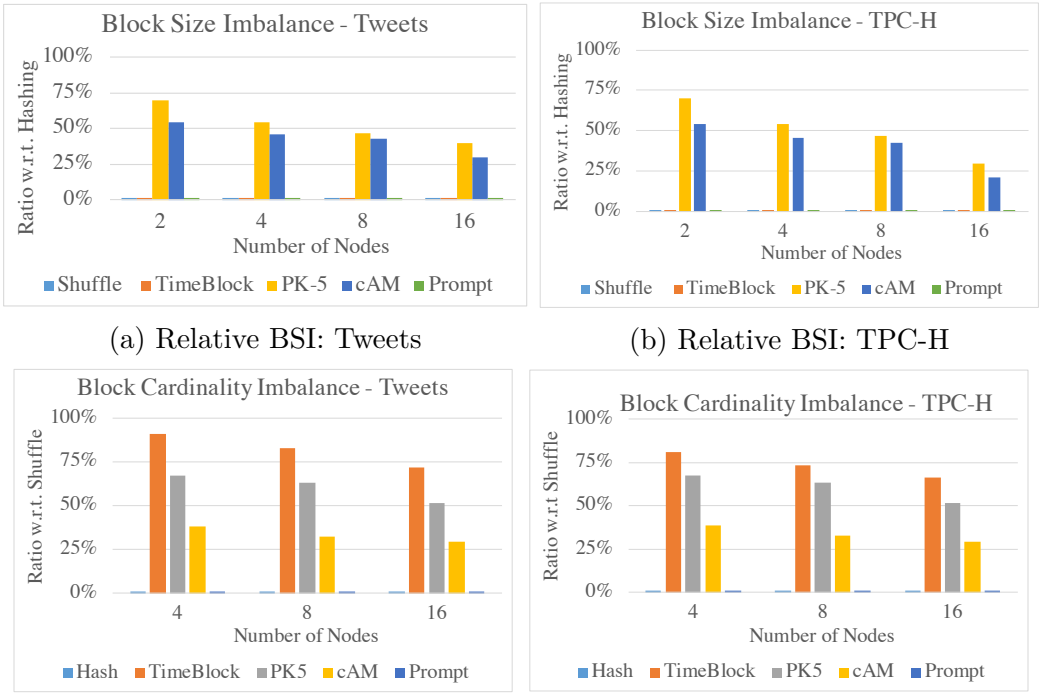
We test the performance of the proposed techniques using various workloads of increasing complexity : *WordCount* performs a sliding window count over 30 seconds, and *TopKCount* finds the k most-frequent words over the past 30 seconds. We use the following two datasets, namely Tweet and SynD. Tweet 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:

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 [32].

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.

2.6.2 Experimental Results



(a) Relative BSI: Tweets

(b) Relative BSI: TPC-H

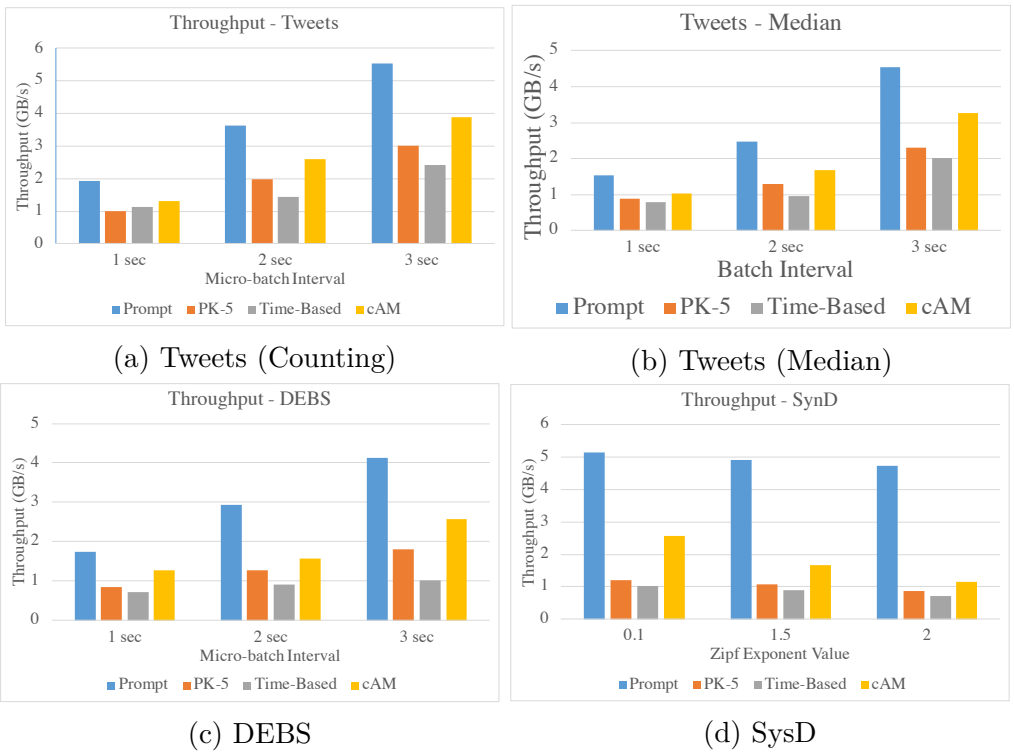
(c) Relative BCI: Tweets

(d) Relative BCI: TPC-H

Figure 2.6. Data Partitioning Metrics

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* [47], *PK-5* [46] and *cAM* [32]. Figures 2.6a and 2.6b compare the *BSI* metric achieved for all the techniques relative to the hashing technique, i.e., as in [32]. 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 2.6c and 2.6d 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 2.7.



(a) Tweets (Counting)

(b) Tweets (Median)

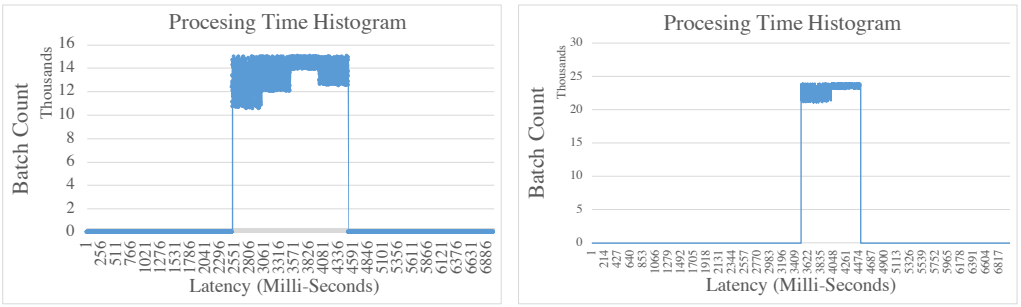
(c) DEBS

(d) SysD

Figure 2.7. Effect of Variable Data Rate and Data Skew on Throughput

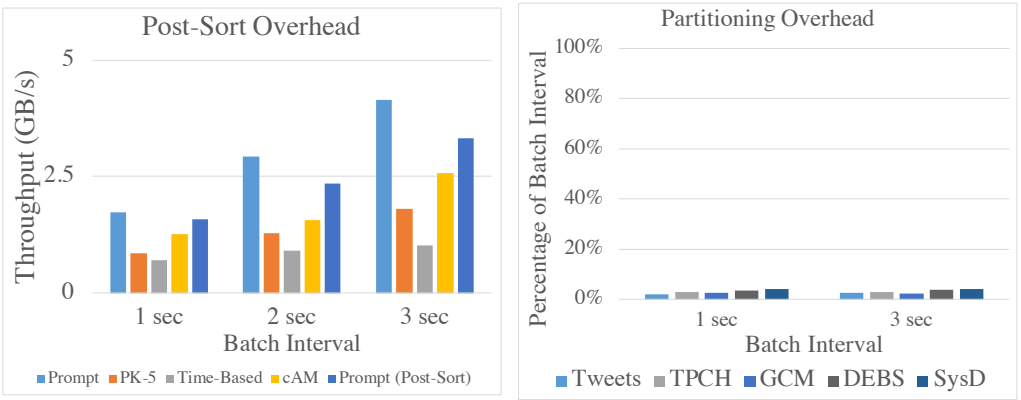
Effect of Variable Input Data Rate. We study the robustness of Prompt against sinusoidal changes to the input data rate. This simulates variable spikes in the workload. The provided resources are fixed, otherwise. Also, the target end-to-end latency is bounded by the batch interval (1,2,3 secs). The 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 2.7 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 minimum throughput as it is highly 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.



(a) Time-based (b) Prompt

Figure 2.8. Latency Distribution: (a) Latency when using Time-based partitioning, (b) Latency when using Prompt.



(a) Post-Sort Throughput (b) Prompt Overhead

Figure 2.9. Post-Sort cost and Partitioning Overhead: (a) Throughput of Prompt with Post-Sort, (b) Partitioning overhead for Prompt.

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 back-pressure is triggered. Figure 2.7d 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 2.8a, 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 2.8b 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.

2.7 Concluding Remarks

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

3. PROMPT+: LATENCY ENFORCEMENT WITH ELASTIC SCHEDULING

3.1 Problem Statement

Streaming-based engines execute workloads as a topology of operators (see Figure 3.1). These topologies are often long-running, deployed on multiple nodes, and controlled through a centralized scheduler. To meet workload demands, each operator in the topology is multi-threaded. The streamed data tuples are continuously ingested by specialized operators (e.g., Spout in Storm [48], Receiver in Spark [61], StreamReader in Flink [12]). These specialized operators are equipped with a data-partitioning strategy to distribute the data tuples to the downstream operators. The objective is for the tuples to flow across the topology as fast and as efficient as possible to meet performance criteria (e.g., minimizing the average tuple latency under some resource constraints).

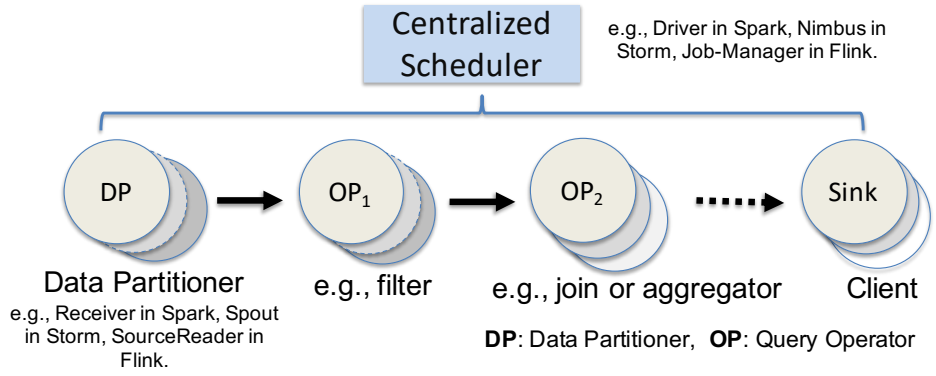


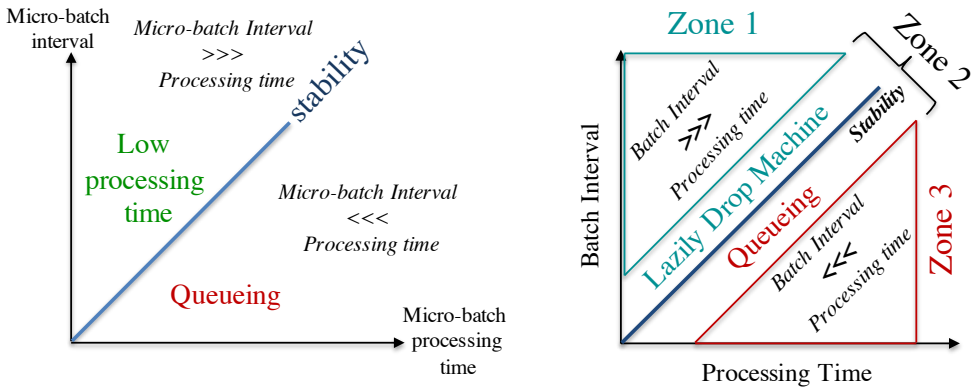
Figure 3.1. Data flow in a distributed streaming topology

In this chapter, we address the problems of scheduling and communication bottlenecks in the micro-batching computational model.

3.2 Related Work

Previous work on improving the performance of micro-batch DSPSs focuses two approaches. The first approach is resizing the batch interval [14, 63]. The batch interval is *resized* to maintain an equal relationship between the processing and batching times. How-

ever, 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. The second approach is *group scheduling of micro-batch computations* [56]. Group scheduling alleviates the scheduling overhead by amortizing the communication and scheduling overheads over multiple micro-batches. However, determining the group size is a challenge. For instance, a larger group minimizes the overheads of scheduling, but limits the system ability to elastically expand or shrink in resources in response to workload changes. The system can only add or remove resources after processing an entire group of computations. In addition, Das et al. [14] 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 sets the batch interval accordingly. The batch interval is set such that it matches the processing time. Similarly, Zhang et al. [63] 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 scheduling aspect of the micro-batch stream processing model. Prompt+ uses online statistics within each batch to guide its elasticity decisions. The objective is to increase the throughput of the system, and to maximize resource utilization for a micro-batch interval.



(a) Workload Behavior

(b) Prompt+'s Zones

Figure 3.2. Prompt+'s Elasticity Zones

3.3 Resource Utilization

We introduce Prompt+'s technique to adaptively adjust the degree of execution parallelism according to workload needs. The objective of this technique is two folds: (1) Enforce latency requirements of the users' applications, while maximizing resource utilization. (2) Eliminate the communication overheads of the centralized scheduler. 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 1.1). To enforce latency, Prompt+ continuously monitors the relationship between the *batch interval* and the *processing time* for the running micro-batches. Figure 3.2a 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. On the other hand, a centralized scheduler is responsible for managing the processing of the execution graph (See Figure 1.4a). Centralized scheduling has many benefits. For instance, it allows the scheduler to make prompt decisions to include more mappers or reducers at any of the stages during the execution. However, as explained earlier, significant amount of the processing time is spent on the communication and coordination between the workers and this centralized scheduler. A sound strategy is to schedule multiple micro-batches at once, where the data dependencies between mappers and reducers are predefined in advance. In this case, the execution graph can be completed without the intervention of the centralized scheduler. This amortizes the scheduling costs across several micro-batches, and reduces the network overhead of serialization and remote procedure calls (RPCs) as tasks get combined into a single message. In our experiments, we have observed 4-8x increase in throughput

when applying this strategy alone. However, the strategy poses a dilemma. As the number of to-be-scheduled micro-batches is increased, the benefits of the strategy is intensified while limiting the scheduler's ability to react to workload changes. To enable efficient elasticity decisions, we introduce a new scheduling mechanism that does not interrupt processing, and can enforce latency guarantees. Our proposed scheme is based on two techniques illustrated below:

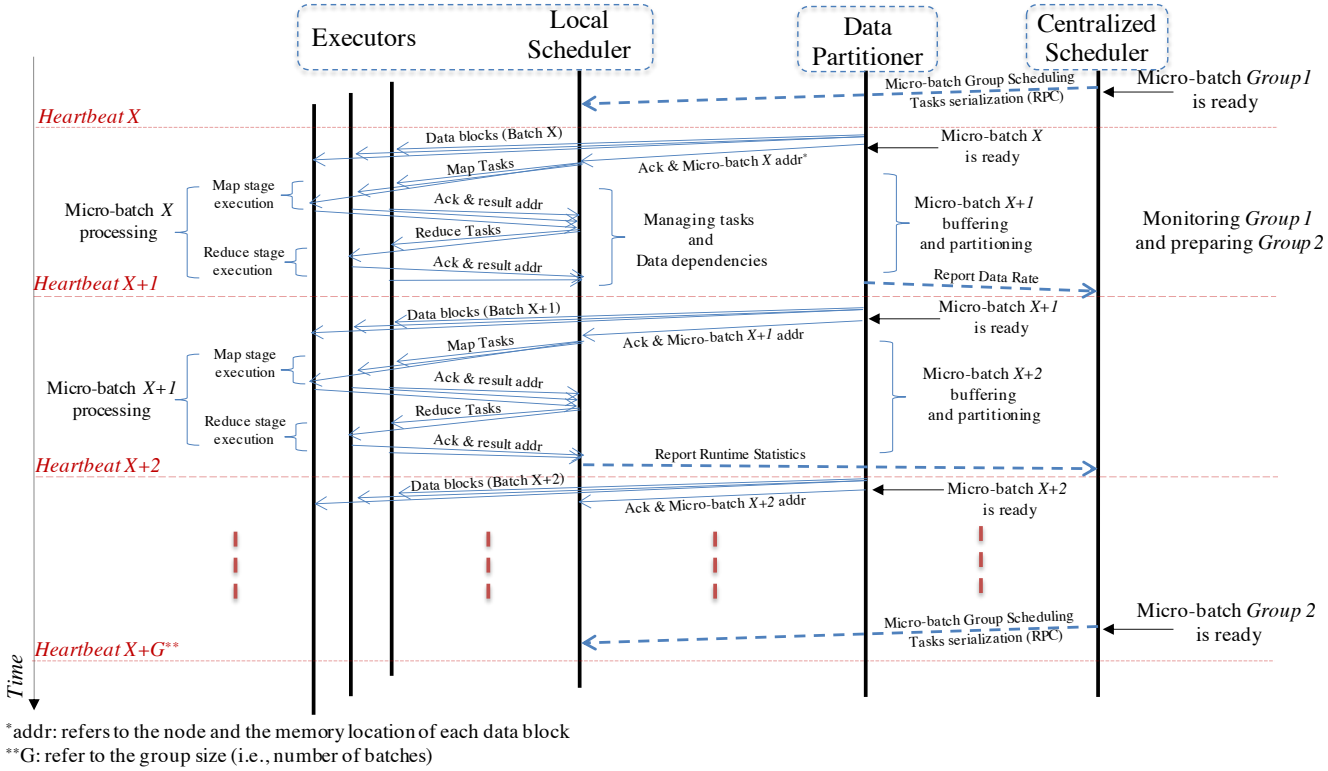


Figure 3.3. Group-Ahead Scheduling

3.4 Group-Ahead Scheduling

The objective of this technique is to eliminate the scheduling and communication overheads discussed in Section 1. The centralized scheduler makes the same scheduling decision (i.e., execution graph) for a consecutive group of micro-batches. Figure 3.3 shows the technique in action. There are subtle differences from the normal activity introduced in Section 1.2. The tasks for a group of batches are sent to the worker nodes using the same

message. Each worker node employs a local-scheduler process to queue its tasks, and keeps track of their dependencies (e.g., data dependencies between the mappers and the reducers). The data partitioner inform local scheduler of the whereabouts of the data-blocks of the new batches. In addition, the centralized scheduler ensures that the execution is not interrupted because it schedules the next group of batches ahead of time (i.e., before the end of the execution of the current group). To enable elasticity decisions, local schedulers and workers send periodic runtime statistics to the centralized scheduler. For instance, the data partitioner shares information input data rates and key distribution, while the local scheduler report the ratio of the micro-batch intervals and processing times. These messages are sent asynchronously and does not interrupt the data processing. Figure 3.4 illustrates the proposed scheduling scheme in action. At Time t, the scheduler starts a group of micro-batches using the *Group-Ahead* approach. While the micro-batches are being processed, the scheduler monitors the relationship between the processing time and the batch interval. If the relationship satisfies the stability condition (e.g., as in Zone 2), the scheduler will not intervene until the group is about to end. We refer to that time as the *scheduling window*. At this time, the scheduler will start a new group using the same query execution plan. Notice that the scheduling window is a parameter that is defined to safely allow the worker nodes to receive the newly scheduled tasks before the current group ends, and hence avoids interruptions. At any point in time, if the scheduler decides to elastically add or remove tasks in the execution plan, it will update the current group immediately. For example, in Figure 3.4 the scheduler changes the execution plan (and hence the schedule) of Group $t+1$ after submitting the entire schedule to the nodes. At Time ①, the scheduler sends Group $t+1$ to workers. At Time ②, the scheduler decides to change the group's execution plan (i.e., change the number of map or reduce tasks). The effect of this change will take some time until it takes place. We refer to this time by the *adjustment window*. The adjustment window allows the workers to update their task queues without interrupting the current micro-batch computation. At Time ③, the effect of the new schedule update takes place.

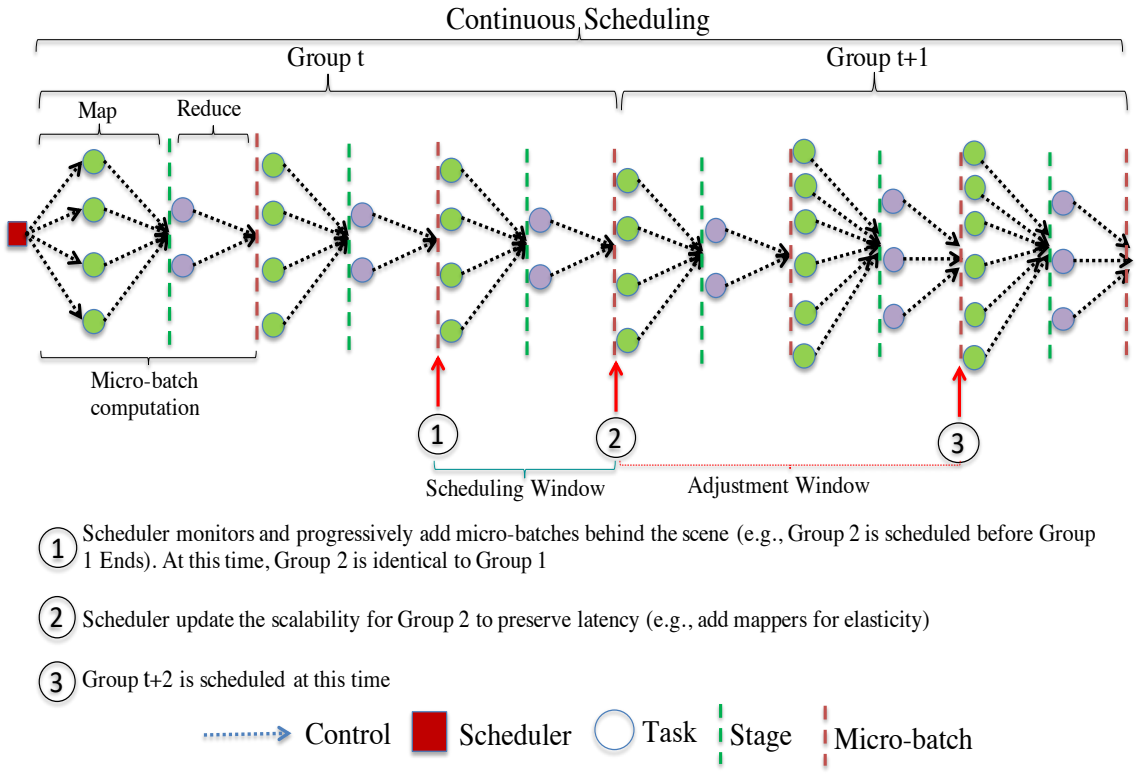


Figure 3.4. Elastic Scheduling Strategy

3.5 Elastic Scheduling

The objective of this technique is to enforce latency requirements of users' applications. Prompt+ seeks to meet the latency requirement using minimum resources. Figure 3.2b 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 number of 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 keep the execution engine 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 2.4.1). 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%)

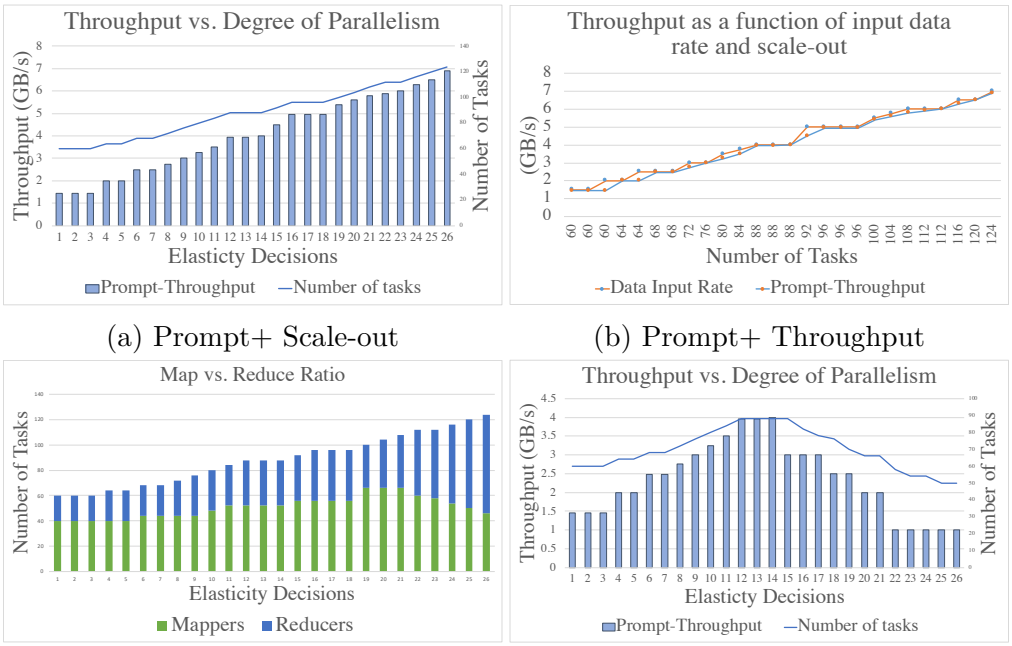
Output: $New\ Execution\ Plan : p\ and\ r$

```
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;
```

3.6 Experimental Evaluations

3.6.1 Experimental Setup

We use the same experimental setup illustrated in Section 2.6.1.



(a) Prompt+ Scale-out

(b) Prompt+ Throughput

(c) Prompt+ Map/Reduce

(d) Prompt+ Scale-In

Figure 3.5. Prompt+ Elasticity

3.6.2 Experimental Results

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 3.5a 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 3.5b illustrates how Prompt+ responds swiftly to the increase in workload by adding more execution tasks. Notice that when Prompt+'s throughput matches the input rates, it maintains its stability and provide latency guarantees (Figure 3.2a). Figures 3.5c and 3.5d 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.

Elastic Scheduling. We assess Prompt+'s elastic scheduling capability and responsiveness to workload changes. Figure 3.7a shows the effect of applying the group-ahead schedul-

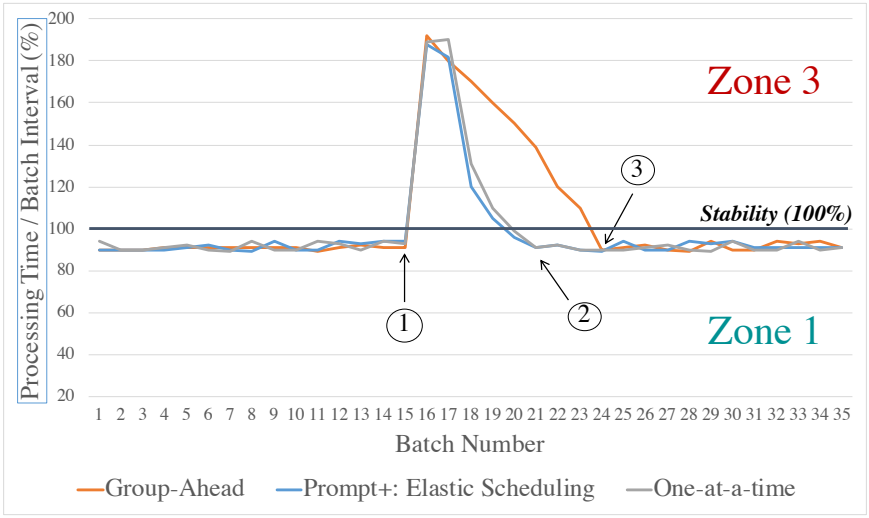
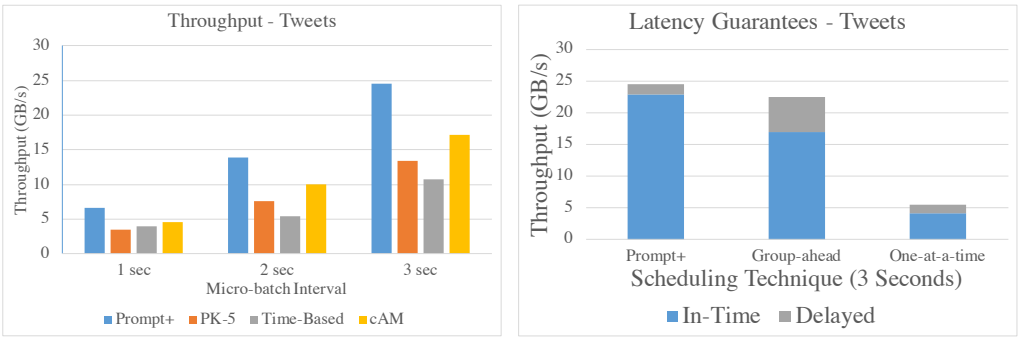


Figure 3.6. Prompt+'s Elastic Scheduling

ing strategy on system's throughput. It delivers significant increase in system throughput (i.e., up to 5x) regardless of the partitioning technique adopted. Figure 3.6 demonstrates Prompt+'s responsiveness to workload changes, compared to group-ahead scheduling and Spark's one-at-a-time scheduling strategies. In this experiment, we set the group size to 10 batches, e.g., $1-10$, $11-20$, $21-30$, etc. The batch interval is set to 1 second. Initially, each technique is provided with the maximum data rate it can sustain (i.e., maximum throughput achieved while maintaining the stability condition). We introduce a load spike at the middle of on of the groups (e.g., Batch 15 ①), where the amount of data is significantly increased for this batch (i.e., up to 2x). This is quite in the middle of the group of Batches $11-20$. All the techniques are fixed to restore the stability as fast as possible. For each technique, we report the ratio of Processing time vs. the batch interval. This ratio is a measure of how the system adheres to the stability condition while applying the scheduling technique. Prompt+'s elastic scheduler adjusts the running query execution-plan where this adjustment starts to show its effects during the execution of Batch 17. As a result, the stability condition is quickly restored by the time the system is processing Batch 20. Meanwhile, group-scheduling makes the change at the end of the group, and starts a the new adjusted plan at Batch 21 ②. Hence, it takes longer to restore the stability condition at Time ③.



(a) Group-Ahead Scheduling (b) Latency Guarantees

Figure 3.7. Group-Ahead Scheduling and Latency Guarantees: (a) System's throughput when applying Group-Ahead scheduling, (b) Latency Guarantees when applying Prompt+

More importantly, notice that Prompt+ is comparable to native Spark Scheduler responsiveness in which the scheduler changes the query plan immediately after the induced Spike. Figure 3.7b shows the effect of applying Prompt+ on latency guarantees. The percentages of times the latency is violated is much less than the case with group-ahead or one-time scheduling. Prompt+ enables significant increase in system's throughput and latency enforcement, while maintaining resource utilization.

3.7 Concluding Remarks

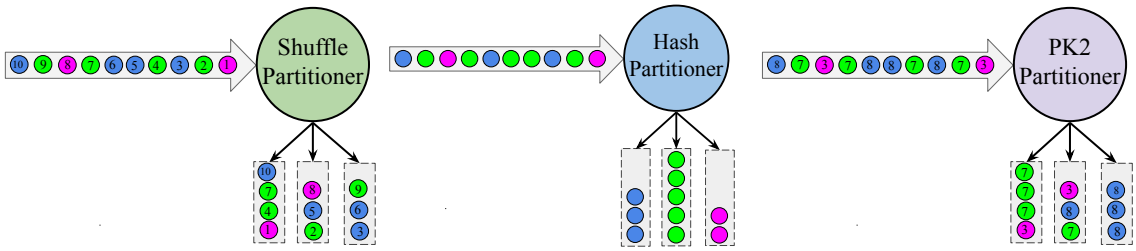
This chapter presents Prompt+, an elastic scheduling technique for the micro-batched stream processing systems. Prompt+ enforces latency by dynamically adjusting the degree of parallelism of the micro-batch computations. It uses asynchronous updates to avoid interruptions of data processing. Prompt+ defines three elasticity zones to maintain robustness against workload spikes.

4. PARTLY: LEARNED DATA PARTITIONING WITH DEEP REINFORCEMENT LEARNING

4.1 Problem Statement

Data partitioning plays a critical role in distributed data stream processing systems. The basic and well-known partitioning strategies are *shuffling*, and *hashing*. Shuffle partitioning assigns data tuples in round-robin to processing operators based on the order of arrival (Figure 4.1a). Shuffle partitioning guarantees that all processing operators receive almost equal number of data tuples. However, it does not insure *key locality*, i.e., tuples with the same key are not necessarily sent to the same processing operator, thus increasing the overhead of transferring data to compute per-key aggregates, e.g., [47]. In contrast, hash partitioning, also termed *Key Grouping* [47], applies a hash function over one or more fields of each tuple, i.e., a *partitioning key*, to route the tuple into a processing operator (Figure 4.1b). Hash partitioning assigns tuples with the same keys to the same processing operator. When the input data stream is skewed, some key values will appear more often than others. Thus, hash partitioning would result in unbalanced input to the processing operators.

State-of-the-art stream data partitioning techniques apply heuristics to achieve the benefits of both the shuffling and the hashing techniques while minimizing their drawbacks. One example heuristic is to split the skewed keys over multiple operator instances, e.g., [32,46,47]. (see Figure 4.1c). The data partitioner applies multiple hash functions to the tuple's *partitioning key* to generate multiple candidate assignments for the data tuple. Then, the partitioner selects the operator instance with the least number of tuples at the time of the decision. To realize this objective, the partitioner maintains two statistics in real-time: (1) The number of tuples and distinct keys assigned to each operator instance, and (2) Counts on the input data distribution to detect the skewed keys and split them. The partitioner may need to determine heavy hitters (i.e., most common key values) or apply periodic sorting [2, 46]. These bookkeeping statistics and operations can overwhelm the data partitioner memory and pose a risk on the stability of the streaming engine [32] (e.g., when applied to millions of keys in long-running topologies). These partitioning techniques rely on static heuristics, and do not learn from past experiences, i.e., the good or the bad decisions.



(a) Shuffle Partitioning (b) Hash Partitioning (c) Partial Key Grouping [47]

Figure 4.1. Traditional Data Partitioning Techniques: simple and static heuristics

In this chapter, we envision the use of deep reinforcement learning to partition data in distributed streaming systems. The use of artificial neural networks can facilitate the learning of efficient partitioning policies that match the dynamic nature of streaming workloads. Furthermore, we showcase *PartLy*, a proof-of-concept learned data-partitioner for streaming engines. Evaluations show *PartLy*'s potential to match the performance of state-of-the-art techniques in terms of partitioning quality while minimizing bookkeeping overheads. We identify potential challenges when applying machine learning techniques to the data partitioning problem in distributed data stream processing. Furthermore, we incorporate new multi-agent learning concepts that can enable learned data-flow optimization for distributed streaming systems. These concepts include state-space partitioning, cooperative multi-agents, and distributed learning.

4.2 Related Work

Deep Reinforcement Learning (DRL) has been applied successfully to solve challenging problems in computer systems including network routing [55], and memory caching [57]. Furthermore, DRL has been applied to learn policies for different optimizations in distributed database processing. Examples include query optimization [42], data indexing [58], and task-scheduling [41]. Recently, Yang et al. [58] and Hilprecht et al. [28] propose to learn data partitioning for relational and cloud-based databases. To the best of our knowledge, PartLy is the first to envision a stream data-partitioner using deep reinforcement learning. Furthermore, recent research, e.g., [28] and [58], apply DRL for data partitioning in relational and cloud-based databases.

4.3 Deep Reinforcement Learning

DRL is a machine learning technique that enables an agent (e.g., a neural network) to learn a task by trial-and-error. The agent learns from its own actions, and experiences through continuous interactions with a defined environment (Figure 4.2). The environment, action space, and reward calculation mechanism are predefined. Training is carried over several training episodes until a target convergence is achieved. In each episode, the environment informs the agent of its current state, s_t, and the potential actions $a_t = \{a_0, a_1, ..., a_n\}$ that the agent can choose from. The agent executes an action $a \in A_t$, and the environment responds to the agent with a reward r_t. The environment provides the agent with a new state s_{t+1} and a new action set a_{t+1} that reflects the status after its recent action. This process repeats until a terminal state is reached (i.e., where no more actions are possible. This marks the end of a training episode after which a new episode may begin. The objective of the agent is to maximize the reward over episodes by learning from its previous actions. The training objective is to adjust the weights θ of the neural network to learn a policy $\boldsymbol{\pi}_\theta$ that achieves the maximum reward. Various training methods include [18]:

Deep Q-Learning: Deep Q-Learning [27] is a widely used reinforcement learning method. Q-learning leverages an action-value function to define a policy $\boldsymbol{\pi}_\theta$ as $Q^{\boldsymbol{\pi}}(s, a) = E[R|s^t = s, a^t = a]$. It recursively learns an action-value function Q^* to approximate the optimal policy. To facilitate training, Deep Q-Learning uses a *Reply Buffer* to accumulate tuples of the form: (s, a, r, s) that represent the current state s, the action a, the reward achieved r, and the new state s. The Reply Buffer is used to periodically update Q^* with the most recent learning parameters θ to stabilize the training.

Policy Gradients: This method is commonly used in a variety of DRL problems [18]. It smoothly adjusts the policy parameters θ in the direction that maximizes the training objective (i.e., the reward). Since drastic changes can cause the policy to fluctuate without converging, policy gradient methods (ascent or descent) tune the policy parameters θ by increasing each parameter by a small value if the gradient is positive (i.e., the positive gradient indicates that a larger value of θ will increase the reward). Similarly, decreasing the parameters a small value when the gradient is negative.

Actor-Critic: Actor-Critic [34] is a variant of policy gradient methods. Instead of continuously updating the agent's policy in each training episode, a neural network, referred to as the critic, decides when it is beneficial to update the agent policy using the experience of the recent training episode (i.e., actions and rewards). The critic limits possible inverse training by deciding whether or not the experience of a training episode will improve the agent's performance. This allows the agent to explore the decision space without negatively affecting its policy weights. Both the agents' policy and the critic are approximated by deep neural networks.

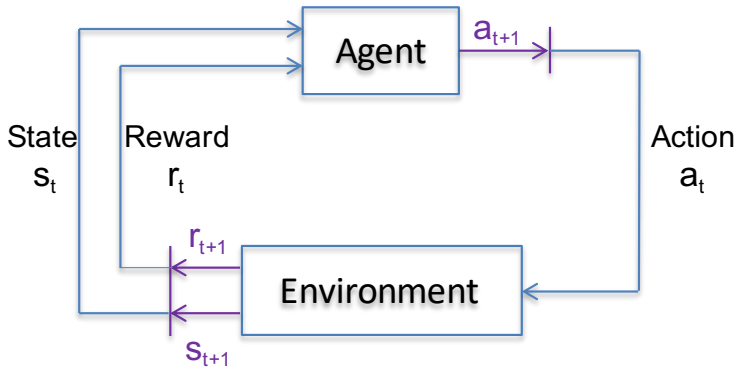


Figure 4.2. DRL agent interacts with the environment

4.4 Learned Data Partitioning

The real-time execution in data stream processing systems requires the partitioning technique to make a swift per-tuple decision upon tuple arrival. Otherwise, data partitioning may lead to performance bottlenecks by increasing end-to-end tuple processing times. In addition, the input data rate is typically in the order of millions of tuples per second (e.g., see [54]). Processing individual tuples through a neural network in real-time is challenging. One possible solution is to use micro-batched stream processing (e.g., [61]) to amortize the cost over a batch of tuples. In contrast to tuple-at-a-time stream processing, the partitioning decision is taken collectively for a group of tuples that are buffered within a batch. Hence, the data tuples are assigned to data blocks, and consequently, each data block is assigned to a processing node. Furthermore, the data partitioner can operate on the key value of tuples

within a batch (i.e., one decision is given to all tuples sharing the same key value within a micro-batch).

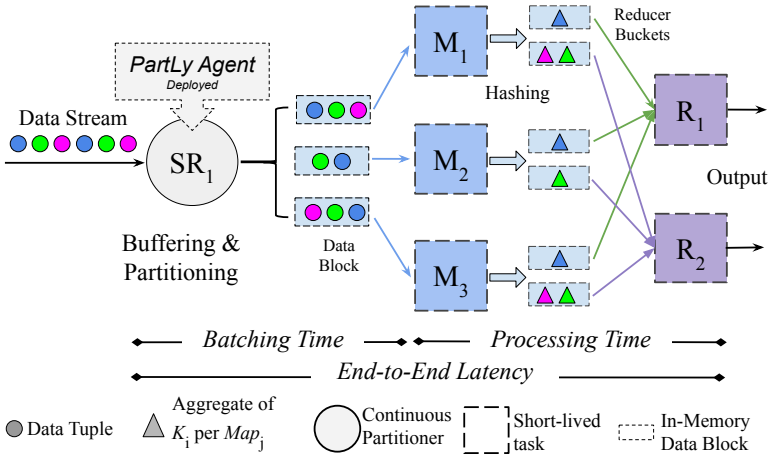


Figure 4.3. Example of micro-batch stream processing with three Map and two Reduce tasks, and a Stream Receiver (SR_1) with PartLy to partition a micro-batch into data blocks.

We present a case for applying DRL to solve the problem of data partitioning in stream processing systems. PartLy leverages a single agent and can be deployed to an instance of a data-partitioner process in a streaming topology. PartLy shows how the streaming data-partitioning can be modeled in the context of DRL. We discuss the design of PartLy, how states are represented, and the training procedure:

4.5 State Representation

PartLy applies vectorization to model the intermediate states of the training episodes. The vectors hold information about the batched data and the progress of assignment to data blocks. Each non-terminal state represents a partial assignment of the data keys to data blocks. Each vector is a row of size n, where n is the number of unique keys in the batch. The assignment of keys to data blocks is captured using a matrix M of size $n * m$ for each episode, where m is the number of data blocks. For example, the value M_{ij} is 1 if k_i is fully-assigned to $Block_j$. This value is 0 if no tuples of k_i are assigned to $Block_j$. $M_{ij} = 0.5$ if k_i is equally-split across two data blocks. Figure 4.4 gives a generic representation of a training episode in PartLy.

4.5.1 Reward Design

DRL algorithms require a performance metric to use as a reward signal. The variance of latencies in task-execution (i.e., downstream operator instances) is a straightforward representation of the balancing quality of the data partitioning policy. However, latency variance does not offer a dense reward signal. The agent's training is efficient if it is able to receive a reward signal as it navigates the environment (i.e., each action in a training episode receives a partial reward). Moreover, calculating the latency variance requires the actual execution of the partitioned data stream. This poses an overhead on the training and it can be prohibitive to arrive to convergence. In fact, in this case, the reward calculation is more time-consuming than traversing a training episode by the agent. One possible approach to resolve this is the use of traditional data-partitioning cost models [32, 46, 47] to estimate the reward signal. These cost models can be adopted to offer both; partial rewards for the agent actions during an episode and a final reward signal at the end of the episode. For instance, PartLy adopts the PK2's [47] cost model when using the number of tuples assigned to a processing node to calculate the final reward for the training episodes. After the agent has trained an acceptable policy using a cost model, it can be deployed in a streaming topology.

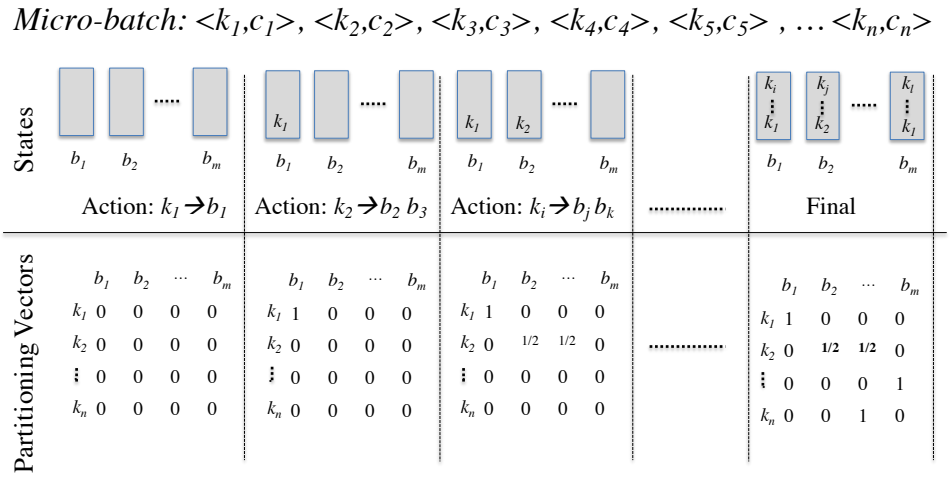


Figure 4.4. Actions in PartLy training episode

4.5.2 Decision Space

Efficient processing of data streams entails various optimizations. Even with the use of batching, the number of possible assignments of data tuples within a batch to processing nodes is exponential, i.e., M^K, where M is the number of processing nodes and K is the number of distinct keys. One solution is to limit the exploration of the learning-agent to a solution known to be efficient. For example, in earlier work [46, 47], partitioning of a skewed key value to only more than five processing nodes does not contribute to balancing the load over the workers (as in [47]). Applying this strategy alone can reduce the decision space to the range of $2*K*M$ - $5*K*M$. In general, increasing the possible optimization actions have a significant effect on the training convergence time. Optimization tasks include deciding the granularity of keys partitioning over the processing nodes (i.e., splitting a key equally over nodes or with variable percentages).

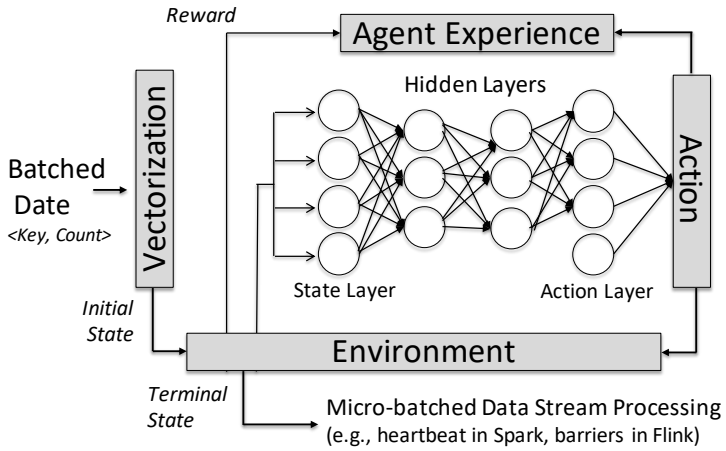


Figure 4.5. Design of PartLy

4.6 Training Process

PartLy uses deep reinforcement learning, where an agent interacts with the defined environment (See Figure 4.5). The environment informs the agent of its current state, s_t, and the set of potential actions $A_t = \{a_0, a_1, ..., a_n\}$ that the agent can choose from. The agent executes an action $a \in A_t$, and the environment responds to the agent with a reward r_t. The

environment provides the agent with a new state s_{t+1} and a new action set A_{t+1} that reflects the status after the recent action. This process repeats until a terminal state is reached (i.e., when no more actions are available to execute). This marks the end of an episode after which a new episode may begin. The objective of an agent is to maximize the reward over episodes by learning from the agent's previous actions. PartLy treats every batch of data as an episode and learns continuously over the multiple batches. PartLy uses a policy gradient method to select actions based on Policy π_θ (i.e., neural network), where θ is a vector of policy parameters. The policy π_θ is optimized over episodes by modifying its parameters θ (i.e., the neurons' weights) to generate the best reward. PartLy uses the cost model of the state-of-the-art techniques in [2, 32, 46, 47] to compute the rewards of episodes. The cost model relies on checking the difference in sizes and cardinality between the maximum and average data blocks. The agent's objective is to minimize this difference by maximizing the reward. For example, in the case of the Pk2-partitioner [47] the reward is set to the negative of the difference between the maximum size and the average size of all generated data blocks for the batch: $max|Block_{\mathrm{i}}| - avg|Block_{\mathrm{i}}| \quad \mathrm{i} \in p$, where p is the number of data blocks. Notice that, when the agent is trying to generate even-sized data blocks using the cost model of PK2 [47], the action space only allows a key to be assigned to one data block or split over two data blocks, and thus preserving key-locality to two data blocks in the worst case. In other words, the action space of the agent is defined according to the adopted cost model of the partitioning technique. Figure 4.5 gives an overall view of PartLy. The micro-batch statistics (i.e., the list of key counts) are vectorized and are inserted into the state layer. The statistics are collected using an online technique while buffering the data tuples similar to the one discussed in Chapter 2.4.1. Count values of the distinct keys are transformed and are passed to hidden layers, and finally to the action layer. We apply the following techniques during the training of the agent. These techniques significantly reduce the number of training episodes required for convergence, and provide performance enhancements:

(1) Cost-Model Bootstrapping: We apply cost-model bootstrapping [43] by assigning partial rewards to the agent's actions using the Prompt's cost model for data partitioning presented in Chapter 2.3.3. The cost model provides an indication of the balanced partitioning using the variance of data-blocks sizes and cardinality (i.e., number of unique keys). The

partial rewards are defined in the following way: For each action in the training process, we assign a positive reward if the assignment action (i.e., split or assign) does not violate the following criteria:

- **Cardinality Imbalance**: The number of unique keys will remain constant over the data blocks given this action. Note that the number of unique keys per data block is known and is calculated as the number of keys per batch divided by the number of data blocks.
- **Size Imbalance**: The data block(s) affected by the agent's actions will not exceed the expected maximum capacity. Note that each data block capacity is estimated as the size of the batched data divided by the number of data blocks.

(2) Incremental Learning: During the training process, we apply incremental training by:

- Gradually increasing the number of possible key splits over data blocks from two to five partitions.
- Gradually allowing a different magnitude of splitting ratios (50%, 25%, 12.5%).

For instance, we start by allowing the model to only split keys equally over data blocks, then enrich the action space by adding the ability to split equally over up to five partitions. When the model starts to show convergence, we add the ability to split with different ratios over the five data blocks.

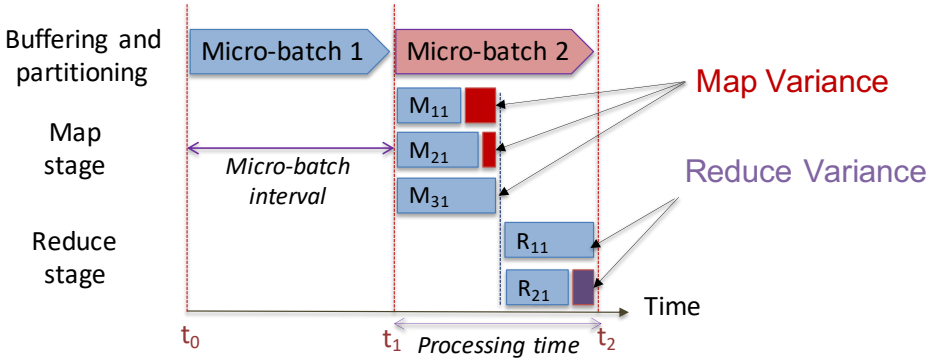
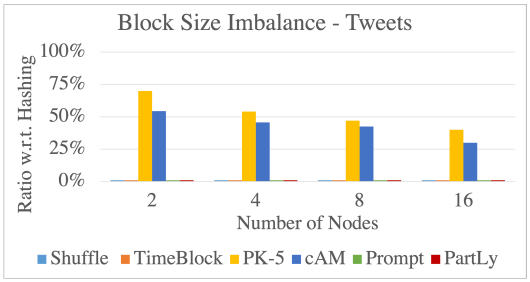


Figure 4.6. Execution Latency Variance as a Reward

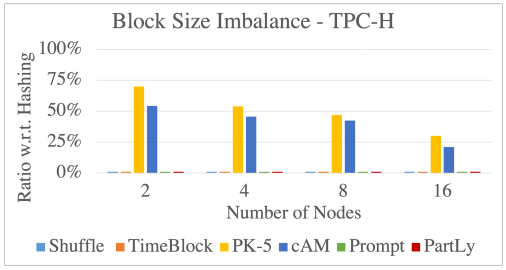
(3) Learning with Run-time Statistics: We use the variance of execution-times (i.e., latency) of the MapReduce tasks to compute the reward signal (See Figure 4.6). The reward signal can be only used for the terminal state of the training episode when the partitioning is complete and the batch is ready for processing. Initially, the agent is trained using cost-model bootstrapping. When the model starts to convergence and performs near the cost model, the agents start to use the variance of the mappers and reducers as the reward signal for its training episodes. Note that: (1) Since we can only use one value as the reward signal, we select the higher variance (i.e., either mappers' or reducers' variance). This strategy forces the model to explore actions that lead to decreasing the variance used as the reward. (2) Due to differences in the ranges of the values of the cost-model estimates and possible latency-variance, we use the equation below to convert the latency variance to the cost model range.

$$Reward = Cost_{min} + \frac{LatencyVar. - LatencyVar._{min}}{LatencyVar._{max} - LatencyVar._{min}}(Cost_{max} - Cost_{min})$$

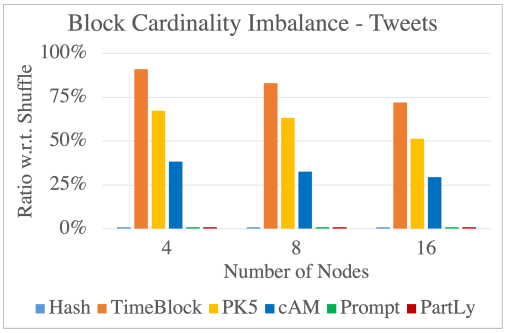
This conversion avoids possible inverse training by confusing the model with wrong reward signals.



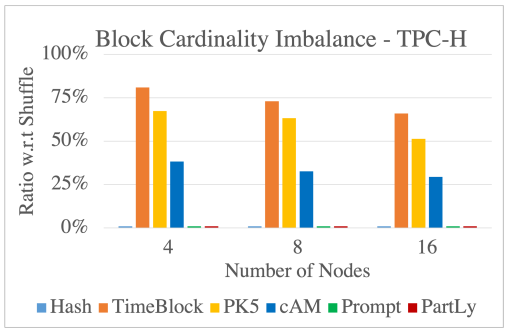
(a) Relative BSI: Tweets



(b) Relative BSI: TPC-H



(c) Relative BCI: Tweets



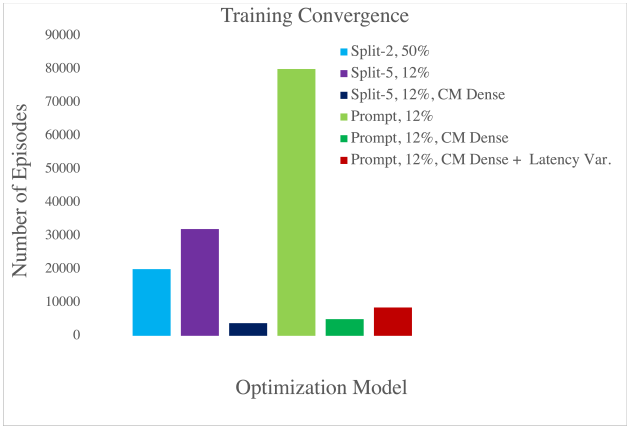
(d) Relative BCI: TPC-H

Figure 4.7. PartLy's Partitioning Quality

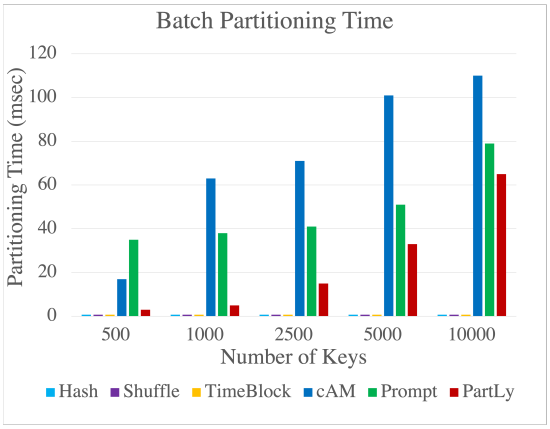
4.7 Experimental Evaluations

4.7.1 Experimental Setup

In each experimental setup, PartLy assumes a maximum number of keys. If the number of keys within a batch is less than expected, zero-padding is applied (i.e., counts are set to 0). The output of the action layer is normalized to form a probability distribution to allow for action selection. Unless a cost model is used, rewards are computed only for a terminal state, i.e., when all keys are assigned. In addition, if a cost model is used the final reward is computed using a running average over the previous episodes to reduce the randomness effect and promote generating a general policy. To train the model, PartLy uses the *Proximal Policy Optimization (PPO)* algorithm [20] within TensorForce [21]. Training takes the range of 5,000 to 100,000 simulated batches of data. In the experiments, we use the three data sets, Tweets, TPC-H, and GCM (Refer to Section 2.6.1 for more details). The queries perform a sliding window count a data stream of tuples. The queries are written in map-reduce. Experiments are conducted for an execution setup of 5 nodes with 8 cores each (i.e., the number of data blocks is 24). Apache Spark v2.0.0 is the processing engine. We generate batches with different numbers of keys (i.e., ranging from few hundreds of keys to thousands of keys).



(a) PartLy's Training Convergence

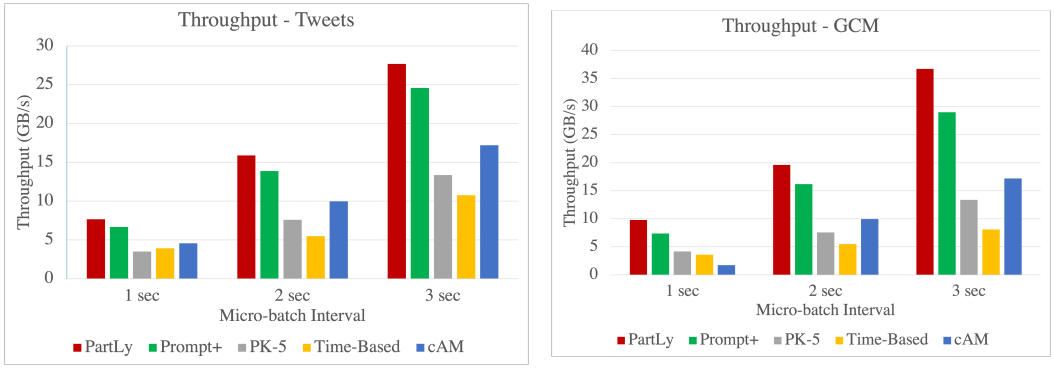


(b) PartLy's runtime

Figure 4.8. PartLy's Cost

4.7.2 Experimental Results

We assess the effectiveness of PartLy against traditional and state-of-the-art techniques: *Shuffle*, *Hashing*, *PK-5* [46], *CAM* [32], and *Prompt* [2]. Figure 4.7 compares the *partitioning quality* metric for all the techniques. The partitioning quality is measured using two metrics: $BlockSizeImbalance(BSI)$ and $BlockCardinalitySize(BCI)$. The BSI defined as $max|Block_i| - avg|Block_i| \quad i \in p$ and is computed relative to the Shuffle technique. Note that Shuffle partitioning guarantees size balancing at the expense of broadcasting keys to all data blocks (i.e., potentially increased overhead at the compute nodes). Similarly, BCI is defined as $max||Block_i|| - avg||Block_i|| \quad i \in p$, and is computed relative to the Hash technique. We verify the partitioning strategies for all algorithms by feeding the generated data blocks to the Spark Streaming engine. PartLy demonstrates the ability to outperform state-of-the-art partitioning techniques when trained using an optimized-partitioning decision space.



(a) PartLy's Throughput - Tweets

(b) PartLy's Throughput - GCM

Figure 4.9. PartLy's Throughput

Figure 4.8a demonstrates the effect of the training technique used and the decision space (i.e., possible optimizations) on the convergence rate. The training technique and the provided action space have a profound impact on the number of training episodes necessary for convergence. For instance, using the cost-model bootstrapping to generate dense rewards leads to a significant drop in the number of training episodes necessary. The use of partial rewards for intermediate states in a training episode shortens the training time by up to 70%. Furthermore, the use of runtime statistics to improve the model training provides a

performance gain in throughput. Applying this training strategy achieves a 12% decrease in reduce-variance compared to Prompt. PartLy significantly improves the partitioning quality and reduces the split of keys by 38%. Figure 4.9 demonstrates PartLy's ability to maintain a competitive throughput while changing data distribution and batch intervals. Figure 4.8b gives the partitioning cost in terms of the required time to partition a micro-batch into data blocks. PartLy shows a potential to outperform CAM [32] and Prompt [2] in terms of speed as the number of keys increases.

4.8 Future Directions: The Multi-Agent Approach

In this section, we study how recent advances in multi-agent DRL techniques can be leveraged in the context of distributed data stream processing systems.

4.8.1 State-Space Partitioning

The data-partitioning problem can be formulated as a Markov Decision Process (MDP) [18], where at each state it is required to assign an item (i.e., either a data tuple or a key-value) to an existing or new data partition. The problem follows the Markov Property [18] where: (1) Each action requires the knowledge of the current state only, and (2) The current state contains the accumulative results of all previous actions since the initial state. However, formulating the distributed data-partitioning problem using one big space to optimize the data-flow through an entire topology is not practical (i.e., training convergence is not achievable).

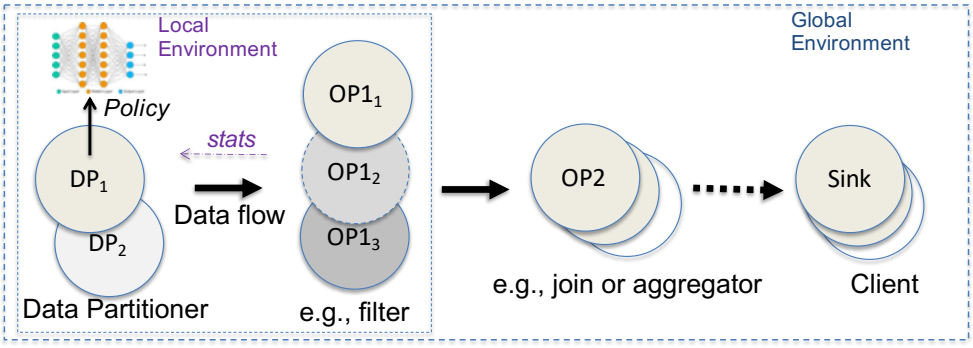


Figure 4.10. Localized Reward Shaping

Applying state-space partitioning with localized reward shaping is crucial to realizing a working solution. One example of state-space partitioning is to decouple the decision of the number of data partitions from the process of applying balanced data assignment. This approach chunks the complex problem into smaller solvable ones and paves the way to introduce multiple agents. It simplifies the state-space of the agents, and increases training efficiency (i.e., rate of convergence). Figure 4.11 gives an example, where a scheduler-agent cooperates with the data-partitioner agent through "advising" on updating the number of data partitions when necessary. Figure 4.10 demonstrates another issue of applying state-space partitioning to the data partitioning problem in a streaming topology. In this topology, two data partitioning agents are deployed, say DP_1 and DP_2. Assume that the average tuple latency of the topology is increased due to unexpected DP_2 behavior, while DP_1 achieves optimal load-balancing to its immediate downstream operators. One solution is to shape the rewarding of each agent to its local environment (e.g., by using the run-time and load statistics of the immediate downstream operators of a data partitioner instance). This enables $DP1$ to avoid triggering any modifications, and limits the retraining work to DP_2. Moreover, Both DP_1 and DP_2 will not get affected by any other factors that contribute to the average tuple latency of the entire topology.

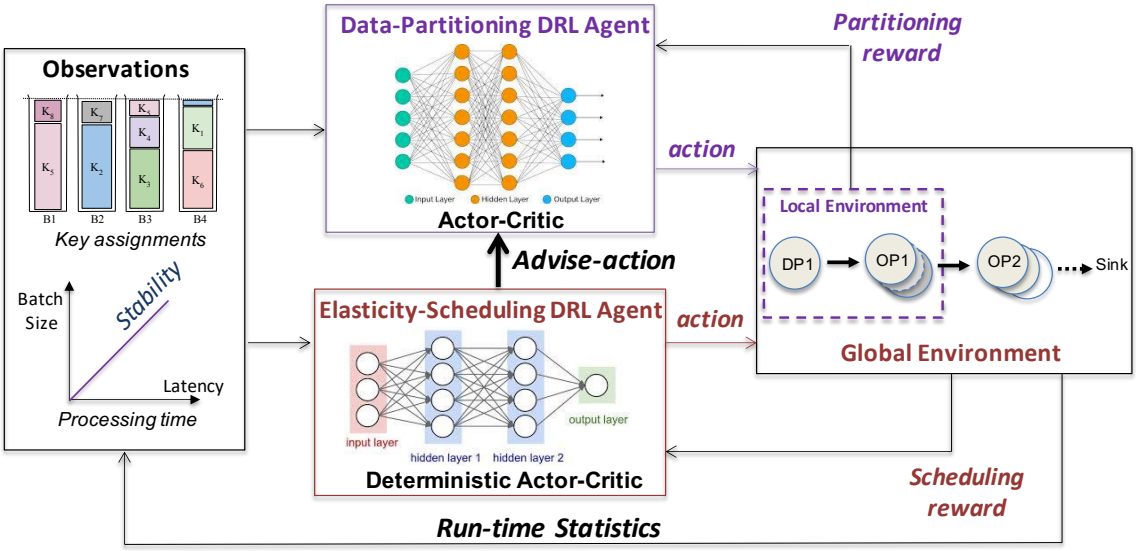


Figure 4.11. Proposed Multi-Agent Model

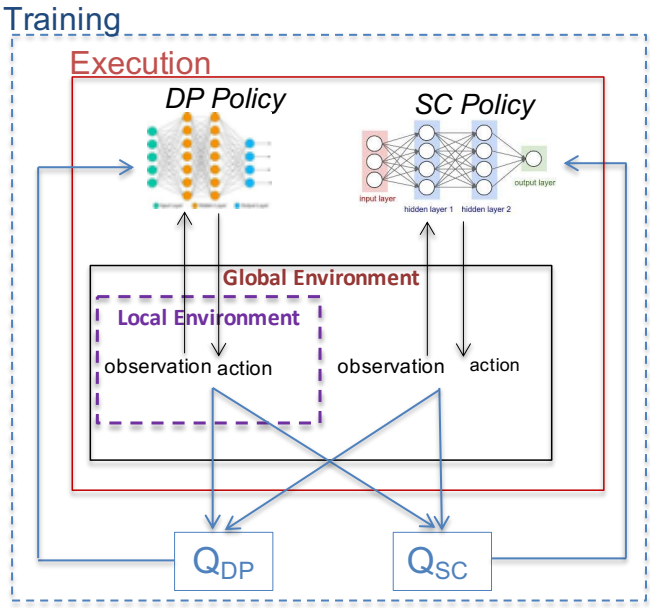
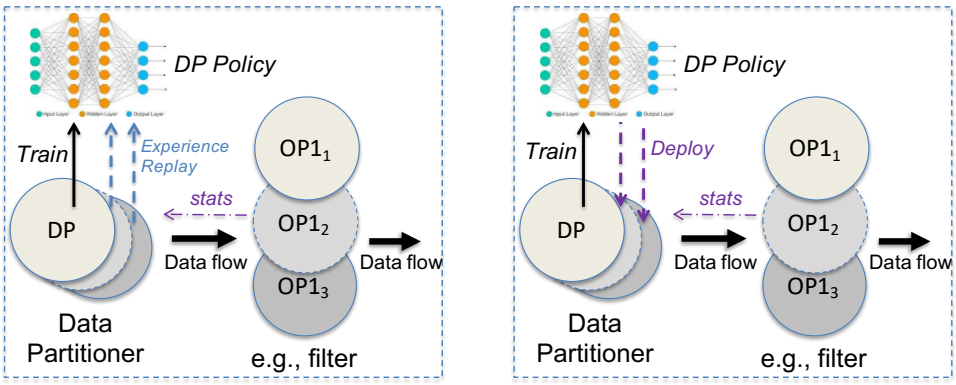


Figure 4.12. Actor-Critic Multi-Agent Training

4.8.2 Cooperative Multi-Agents

The data partitioning problem in distributed streaming topologies can be formulated in the context of *Multi-Agent DRL*. The problem can be formulated as a multi-agent extension of the traditional Markov Decision Processes (MDP), where a set of states $\mathbf{S} = S_1, ..., S_n$ represents all possible state-spaces of the agents (analogously, the set of action-spaces $\mathbf{A} = A_1, ..., A_n$ and the set of observations $\mathbf{O} = O_1, ..., O_n$ for all agent). Each agent independently leverages a policy π_i to choose the next action and produce its next state. The reward of each agent is defined as a function of its state and action. The environment provides each of the agents a private observation correlated with the agent's state and action. Each agent aims to maximize its own reward R_i. Figure 4.11 demonstrates the use of cooperative agents in the context of distributed micro-batched stream processing. Two types of agents are utilized. One centralized-scheduler agent (**SC**) and multiple data-partitioners agents (**DP**). All agents interact with the same environment. The centralized-scheduler learns an elasticity policy to advise on the number of data blocks, and uses system-wide observations, e.g., the relationship between batch interval and processing time. The DP agents learn a balanced data partitioning policy and use local observation, e.g., balance key-assignments over data

blocks. Furthermore, no assumptions are made on the communication pattern between the agents. For instance, the deployed data-partitioning agents may not communicate (e.g., DP_1 and DP_2 in Figure 4.10), while the centralized-scheduler agent might communicate with a data-partitioner agent to advise on the number of data partitions (Figure 4.11). The *actor-critic policy gradient* methods can be extended to facilitate agents' training in this setup [19]. In this training scheme, each agent's critic is enriched with the policies of other agents. Figure 4.12 demonstrates the training algorithm in action. An agent can use additional information of the other agent policies during training, but it cannot be used at execution time (i.e., test time). The learned policies of the DP agents can only use local information at execution time (i.e., observation from the immediate local environment). The critics serve to avoid possible inverse training of the agents. For instance, the data-partitioner agent must be aware of the centralized-scheduler agent involvement if the load is beyond current resources (i.e., the partitioning policy is balanced). Otherwise, the DP agent will mistakenly explore other partitioning policies. Similarly, the training of the scheduler agent should not get negatively affected by the data-partitioner decisions. The scheduler needs to be aware when the data-partitioner must take responsibility for the current increase in average tuple latency (e.g., when it is due to unbalanced partitioning). Algorithm 5 provides an example of applying this training scheme.



(a) Decentralized Training (b) Centralized Training

Figure 4.13. Distributed Learning Techniques

Algorithm 5: Distributed Multi-Agent Training

Input: M: Max. number of training episodes, N: Number of Agents, P: Random Action Exploration Process, B: Replay Buffer, K: Max Batch-Size-Keys
Output: *Trained N Agents*

```
For i =1 to Max-Number-Episodes(M);
    s<– Initial State;      P<– Initial Action Exploration Process;
    For k= 1 to Max-Episode-Length (K);
        Foreach agent i ∈ N do;
            Choose action a_i according to current policy π_i ;
        Execute actions a_1, ..., a_N;
        obtain reward r and new state s;
        Add new tuple (s,a,r,s') to Replay Buffer(B);
        set new state s‘ in s;
        Foreach agent i ∈ N do;
            Sample random batch from B: (s,a,r,s');
            Update critic_i by minimizing the Loss;
            Update actor_i using policy gradient;
        Update target network parameters for all agents i ∈ N;
    end for;
end for;
```

4.8.3 Distributed Learning

The real-time processing and the online-execution of streaming workloads requires efficient and fast training of the agents' policies. One possible approach is the distributed-mode execution of the training through parallelization. We propose two execution schemes that offer a trade-off between training convergence-speed and training overheads when learning a data partitioning policy:

(1) All-at-once: In this scheme, the partitioning policy is trained by all operator instances. Figure 4.13a gives an example. All the DP instances (e.g., Spouts in Storm) in the streaming topology provide training episodes information necessary to update the learned policy (i.e., experience reply buffer). This scheme is centralized training but decentralized execution. It serves to minimize the training time as much as possible.

(2) One-at-a-time: In this scheme, the data partitioning policy is only trained at one operator instance until convergence, while other operator instances use a heuristic approach. Figure 4.13b gives an example, where a single DP instance is used to train the data partitioning policy. When the learned policy converges and outperforms the heuristic, it is

deployed to other DP instances. This is an example of centralized training and execution. The objective is to limit the disruption in the execution of the streaming topology that can be caused by the training. Algorithm 5 is centralized training with decentralized execution, which is *One-at-a-time*. It can be easily altered to an *All-at-once* version.

4.9 Concluding Remarks

We envision that DRL can automate the operations of stream processing engines. We showcase PartLy that demonstrates a potential for applying DRL to the data partitioning problem in distributed stream processing. Data partitioning is the starting point to apply DRL to streaming engines. However, the wide adoption of DRL techniques across the streaming engine internals can significantly improve the performance and operating costs. It opens the possibility of autonomous execution and reducing the overheads of tuning heuristics by humans. We show how the nature of distributed stream processing provides hurdles for the direct application of DRL techniques. We identify some of the major challenges and show how recent advances in multi-agent DRL provide exciting research directions for addressing these challenges.

5. REALIZATION IN DISTRIBUTED MICRO-BATCH STREAM PROCESSING SYSTEMS

In this chapter, we present the techniques of Prompt, Prompt+, and PartLy as a generic abstraction that can be adopted in Distributed Micro-batch Stream Processing Systems. We demonstrate how these techniques can be realized in three widely used stream processing engines namely, Apache Spark [61], Apache Flink [12] and Apache Storm [48]. Figure 5.1 depicts a five-technique abstraction applied to the generic concept of the micro-batch computational model. Using this abstraction, we discuss how the proposed schemes can be realized to optimize windowed-aggregates streaming queries in these three systems when a batching execution model is used.

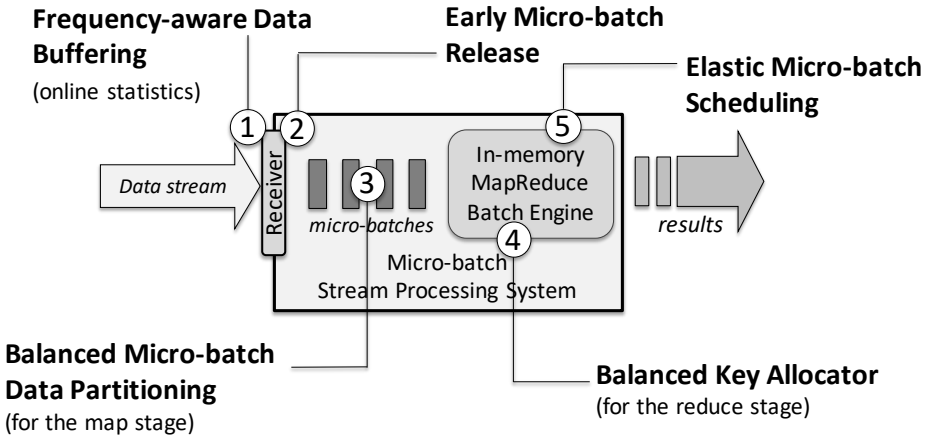


Figure 5.1. General Abstraction

The first two techniques, namely *Frequency-aware Buffering* and *Early-Batch-Release*, facilitate data partitioning. *Frequency-aware Buffering* collects data statistics to facilitate the subsequent partitioning step. In contrast, *Early-Batch-Release* makes time for applying optimized data-partitioning without violating the synchronization between the subsequent batch computations. The third and fourth techniques, namely *Balanced Micro-batch Data Partitioning* and *Balanced Key Allocator*, are responsible for partitioning the batched data and the intermediate query results, respectively. The last technique, namely *Elastic Micro-batch Scheduling* is responsible for enforcing latency by dynamically varying the number of execution-tasks in response to variation in the workload. The realization of PartLy include

the first two techniques. The first four techniques represent Prompt while the fifth technique represent Prompt+. We use this abstraction to demonstrate how to the proposed optimizations in the studied systems. For each of the systems, we present a brief overview of the system's architecture, and then discuss how the proposed scheme can be realized within that system's architecture.

5.1 Spark Streaming

5.1.1 Architecture of Spark

Spark Streaming [61] is a native micro-batch stream processing engine. It operates by dividing the continuous data stream tuples into batches that are then processed by the Spark data processing engine [60]. The results are generated in a stream of batches. Its in-memory data abstraction, *DStream* (short for a Discretized Stream), represents a stream of data tuples that is divided into small batches. The DStreams are built over the Spark's main data type (i.e., the Resilient Distributed Datasets (RDD, for short) [59]). Figure 5.2 demonstrates the batching and execution cycles within the Spark Streaming engine. A *Data Receiver* process buffers the input data stream for a predefined batch interval. The Data Receiver partitions the batch into smaller data blocks. The *Block Manager* stores the data blocks in the memory of the worker nodes, and informs the master node of the block's address. The master node, termed the *Driver*, continuously runs administrative processes to manage the execution.

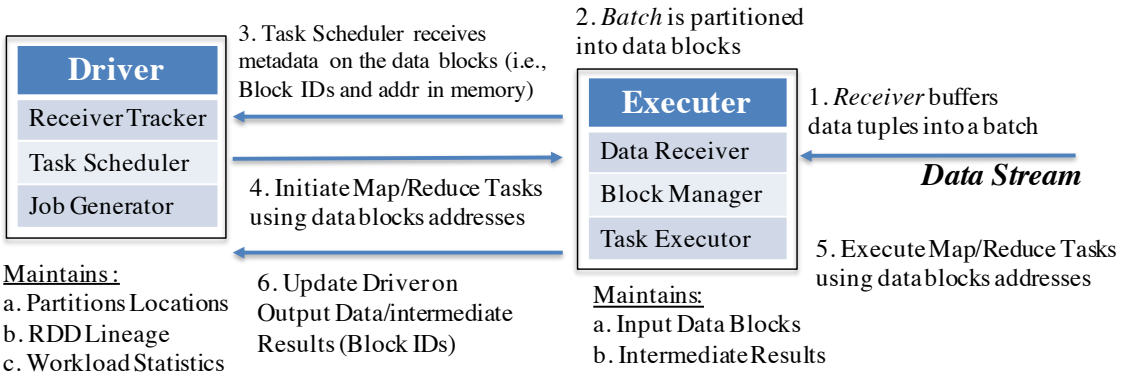


Figure 5.2. Data Flow in Spark Streaming

The *Job Generator* is responsible for generating the execution graph (e.g., see Figure 1.1). The *Receiver Tracker* keeps tracks of the addresses of the data blocks stored in the workers' memory. It also holds the data dependencies between the subsequent batches and the intermediate results. This information is leveraged by the *Task Scheduler* to initiate Map-Reduce execution tasks over the data-blocks. The Spark Streaming engine relies on sending computation to data, and uses short-lived tasks to perform computation over the accumulated batches of the streaming data.

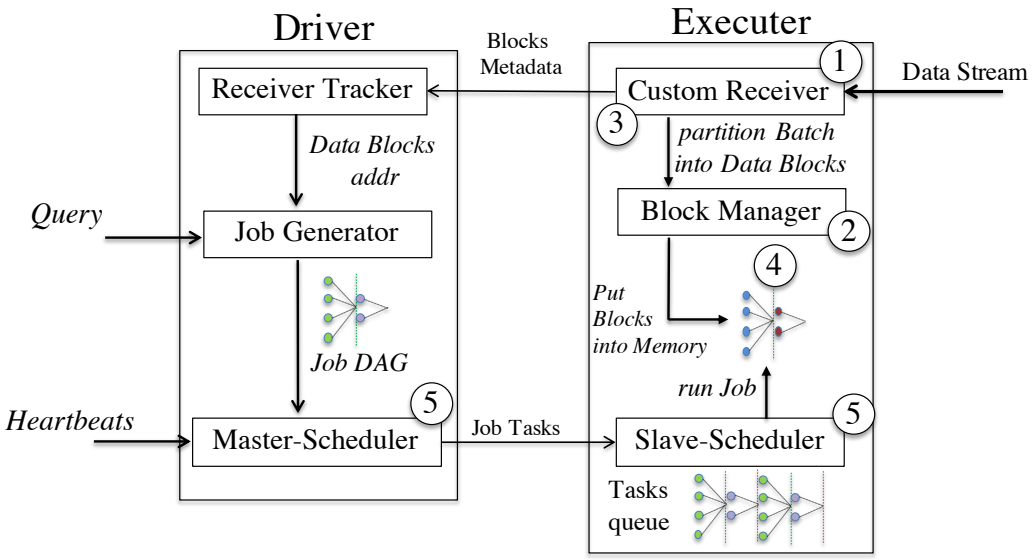


Figure 5.3. Realization in Spark Streaming

5.1.2 Realization in Spark Streaming

The proposed schemes are realized in Spark Streaming [61] by modifying four components. Figure 5.3 presents the realization of the generic abstraction in the Spark Streaming engine. *Frequency-aware Data Buffering* is implemented as a customized data receiver. The customized data-receiver ingests the data tuples and maintains the two data structures *CountTree* and *BatchedData*. The two data structures are leveraged to apply *Balanced Micro-batch Data Partitioning* at the end of the batch interval. *Early Micro-batch Release* is implemented in the Block Manager process. It maintains synchronization by sealing and serializing the data blocks and placing them in the memory of the worker nodes. The *Balanced*

Key Allocator is implemented in the shuffle phase of the mappers. The *Elastic Micro-batch Scheduler* is realized using the following two processes: (1) The *Task Scheduler* within the driver node, (2) a new process, termed *Local Scheduler* within each of the worker nodes. The first process is modified to allow for dynamically deciding the number of mappers and reducers for each micro-batch computation. The *Local Scheduler* enables group-ahead scheduling. It maintains the scheduled Map-Reduce execution tasks and executes them at the designated batch intervals.

5.2 Apache Flink

Apache Flink is a distributed data processing engine. It supports both tuple-at-a-time stream processing and batch processing. In this section, we illustrate Flink's execution model for data streams, and then we show how our generic abstraction can be realized in its streaming workloads.

5.2.1 Architecture of Flink

Flink expresses and executes streaming queries as pipelined dataflow programs. A Flink's dataflow program is a directed acyclic graph (DAG) that represents operators and their data dependencies. The basic data abstraction in Flink is *data streams*. A data stream represents the input and output of the processing operators. The data dependencies show how data streams produced by an operator are consumed by other operators. Flink's core runtime engine can be seen as a streaming dataflow engine. As depicted in Figure 5.4, Flink has three types of processes: *client*, *Job Manager*, and *Task Manager*. The client transforms the program code into a dataflow graph, and sends it to the JobManager. The JobManager schedules the execution of the operators within the dataflow over the TaskManager(s). It keeps track of each operator's progress. The TaskManagers are responsible for the actual execution of the operators. The TaskManagers communicate the status of the running operators to the JobManager. Each of the TaskManagers maintain two sets of pools: (1) The buffer pool: Hold the outputs of the operators (i.e., the data streams), and (2) The network connections pool: Transfer the data streams between the operators over the network. The

dataflow graphs are executed in parallel. The operators are parallelized into one or more instances, referred to as subtasks, and the data streams are split into one or more stream partitions (i.e., one partition per consuming subtask). The operators implement the query processing logic (e.g., filters). Flink's intermediate data transfers are implemented through the exchange of buffers. When a data record is ready on the producer side (i.e., on an upstream operator instance), the data record is serialized and can be split into one or more buffers (a buffer can also fit multiple records) that can be forwarded to consumers (i.e., to a downstream operator instance). A buffer is sent to its designated consumer(s) as soon as it is full or when a timeout condition is triggered. This enables Flink to provide higher throughput by setting the size of buffers to a high value (e.g., a few kilobytes), or provide low latency by setting the buffer timeout condition to a low value (e.g., a few milliseconds).

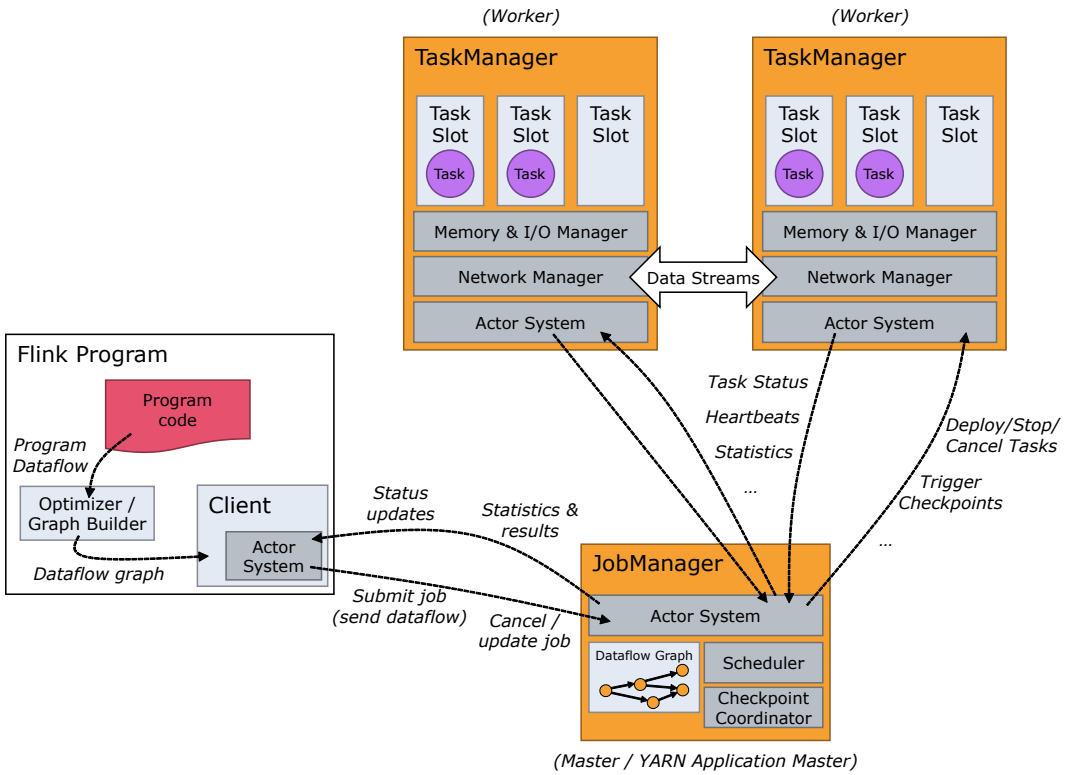


Figure 5.4. Apache Flink Architecture [12]

5.2.2 Realization in Flink

The realization of the abstraction in Flink relies on some core components within Flink's runtime engine. Figure 5.5 illustrates the architecture of Flink, and how the abstraction's five-techniques are adopted. The *Frequency-aware Data Buffering* technique ① is implemented in the *SourceReader* process. The *SourceReader*, analogous to a receiver in Spark, is responsible for ingesting the input data stream and for splitting it in memory to be consumed by the operators. The *SourceReader* process is to be enriched with the two data structures, *CountTree* and *BatchData*. The *Blocking Intermediate Data Streams* is the core abstraction for data-exchange between operators. An intermediate data stream represents a logical handle to the data that is produced by an operator, and can be consumed by one or more operators. It buffers all of the producing operator's data before making the data available for consumption, thereby separating the producing and consuming operators into different stages (i.e., batching and processing). The *SourceReader* applies the *Balanced Micro-batch Data Partitioning* ③ using the statistics collected in *CountTree* and *BatchData*, and emit balanced input into intermediate data stream buffers. The Buffers are used as a data-exchange mechanism to transfer intermediate results between operators. The *Early Micro-batch Release* ② leverages the blocking property of the buffers to synchronize the processing of the operators. The *Balanced Key Allocator* ④ leverages the *Custom Broadcasting* of the intermediate data buffers to enable balanced input to the aggregation stage of the dataflow (i.e., the Reduce stage). For correctness, and due to the continuous nature of the operator processes, *Control Events* are used to synchronize the boundaries of consecutive batches over all the operator instances. Control Events are special tuples that can be injected into the data stream by operators to communicate signals with downstream operators. For instance, a *watermark* control event includes a time attribute t indicating that all tuples with timestamps lower than t have already been processed by the upstream operator. These watermark control tuples are used as a unified measure of progress, and aids the execution engine in processing tuples in the correct event order, and serializes the operations, e.g., window computations via a unified measure of progress. Watermarks originate at the sources of a dataflow, and

propagate from the source-readers throughout the downstream operators of the dataflow. Finally, the JobManager is responsible for the *Elastic Scheduling* ⑤ of the operators.

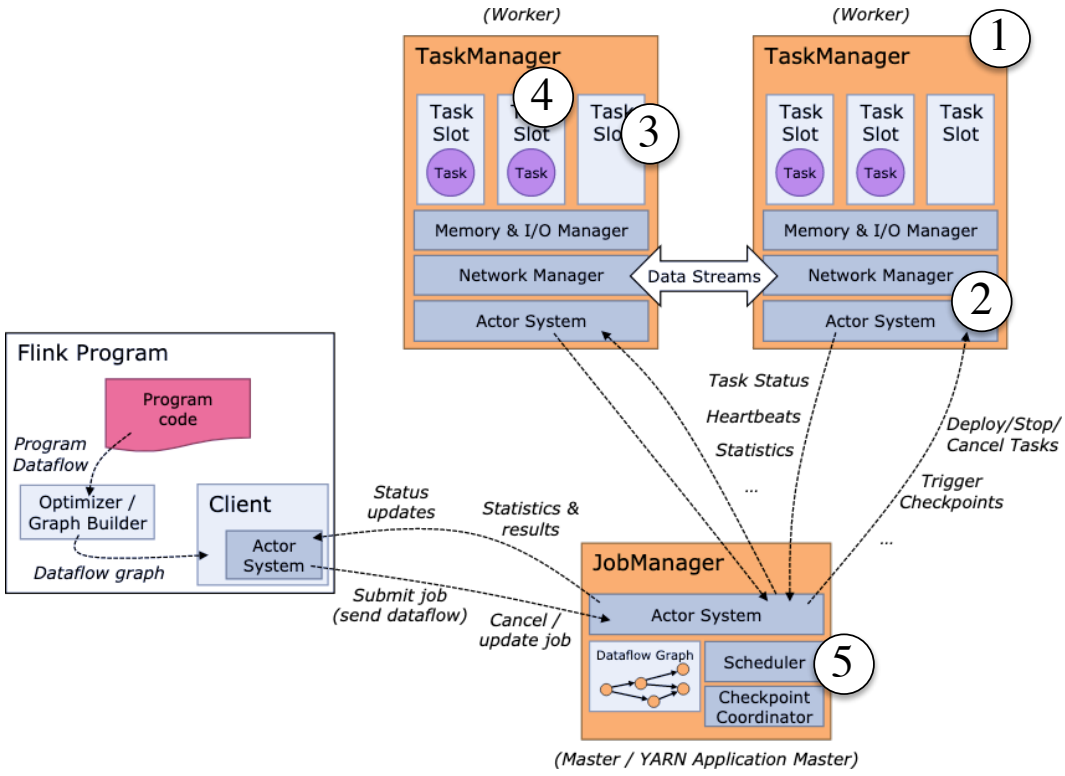


Figure 5.5. Realization in Apache Flink

5.3 Apache Storm

5.3.1 Architecture of Storm

Apache Storm is a distributed stream processing engine [48]. It has been widely used in industries and research institutions [54]. Originally, Storm's execution engine has adopted the tuple-at-a-time stream processing model. Nevertheless, *Trident* has been introduced to enable the micro-batch processing model on-top of Storm's engine [1]. Trident is an abstraction that is built over Storm to achieve higher-throughput processing. Storm's engine executes each streaming query as a topology of operators (e.g., see Figure 5.6). Similar to Spark and Flink, each topology is represented as a directed acyclic graph of tasks that are deployed over a cluster of machines. Storm's cluster includes a master machine and one or more

worker machines. The master runs the *Nimbus* process that is similar to Flink's*JobManager*. Nimbus is responsible for orchestrating the execution of the topology (e.g., assigning tasks to worker machines, and monitoring failures). Each worker runs a *Supervisor* instance. The supervisor continuously listens to Nimbus for assignments (i.e., tasks to execute). The supervisor starts and stops tasks on its worker machine according to Nimbus's assignments. In this setup, each supervisor executes a subset of the topology (i.e., a user's query). Typically, the topology consists of several tasks that are spread across the worker's machines. The core abstraction in Storm are the *spout* and *bolt* tasks. The spout connects to the stream source to ingest the input data tuples. The bolt performs the query operations on the input stream, and emits new processed streams. A bolt can consume multiple input streams and partitions its output to multiple bolts (i.e., downstream processes). The spouts and the bolts apply data-partitioning to send tuples to downstream processes (e.g., Shuffling or Key-grouping). Trident differs from the original Storm by applying the partitioning decision for a batch of tuples instead of tuple-at-a-time. In the next section, we show how the generic abstraction can be applied to Storm's Trident.

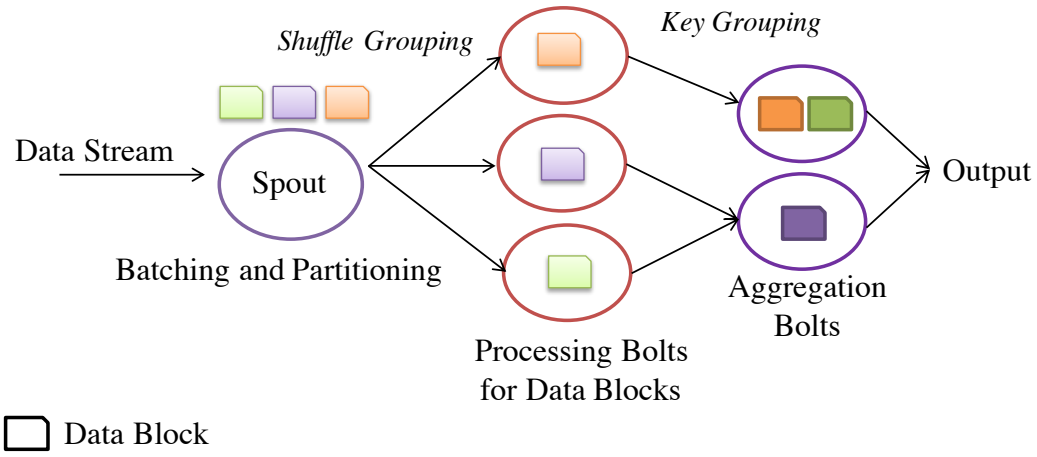


Figure 5.6. Apache Storm Topology

5.3.2 Realizing in Storm

The realization of the proposed schemes in Apache Storm is applied to its Trident abstraction. Both Storm and the Trident share the same architecture explained in Section 5.3.1.

However, Trident processes data streams as a series of batches that are referred to as transactions. In a Trident topology, the spout buffers the data tuples to accumulate small data batches. The size of these batches can be on the order of millions of tuples. These batches are partitioned into data-blocks, and are sent to downstream bolts (See Figure 5.6). The bolts operate on the granularity of data blocks. Figure 5.7 illustrates how the abstraction's five techniques can be deployed on a Trident's topology. In this topology, one spout is responsible for ingesting the input data source and two tiers of bolts are responsible for the processing. The first tier of bolts applies a parallel function analogous to Map while the second tier performs the aggregation (i.e., analogous to Reduce). The *Frequency-aware Buffering* ① is applied at the spout process. The two data structures *CountTree* and *BatchData* get populated as the spout ingest the input data tuples. The spout carries the *Balanced Micro-batch Data Partitioning* ③ to provide even-input to the downstream bolts. The *Early Micro-batch Release* ② allows the spout to partition the data batch at hand while continuously ingesting data tuples of the next data batch. The bolts on the first-tier applies the *Balanced Key Allocator* ④ to provide balanced input to the aggregation bolts. The Nimbus process monitors the run-time statistics of the topology (e.g., the variance in the bolts' throughputs) and triggers *Elastic Scheduling* ⑤ when necessary to adjust the number of bolts according to workload changes.

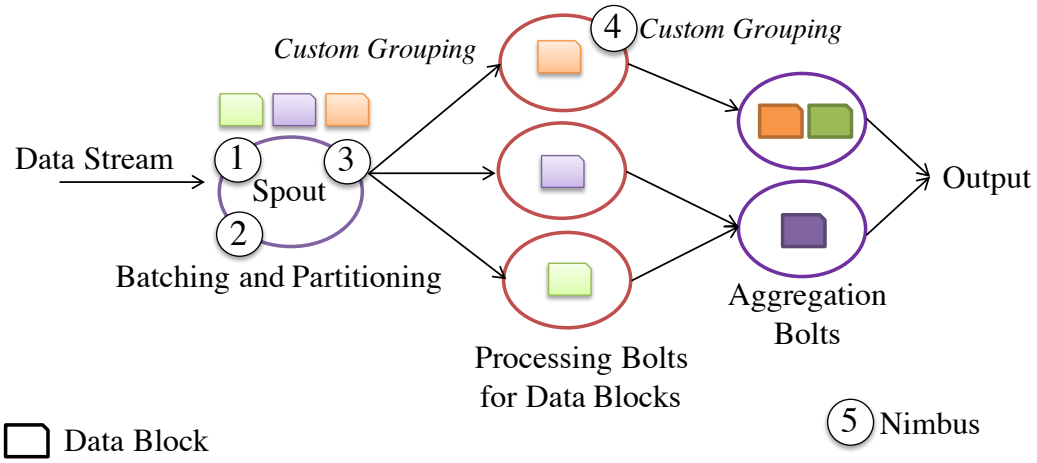


Figure 5.7. Realization in Apache Storm

5.4 Discussion on Performance Contributions

In this section, we discuss the performance gains expected in the three streaming engines, Spark, Flink and Storm, when adopting the five-techniques. In general, increasing the batch interval increases throughput and latency specially if the network bandwidth is not a bottleneck. In addition, when resources are not over-stressed (i.e., when batch interval $\geq$ processing time), latency is bounded by the batch interval. However, each of the three systems, Flink, Spark, and Storm, has distinct design features that lead to different degrees of throughput gains when adopting Prompt, Prompt+ or PartLy. We categorize these design features into five categories:

(1) In-Memory Data Management: Compared to Flink and Storm, Spark's execution cycle includes more data-transformation overheads. For instance, Spark requires the batched data to be transferred from the data receiver to the computing nodes through a block manager. This transfer is responsible for putting the data into the Spark Streaming abstract data type (i.e., a group of RDDs referred to as DStream). Even with the synchronization component of *Early Micro-batch Release*, the repetition of this operation for each batch is an overhead and contributes to the execution time. The Early-batch Release is able to make room for the data partitioning within the receiver processes but does not interfere with the memory management or the execution engine. Hence, this data transformation reflects on the overall system throughput. In contrast, Flink and Storm apply a simple data movement procedure. The batched data is directly moved from the data ingestion processes to the execution tasks (e.g., from the Spout to the Bolt in the Storm case). This simplicity enables Flink and Storm to have higher gains in throughput when optimized data-partitioning is applied.

(2) Physical Query Execution: Spark executes queries as a set of blocking transformations applied to the RDDs. The logical to physical execution of the transformation is not always one-to-one. For example, in the case of the Reduce stage, the transformation is executed by creating three types of RDDs (i.e., MappedValuesRDD followed by ShuffledRDD and CoGroupedRDD). Flink and Storm use a straightforward non-blocking operation, where the before-mentioned Reduce stage is executed as a single operation. This avoids expensive

in-memory data migration and enables Flink and Storm to benefit more from the optimized data partitioning.

(3) Back Pressure Mechanism: All three engines have a back-pressure mechanism to notify the data source if the data arrival rate is beyond the system's capacity. The three systems differ in their implementations of the back-pressure mechanism, which results in fluctuating the sizes of the micro-batched data over time. For example, Storm's back-pressure implementation is known for its fluctuating input rate [54]. Given a sinusoidal (i.e., variable) data rate, each of the systems will vary in their ingestion rates, and hence the micro-batch sizes and system throughputs.

(4) Scheduling Strategy: Spark would benefit the most from applying the elastic scheduling technique of Prompt+. The overhead of the centralized scheduler is a major bottleneck in Spark's performance. The main reason is due to Spark's reliance on short-lived computation tasks. This is in contrast to Flink and Storm that make use of continuous tasks and apply the data-to-computation principle. On the other hand, when the data arrival rate increases, the system has to adapt (e.g., by scaling out) in order to handle the increased arrival rate and process tuples without exhibiting back-pressure. Unlike Flink and Spark, Storm does not provide means to dynamically change the number of execution tasks (e.g., the number of bolts) during query execution. Hence there is no way to provide elasticity except by over-provisioning the execution tasks, and sharing them on concurrent queries.

(5) Fault Tolerance: Finally, the proposed techniques do not affect the fault tolerance mechanisms or exactly-once semantics provided by the underlying systems. For instance, the driver process in Spark may decide to apply parallel recovery in response to a failed worker. Similarly, Apache Flink may apply asynchronous incremental snapshots to provide periodic checkpoints. In both scenarios, the underlying system relies on the data dependencies in the execution graph to recover only the lost data due to failures and avoids repeating entire computations. The proposed techniques do not interfere with the generation of the data dependencies between the execution-tasks. However, it only changes how data keys are assigned to the data-blocks communicated through the data dependencies between the execution-tasks. Nevertheless, unlike the advanced recovery mechanisms in Spark and Flink, Storm replays the entire micro-batch through the processing topology in case of a failed task.

Even with an optimized data partitioning scheme in place, Storm's throughput will suffer in the case of failures.

5.5 Concluding Remarks

In this chapter, we show how the proposed techniques in Prompt, Prompt+, and PartLy can be realized in three widely-used stream processing systems. We provide a discussion on the expected performance gains due to the design differences in these systems.

6. CONCLUSIONS

In this dissertation, we study efficient data processing over micro-batched stream processing systems. In Chapter 1, we investigate the micro-batching paradigm and pinpoint some of its design principles that can benefit from further optimization. We show that existing micro-batch stream processing systems lack *Load-awareness* optimizations that are necessary to maintain performance and enhance resource utilization. We provide motivating break-down analysis of the micro-batching computational model and highlight the need for mitigating three challenges. These challenges are data partitioning, centralized-scheduling, and performance stability.

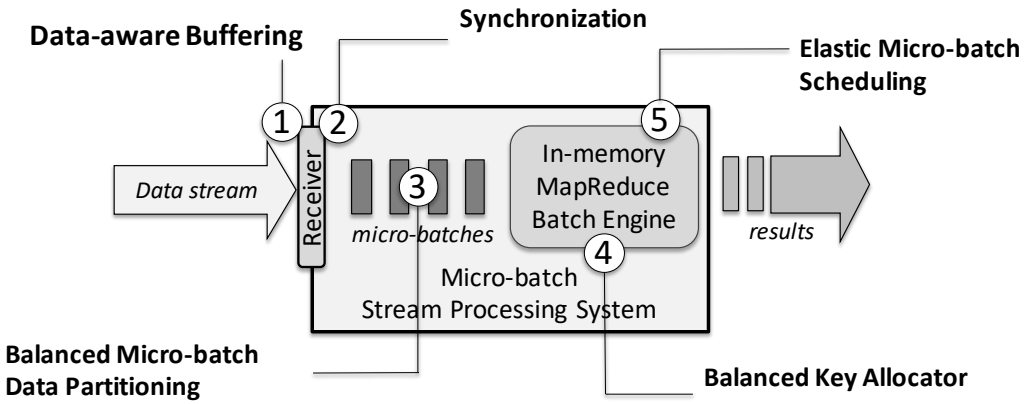
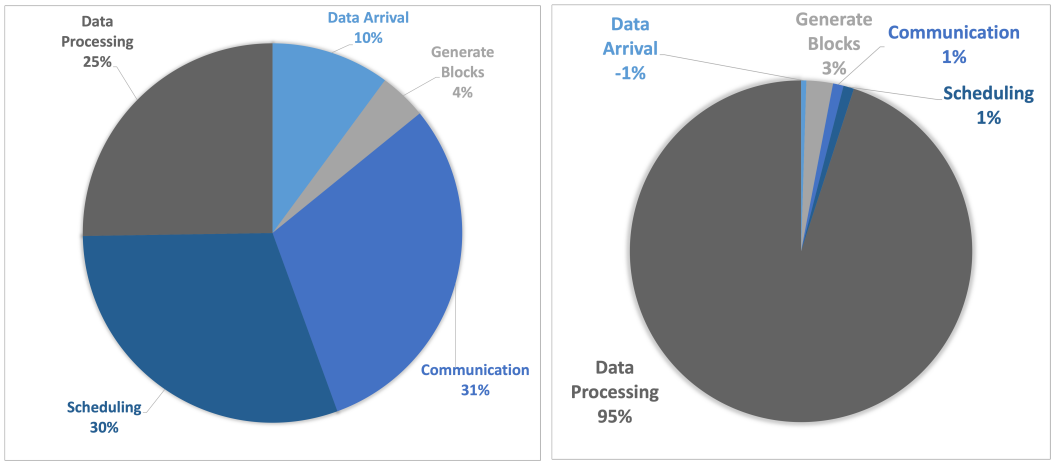


Figure 6.1. Optimizations in Micro-batched Stream Processing Systems

Chapter 2 presents an efficient data-partitioning for the micro-batch processing model. We formulate the problem of data partitioning in distributed micro-batch stream processing systems and prove that this problem in both the batching and processing phases is NP-hard. The data partitioning problems in the batching and processing phases are reduced 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 leverages a look-ahead data partitioning strategy that optimizes the performance of the micro-batch processing model [2]. Prompt improves system throughput by up to 2x using real and synthetic datasets over state-of-the-art techniques.



(a) Micro-batch Processing (b) Optimized Micro-batch Processing

Figure 6.2. Processing Time Breakdown of Micro-batch Computations

Chapter 3 provides a solution to mitigate the centralized-scheduling bottleneck and the performance stability concerns of the micro-batching computational model. We introduce Prompt+ with an elastic-scheduling technique for the micro-batching model. Prompt+ aims to overcome the overheads of task scheduling (i.e., serialization and communication) and maintaining resource utilization while enforcing latency. Prompt+ schedules multiple batch-computations at once to amortize the scheduling cost over a group of batches. In addition, Prompt+ uses three elasticity zones to monitor system stability and updates the degree of task-parallelism without interrupting the execution. Prompt+ is robust to fluctuations in data distribution and arrival rates. Experiments on the evaluation of the proposed elastic-scheduling technique and the combined effect, when applied along with the data partitioning techniques, show up to 5x improvement in throughput while enforcing latency guarantees.

Chapter 4 introduces PartLy, a learning-based technique to provide efficient data-partitioning of micro-batched data over execution tasks. PartLy leverages deep reinforcement learning model to learn a data partitioning policy using the run-time statistics of the execution tasks. PartLy enhances over Prompt through reducing the variance of execution-tasks. We formulate the data partitioning of the micro-batching model in the context of deep reinforcement learning. PartLy opens the possibility of autonomous execution and reduces the overheads of tuning heuristics by humans. We provide directions on how to extend this strategy to enable fully-autonomous stream processing systems.

Finally, Chapter 5 studies the realization of the proposed techniques in Prompt, Prompt+, and PartLy within three widely adopted micro-batched stream processing systems. We demonstrate how the ideas proposed in this dissertation are general and are applicable in Spark Streaming [61], Apache Flink [12] and Apache Storm [48]. Figure 6.2 summarizes the effect of all the proposed techniques on improving the ratio of data execution in a micro-batch processing time.

Bibliography

[1] Trident trident api for storm. http://storm.apache.org/releases/1.1.1/Trident-API-Overview.html.

[2] A. S. Abdelhamid, A. R. Mahmood, A. Daghistani, and W. G. Aref. Prompt: Online data-partitioning for distributed micro-batch streaming systems. In *Sigmod*, 2020.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. In *IEEE Data Eng. Bull.*, 2015.

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

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

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

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

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

[18] A. K. et. al. Brief survey of drl. In *IEEE Signal Processing*, 2017.

[19] R. L. et al. Multi-agent actor-critic for mixed cooperative-competitive environments. In *arXiv*, 2020.

[20] S. J. et al. Proximal policy optimization algorithms. In *arXiv*, 2017.

[21] S. M. et. al. Tensorforce: A tensorflow library for applied reinforcement learning. In *https://github.com/reinforceio/tensorforce*.

[22] 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.

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

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

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

[26] 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.

[27] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, G. Dulac-Arnold, J. Agapiou, J. Z. Leibo, and A. Gruslys. Deep q-learning from demonstrations. In *The Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[28] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In *SIGMOD*, 2020.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] V. R. Konda and J. N. Tsitsiklis. On actor-critic algorithms. In *SIAM J. CONTROL OPTIM.*, 2003.

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

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

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

[38] 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.

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

[40] 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.

[41] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*, 2019.

[42] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. In *arXiv*, 2018.

[43] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR*, 2019.

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

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

[46] 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.

[47] 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.

[48] opensourcedsystem. Storm. https://storm.apache.org/, March 2015.

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

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

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

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

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

[54] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @twitter. In *Sigmod*, 2014.

[55] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. Learning to route with deep rl. In *NIPS*, 2019.

[56] 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.

[57] H. Wang, H. He, M. Alizadeh, and H. Mao. Learning caching policies with subsampling. In *NIPS*, 2019.

[58] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P. Åke Larson, D. Kossmann, and R. Acharya. Qd-tree: Learning data layouts for big data analytics. In *SIGMOD*, 2020.

[59] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[60] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

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

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

[63] 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.

VITA

Ahmed S. Abdelhamid received his bachelor's degree in computer science from Cairo University in Egypt in 2003. He received his master's degree in computers science from Purdue University in 2014. He joined the Ph.D. program at Purdue University in fall 2011. His research interests are in the area of cloud data management where he focuses on efficient query processing over distributed systems. Before joining Purdue, he was a Staff Software Engineer in the Rational Series team at IBM Egypt. As a graduate student, he interned at Microsoft Research in Summer of 2016 and Summer of 2017.

PUBLICATIONS

- **Ahmed S. Abdelhamid** and Walid G. Aref, "Learning Data-Partitioning for Distributed Data Stream Processing with Cooperative Multi-Agents," Under Review for VLDB Conference 2021.
- **Ahmed S. Abdelhamid** and Walid G. Aref, "Prompt+: Efficient Distributed Processing over Micro-batched Data Streams," Under Review for VLDB Journal 2021.
- **Ahmed S. Abdelhamid** and Walid G. Aref, "PartLy: Learning Data Partitioning for Distributed Data Stream Processing," aiDM@SIGMOD 2020.
- **Ahmed S. Abdelhamid**,Ahmed R. Mahmood, Anas Daghistani, and Walid G. Aref, "Prompt: Dynamic Data Partitioning for Distributed Micro-batch Stream Processing Systems," SIGMOD 2020.
- Jonathan Goldstein, **Ahmed S. Abdelhamid**, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, and Irene Zhang, "AMBROSIA: Providing Performant Virtual Resiliency for Distributed Applications," VLDB 2020.
- **Ahmed S. Abdelhamid**,Mingjie Tang, Ahmed M. Aly, Ahmed R. Mahmood, Thamir Qadah, Walid G. Aref, and Saleh Basalamah, "Cruncher: Distributed In-Memory Processing for Location-Based Services," ICDE 2016.
- Ahmed M. Aly, **Ahmed S. Abdelhamid**, Mohamed S. Hassan, Ahmed R. Mahmood, Walid G. Aref , "AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data," VLDB 2015.
- Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, **Ahmed S. Abdelhamid**, Mohamed S. Hassan, Walid G. Aref, Saleh M. Basalamah, "Tornado: A Distributed Spatio-Textual Stream Processing System," VLDB 2015.