AdCom: Adaptive Combiner for Streaming Aggregations

Felipe Gutierrez[1]   Kaustubh Beedkar[1]   Abel Souza[2]   Volker Markl[1]

[1]Technische Universität Berlin, Germany   [2]Umeå University, Sweden

firstname.lastname@tu-berlin.de   firstname.lastname@cs.umu.se

ABSTRACT

Continuous applications such as device monitoring and anomaly detection often require real-time aggregated statistics over unbounded data streams. While existing stream processing systems such as Flink, Spark, and Storm support processing of streaming aggregations, their optimizations are limited with respect to the dynamic nature of the data, and therefore are suboptimal when the workload changes and/or when there is data skew. In this paper we present AdCom, which is an adaptive combiner for stream processing engines. The use of AdCom in aggregation queries enables pre-aggregating tuples upstream (i.e., before data shuffling) followed by global aggregation downstream. In contrast to existing approaches, AdCom can automatically adjust the number of tuples to pre-aggregate depending on the data rate and available network. Our experimental study using real-world streaming workloads shows that using AdCom leads to 2.5–9× higher sustainable throughput without compromising latency.

1 INTRODUCTION

Continuous or real-time applications often require real-time aggregated statistics over an unbounded stream of events or tuples. For example, ride-sharing platforms such as Uber and Lyft utilize real-time aggregated statistics about traffic conditions to provide suggestions on trip routes [1, 27]. As another example, interactive entertainment platforms such as King.com provide their data science teams with real-time aggregated statistics over billions of user events from different games and systems [11].

To efficiently process continuous streams of data in real-time, applications rely on Distributed Stream Processing Engines (SPEs) such as Spark Streaming [29], Apache Flink [5], Apache Samza [19], or Apache Storm [24]. These systems enable data stream processing with low-latency and high throughput, and can scale by distributing computations among a cluster of machines.

In the particular case of *streaming aggregations*, which are `groupBy-aggregation` queries on continuous data, query execution involves shuffling of data stream tuples among compute machines of the cluster. This data shuffling involves communication between machines via network, which incurs a performance overhead in terms of a decrease in throughput and an increase in end-to-end latency. The shuffling overhead—of which the network bandwidth between machines is an important factor—also increases as the degree of parallel processing increases. Therefore, it is essential to reduce the shuffling overhead to improve the overall performance of SPEs for streaming aggregations.

Current SPEs optimize data shuffling by extending the "combine plus reduce" pattern of MapReduce (batch) to streaming. In particular, SPEs first pre-aggregate upstream (i.e., locally aggregate tuples in each partition before shuffling over the network) and then perform a global aggregation operation downstream. To cater to the needs of continuous applications, the number

of input tuples accumulated prior to applying local aggregation is based on a pre-configured mini-batch interval (e.g., for every 1000 tuples or every 5 seconds).

While SPEs allow configuring the mini-batch interval, its fixed size during query execution is not ideal for real-time applications. This is because, it lacks adaptability to the dynamic nature (w.r.t. data rate changes and/or data skew) of datastreams. For example, if the throughput of a data stream suddenly increases to what the SPE cannot sustain (i.e., when the network gets saturated), SPE inflicts a "backpressure" on the upstream operators. Backpressure leads to an increase in the system's latency. Likewise, if there is (sudden) data skew in the stream, some instances of the (downstream) aggregation operation will process more records than others leading to saturation of network buffers and consequently affecting latency. One way to deal with the unpredictable nature of datastreams is to re-configure the size of the network buffers and/or mini-batch intervals to pre-aggregate. However, this requires re-starting the query, which is expensive and undesirable for continuous applications.

In this paper, we propose AdCom, which is an adaptive combiner for SPEs. In contrast to pre-aggregating tuples based on a fixed mini-batch interval, AdCom uses dynamic mini-batch intervals. This allows "on-the-fly" adjusting of the number of tuples to pre-aggregate. To deal with sudden changes in data rate, AdCom utilizes a feedback mechanism consisting of a proportional-integral controller that continuously monitors its network buffers, and an actuator that signals AdCom to adjusts its mini-batch interval. Thus, a high network usage results in pre-aggregating more number of tuples and vice-versa. This allows SPEs to adapt to sudden data rate changes and/or skew, and achieve a higher sustainable throughput without compromising latency.

We implemented AdCom in Apache Flink[1], which we considered as a representative SPE, and performed an extensive evaluation using real-world and synthetic datasets. Our results indicate that with AdCom, Flink can autonomously adapt to data rate changes and can execute aggregation queries with higher sustainable throughput (up to 2.5×) compared to existing approaches.

2 BACKGROUND

We start with a discussion on streaming aggregations that we consider in this paper. We then describe the limitations of SPEs in execution aggregation queries. We use Apache Flink as a representative SPE to explain key concepts. These concepts also generalize to other SPEs like Spark Streaming or Apache Storm.

2.1 Streaming Aggregations

We focus on the computing of streaming aggregates that are most common in stream analytics applications. More specifically, we consider unbounded aggregations on continuous queries that have the following general form:

```
dataStream.groupBy(...).aggregate(...)
```

In the above general form, the `groupBy(...)` transformation first groups elements of the data stream by the specified key(s).

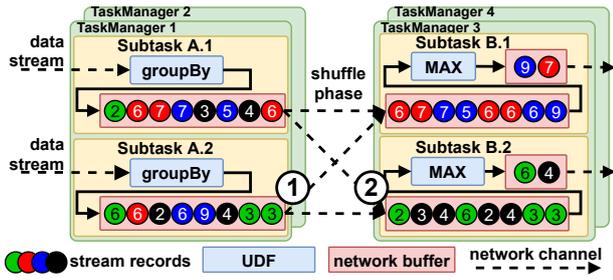[1]https://github.com/TU-Berlin-DIMA/AdCom

Figure 1: Illustration of executing a `groupBy-max` query.

Then, the `aggregate(...)` transformation computes a "rolling" aggregate as an output data stream. Such queries are the backbone of many common data analytic tasks such as retrieving, gathering, and organizing data. Common aggregate functions supported by SPEs include `sum()`, `min()`, `max()`, and `avg()` among others. SPEs also support parameterizing aggregate transformations with User Defined Functions (UDFs).

As an example, consider a weather analytics dashboard that continuously updates the maximum temperature of all regions in a neighborhood. The application receives as an input, a data stream S of `<timeStamp,regionId,temperature>` events, and executes the aggregation query[2]:

> `S.groupBy(regionId).max(temperature)`

For a given stream such as:

> $[1, A, 23], [2, A, 25], [1, B, 19], [1, C, 28], [2, B, 18], \dots$

the aggregation query produces the following resulting stream:

> $[A, 23], [A, 25], [B, 19], [C, 28], [B, 19], \dots$

Figure 1 gives a high level overview of executing such an aggregation query on a Flink cluster. It includes two tasks A and B with their respective subtasks running in parallel. In this example, subtasks A.1 and A.2, which perform the `groupBy` transformation, communicate with the two instances of task B, which perform the `max()` transformation. The `groupBy()` operations leads to shuffling of stream elements over the network[3]. Before shuffling, the output of the upstream (or sender) task is first queued in a (network) buffer at ①. The events, which are already grouped by key, are then flushed on to the network after a pre-configured timeout (e.g., 100ms) or when the buffers are full. Likewise, on the receiver side, the data is first queued in a buffer at ②, which is then consumed by the downstream subtasks.

2.2 Limitations of SPEs

SPEs strive to achieve a high sustainable throughput and low end-to-end latency. In real-world workloads, however, SPEs have to deal with two scenarios that affect its throughput and latency: (1) When the data stream's arrival rate increases to what a system can not handle, and (2) when there is data skew, which causes some instances of the aggregation tasks to process many more records than others. In both these scenarios, SPEs exhibit a backpressure mechanism, where the stream of events is queued up in network buffers before being processed. This leads to an increase in end-to-end latency, and in the worst case stalls the dataflow.

In the case of aggregation queries, backpressure on the sender tasks, either due to a high arrival rate or data skew, can be mitigated to some extent by first locally aggregating. This is akin to the use of a combiner in MapReduce [8]. In the context of data

---

[2]for brevity, we suppress the `map()` (or `project()`) transformation to project out the `timeStamp` attribute
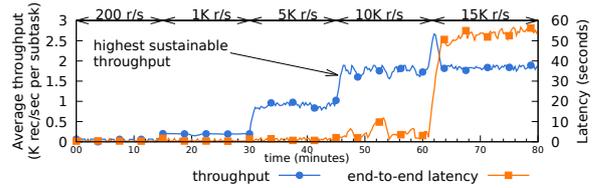[3]Other similar transformation are `keyBy()` and `reduceByKey()`.

Figure 2: Limitations of SPE w.r.t. data rate changes when executing a `groupBy-sum` query.

streaming, SPEs pre-aggregate records based on small mini-batch interval (e.g., for every 5 seconds) before shuffling them over the network. However, this pre-aggregation is suboptimal as using a fixed mini-batch interval leads to an increased end-to-end latency when workload changes and/or when there is skew.

We illustrate this limitation with an experiment that we performed using Apache Flink. We considered a `groupBy-sum` query and a setup comprising four local (pre-) aggregation tasks and four global aggregation tasks. The mini-batch interval to pre-aggregate was set to 5s and 5K records. Figure 2 shows the system's throughput (in blue; left axis) and its end-to-end latency (in orange; right axis).

As we observe in Figure 2, the end-to-end latency remains constant ($\sim$0.5s) as the data rate increases from 200 records/s to 5K records/s. Also, the system achieves its highest sustainable throughput of $\sim$8K records/sec. Note that when the arrival rate further increases to 15K records/s, we see a spike in end-to-end latency. This is because, the system exhibits a backpressure mechanism and consequently an increased (up to 60s) end-to-end latency when it can no longer cope with the arrival rate.

One way to deal with the changes in workload is to re-configure the number of tuples to pre-aggregate and/or the size of the network buffers. However, this requires restarting the query each time the workload changes, which is expensive and undesirable for real-time applications.

3 ADAPTIVE COMBINER

We now present AdCom, an adaptive combiner for SPEs that overcomes the limitations mentioned above. The use of AdCom aids in optimizing aggregation queries in SPEs by dynamically adjusting the mini-batch interval to pre-aggregate prior to data shuffling, which enables the SPE to constantly achieve the highest sustainable throughput without compromising latency.

The key challenge in designing AdCom is to determine when to emit locally aggregated tuples. As discussed above, using a fixed mini-batch interval is useful, but when the arrival rate of records is higher than what the network can handle, the backpressure mechanism is activated and leads to high latency. On the other hand, if the arrival rate is too low, then a large mini-batch interval will diminish the lowest achievable latency for the query.

3.1 The Feedback Mechanism

We tackle the above challenge by making use of a feedback mechanism, which comprises (1) a controller and (2) an actuator. The controller continuously monitors the network buffers and computes an optimal mini-batch interval for the current workload. The actuator periodically sends "signals" (i.e., the new configuration for the mini-batch interval) to the parallel instances of AdCom. This feedback mechanism allows AdCom to dynamically adjust the time (and/or number of tuples) for which it should locally aggregate tuples.

Figure 3 gives an overview of executing an aggregate query using AdCom. Before we delve into its details, recall that backpressure is inflicted when the incoming arrival rate of records surpasses what the system can handle. In the context of query execution, this happens when the (sender's) network buffer is queued up with records that it cannot flush to the network (i.e., shuffle). More formally, denote by λ the rate at which a pre-aggregation task writes records to its buffer (i.e., the mini-batch interval) and by μ the rate at which the records are flushed out from the buffer. Further, let $\rho = \frac{\lambda}{\mu}$ denote the fraction of buffer utilized. Intuitively, a backpressure situation arises when $\rho \geq 1$. The key idea of our approach is to always keep $\rho < 1$ by continuously updating λ, which translates to adapting the mini-batch interval to pre-aggregate[4].

3.2 AdCom's Controller and Actuator

We now detail the working of the controller and actuator. As shown in Figure 3, the controller and the actuator are part of Flink's job manager. This allows us to globally control all (parallel) AdCom instances (i.e., pre-aggregation subtasks) that are executed in the same phase. This is crucial for preserving the query semantics with respect to the order in which the records are processed downstream. Otherwise, having a controller for each of AdCom's parallel instance would result in writing records (to its buffers) at different time intervals. This may lead to a change in the order that records are processed downstream, which may be undesirable depending on the application.

We use a proportional-integral (PI) controller to continuously compute the optimal λ. PI controllers have been widely used in control systems and applications that require continuous modulated control[5]. We refer interested readers to [12] for a comprehensive overview. In what follows, we discuss how we use a PI controller in the context of performing streaming aggregations.

At high level, and as illustrated in Figure 3, the controller continuously calculates an error value $e(t)$ for each time t, which is the difference between the desired value ρ_d of the network buffer utilization and the measured current buffer utilization ρ. Based on the error value, it updates λ based on a proportional (K_P) and integral (K_I) terms. The proportional control term determines the correction in λ, which is proportional to $e(t)$. The integral term further applies a correction based on past values of $e(t)$, to diminish the residual error (i.e., when $e(t) > 0$ or $e(t) < 0$ after applying the proportional correction).

In more detail, we compute $e(t)$ as follows. Denote by $A_1, \ldots, A_n$ the parallel instances of AdCom, and by ρ_{A_i} the buffer utilization for instance A_i. Since we want to globally control all instances (as mentioned above), we compute the error value as:

$$e(t) = - \begin{cases} \rho_d - \mathsf{avg}(\rho_{A_1}, \ldots, \rho_{A_n}) \text{ if } \nexists A_i \text{ s.t. } \rho_{A_i=1} \\ \rho_d - 1 \text{ otherwise} \end{cases}$$

In other words, we compute $e(t)$ using the average buffer utilization across AdCom's parallel instances. To cope with data skew, we compute $e(t)$ as $\rho_d - 1$, i.e., when there is at least one AdCom instance with 100% buffer utilization. The desired value ρ_d is set based on the Service-Level Objective (SLO) to keep low or medium backpressure on the upstream operators, which usually corresponds to 50–80% usage of the network buffers.

When $e(t) > 0$, the ρ signals are too high, then the controller produces an input signal that decreases ρ. This materializes by

---

[4]Alternatively, one can continuously update μ or both.
[5]Some control systems additionally make use of a derivative component, i.e., a PID controller; we do not use it as it is suitable only for slow moving loops.
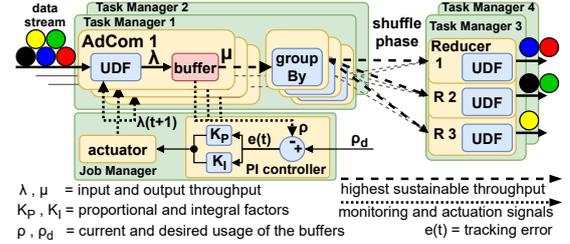

λ , μ = input and output throughput
K_P , K_I = proportional and integral factors
ρ , ρ_d = current and desired usage of the buffers

highest sustainable throughput
monitoring and actuation signals
$e(t)$ = tracking error

Figure 3: Overview of AdCom.

increasing the pre-aggregating time (and decreasing λ) at AdCom. Vice-versa, when $e(t) < 0$, the controller produces an input signal that increases the usage of the buffers. This leads to "less aggregation" and increase in λ at AdCom. Finally, based on the $e(t)$, the controller applies a correction based on proportional (K_P) and integral (K_I) terms, which we explain next.

Proportional Factor (K_P). It is crucial for any controller to let the magnitude of the corrective action depend upon the magnitude of the error. This states how quickly AdCom responds to errors. For instance, small errors lead to small adjustments, whereas larger errors result in greater corrective actions. This is accomplished by the proportional factor K_P (Equation 1). It helps the controller apply the largest control action if $e(t)$ is large and not making the system unstable if $e(t)$ is small. The next proportional parameter (i.e., the mini-batch interval) to pre-aggregate records is denoted by $\lambda(t + 1)$ and it is computed based on the current parameter $\lambda(t)$ summed with the proportional factor K_P times the error $t(e)$, as follows:

$$\lambda(t + 1) = \lambda(t) + [K_P.e(t)] \tag{1}$$

The controller estimates a proportional correction if $e(t)$ is within the desired min and max values of ρ_d. However, since we consider average buffer utilization, it may be possible that $e(t)$ becomes zero. Therefore, no correction is taken in this case. One way to fix this is to reduce the desired range or increase K_P. However, this leads to AdCom making rapid and unstable adjustments, which is very undesirable. Therefore, we further use an integral factor K_I to determine the corrective action.

Integral Factor (K_I). The integral factor makes the controller not react only based on the momentary $e(t)$ but also based on its previous values. It provides a way to amplify small errors and keep adding them up over time. The accumulated values provide a significant control signal and help the system not oscillate when it reaches the optimal time to locally aggregate tuples. We compute the integral factor based the three last values of $e(t)$ with a sliding window-based histogram that implements the reservoir algorithm. In our experiments, this window size was enough to achieve a steady-state on the pre-aggregation parameter of AdCom (see Section 4). Equation 2 composes the AdCom' *PI controller* with its proportional and integral factors that we use to calculate the new pre-aggregation parameter λ.

$$\lambda(t + 1) = [\lambda(t) + (K_P.e(t))] + [\lambda(t) + (K_I. \int_t^{(t-3)} e(\tau).d\tau)] \tag{2}$$

3.3 Using AdCom in Flink

One can make use of AdCom at the API level, by parameterizing it with the groupBy() key and a query specific UDF for pre-aggregation. For example, our modifications to Flink allow us to write the example aggregate query of Section 2 as:

```
S.adcom(regionId, max(temperature))
 .groupBy(regionId).max(temperature)
```

4 EXPERIMENTS

We conducted an experimental study using a real-world dataset in the context of a ride-sharing application. Our goal was to compare the performance of Flink when using AdCom and other existing optimizations. In particular, we investigated: (i) how well AdCom adapts when data rate changes; (ii) how well AdCom fares when there is data skew; (iii) AdCom's resource efficiency, and; (iv) how distributive and user-defined aggregate functions affect its performance. We found that:

- Flink+AdCom achieved up to 2.5−9× higher sustainable throughput when data rate increased by 4×.
- AdCom is competitive to state-of-the-art when handling skew.
- Flink+AdCom with 8 reducers achieved 9× higher throughput than Flink+noOptimization, and 3× higher throughput than state-of-the-art.
- Flink+AdCom achieved similar throughput but lower latency when using distributive and user-defined aggregate functions.

4.1 Experimental Setup

Implementation and cluster. We implemented AdCom as a new operator in Apache Flink. The PI controller and the actuator are implemented as components of Flink's job manager and use Flink's job monitoring API to monitor network buffer usage of AdCom. All of our experiments were run on a local Flink cluster consisting of four machines, each with 16GB of main memory, one hard disk of 2TB, one Intel Xeon 2.66GHz 64-bit 8 core CPU, and Ubuntu 16.04.6 (kernel GNU/Linux 4.4.0) as the operating system. The machines in the cluster are connected via 1 Gbps Ethernet. We used Apache Flink 1.11.2 and Java 1.8 for our implementation. We configured the Flink cluster with four Task Managers and one Job Manager, and set the maximum sub-task parallelism to 8 per node (i.e., the same number of CPU cores).

Datasets and aggregation queries. We used the New York City Taxi and Limousine Commission (TLC) [22] dataset. It consists of three million taxi trip records with fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. We additionally considered the TPC-H benchmark dataset [4] with scale factor 5.

For the TLC dataset, we considered queries with algebraic and distributive aggregate functions. In particular: (i) we considered an event-count query (which we denote by Q1 in the text below) that sums the number of passengers for each taxi driver, and (ii) and an aggregation query that computes for each taxi driver the average number of passengers, trip distance, and trip time (denoted by Q2). For the TPC-H data we considered the TPC-H query 1, which is an aggregation query over a single table (lineitem) consisting of COUNT, SUM, and AVG aggregate functions.

We followed Karimov et al. [15] to generate on-the-fly data streams for benchmarking streaming applications. In particular, we load the datasets in memory and implement a streaming data source that can emit events with different characteristics (see Sections 4.2 and 4.3 below).

Baseline. To evaluate the performance of AdCom, we compare it with no pre-aggregation, and with the state-of-the-art streaming aggregation optimizations available in Flink [13]. In more detail, we compare against (i) MiniBatch, which buffers input records before the shuffle phase, and (ii) Local-Global Aggregation, which executes the MiniBatch and then pre-aggregates records locally (akin to Combine plus Reduce in MapReduce).
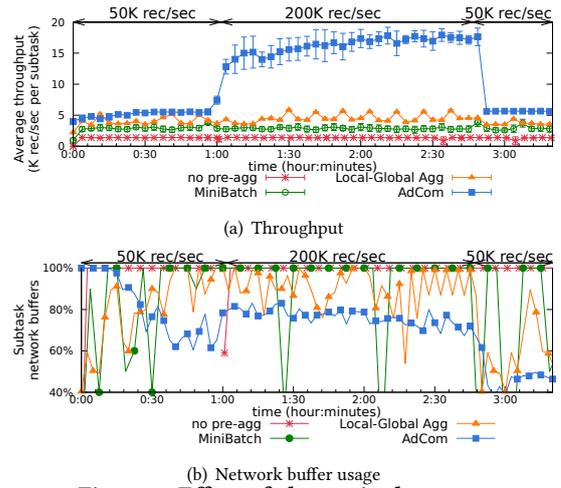

(a) Throughput

(b) Network buffer usage

Figure 4: Effect of change in data rate.

We set the maximum latency and the maximum number of input records that can be used for MiniBatch and Local-Global aggregation to 3 seconds and 3K events, respectively.

4.2 Effect of Change in Data Rate

We first study how well Flink adapts to a sudden surge in data arrival rate when using AdCom and compare its performance with the baseline approaches. We consider the TLC dataset, and a stream data source that for the first 60 mins generates events at 50K records/sec, then for the next 100 mins at 200K records/sec, and finally again reverts to a rate of 50K records/sec. We considered the aggregation query Q1 and measured its throughput and latency, for which the results are shown in Figure 4.

We observed that Flink+AdCom achieved a higher throughput than Flink+MiniBatch and Flink+LocalGlobal (see Figure 4(a)). More specifically, when the input throughput is 50K rec/sec, AdCom achieved an optimal mini-batch interval of 3.5s (starting from the default value of 500ms) compared to a fixed interval (of 3s and 3K events) for Flink+LocalGlobal. This allows us to obtain a slightly higher throughput for the first 60mins. As the input rate surges to 200K records/sec, AdCom further adapts its mini-batch interval (from 3.5s to 8.5s), which allows Flink+AdCom to achieve much higher throughput (from 3.6× up to 9×).

Figure 4(b) also shows the percentage of Flink's network buffer utilization for such a workload, which correlates to its latency. High buffer utilization indicates high backpressure and increased latency, whereas a lower buffer utilization indicates a low backpressure and lower latency. We observed that with AdCom, Flink had an overall lower buffer utilization due to AdCom constantly adjusting its mini-batch interval. When the workload suddenly surges to 200K records/sec, AdCom had a buffer utilization (up to 80%) but stabilizes after it reaches a sustainable throughput of 18K records/sec per subtask (see Figure 4(a)) pre-aggregating records every 8.5s. With baseline approaches, we observe that pre-configured mini-batch intervals are not ideal for dynamic workloads and lead to high buffer usage with surges in data rate.

Our results indicate that AdCom allows Flink to adapt to data rate changes and achieve a higher sustainable throughput.

4.3 Effect of Skew Workloads

We now proceed to evaluate how well Flink performs in the presence of skew workloads. We consider the TLC dataset, and a stream data source at 50K records/sec that for the first 60 mins
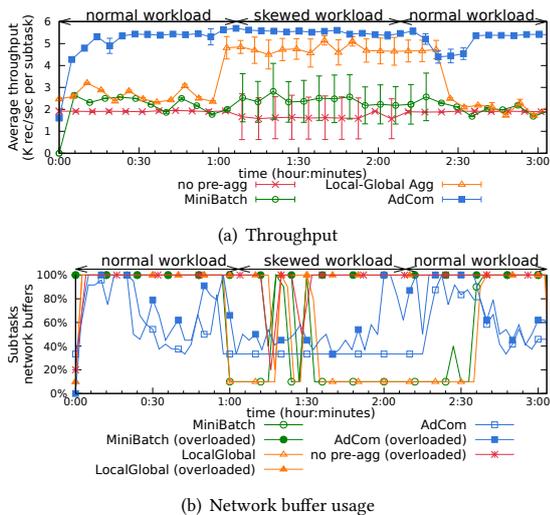
(a) Throughput

(b) Network buffer usage

Figure 5: Effect of data skew.

Figure 6: Effect of subtask's parallelism.

Figure 7: Effect of User-defined and Distributive Aggregate Functions.

generates tuples following a uniform key distribution, then for the next 70 mins following a skewed distribution, and finally again reverts to a normal distribution. We considered the aggregation query Q1 and measured its throughput and Flink's network buffer utilization, for which the results are shown in Figure 5.

We observed that the LocalGlobal optimization allowed Flink to reduce the network shuffle and cost of global aggregation in the presence of skew. However, as shown in Figure 5(a), we observed that throughput across subtasks had some variation. Flink+AdCom also responded well to skew as it also first locally aggregates, and achieved a slightly higher throughput with negligible variation. This is because, AdCom adjusted its mini-batch interval to 3.5s compared to (pre-configured) 3s of LocalGlobal. We also show the network buffer utilization for this setting in Figure 5(b). Note that different subtasks have different buffer utilization, and hence we show two lines (one for the overloaded task and other for remaining tasks). As observed in Figure 5(b), AdCom leads to lower buffer utilization and hence lower backpressure.

Overall, our experiments indicate that AdCom is competitive to LocalGlobal aggregation in dealing with data skew.

4.4 Resource Efficiency

We now evaluate the resource efficiency of Flink when using AdCom for data streams with high arrival rate. A common strategy to cope with high arrival rate is to add more resources, i.e., increase the degree of parallelism. When we add more global (i.e.: reducer) subtasks after the shuffle phase, it helps the query to avoid backpressure. Hence, it can sustain a higher throughput.

We consider the TLC dataset and a streaming source that generates events at a fixed rate of 200K rec/sec and process query Q1 with different aggregation optimizations. To deal with a higher workload, we set the maximum latency and the maximum number of input records that can be used for MiniBatch and LocalGlobal aggregation to 7 seconds and 7K events, respectively.

Figure 6 shows the throughput of the query for different strategies. We observed that Flink+AdCom with 8 reducers achieved 9× higher throughput than Flink+noOptimization, and 3× higher throughput than Flink+MiniBatch and Flink+LocalGlobal. Further, Flink+AdCom required only 8 reducers to achieve the same throughput that Flink+LocalGlobal achieves with 24 reducers. This is because the MiniBatch and the LocalGlobal approaches
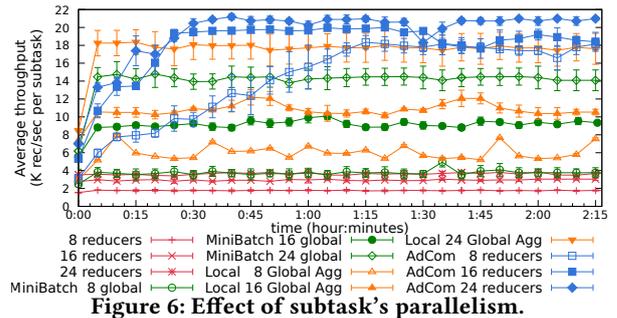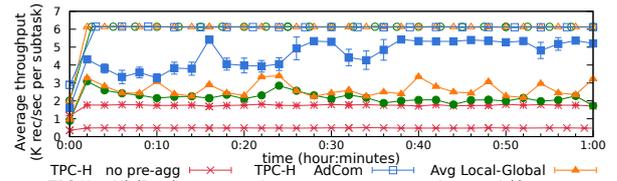
first accumulates (mini-batches) records every 7 seconds, and then aggregate every 7K tuples. In contrast, AdCom leverages a hash-based data structure, which allows us to consume and pre-aggregate tuples in a single pass and hence requires less resources.

Based on these experiments, we conclude that AdCom is more resource efficient compared to MiniBatch and LocalGlobal.

4.5 Results for User-defined and Distributive Aggregate Functions

Lastly, we evaluate the performance of AdCom for aggregation queries with both distributive and algebraic aggregate functions. We considered queries Q2 (on the TLC dataset) and TPC-H Q1 and set the maximum latency and the maximum number of input records that can be used for MiniBatch and LocalGlobal aggregation to 1s and 1K events, respectively. The arrival rate for each stream was 50K records/sec. The results are shown in Figure 7.

We first discuss the results for TPC-H query Q1. We observed that Flink achieves the same throughput when using either of AdCom, MiniBatch, or LocalGlobal. This is because the configured maximum latency of 1s and the maximum number of input records of 1K events that configured for MiniBatch and LocalGlobal aggregation was sufficient to handle the workload and the user-defined aggregate function. Although Flink+AdCom achieved a similar throughput, it adapts its mini-batch interval from 1s to 50ms, and thus achieving a lower latency than Flink+MiniBach and Flink+LocalGlobal (not shown here).

For query Q2, which involves an algebraic aggregate, we observed that Flink+AdCom again achieves a higher throughput than Flink+MiniBatch and Flink+LocalGlobal. We note that, other than AVG(), Flink+LocalGlobal cannot optimize queries with algebraic aggregate functions. In contrast, AdCom allows specifying any UDF for pre-aggregating which makes it suitable to aggregation queries that accept a "useful" combiner.

Overall, we found that Flink+AdCom achieves a higher throughput without compromising on latency even for distributive aggregate functions. We also performed experiments (not shown here due to space constraints) considering queries involving other distributive and algebraic aggregate functions such (e.g., sum, max, min, and top-k) and observed similar performance.

5 RELATED WORK

We now discuss how ideas presented in this paper are related to prior work on adaptive stream processing. Due to space constraints, we discuss works that are most related.

Das et al. [7] studied the effect of batch sizes on throughput and latency, and proposed a control algorithm based on fixed point iteration. Their work focuses on determining the optimal batch size for ingesting the data into SPE and relies on past queries. Zhang et al. [31] proposed Dynamic Block and Batch Sizing (DyBBS) using an online control algorithm integrated with isotonic regression. Besides adjusting the micro-batch size, DyBBS also adjusts the execution parallelism (i.e., batch size/block size in Spark). Instead of adapting the batch size, Drizzle [26] focuses on optimally scheduling multiple batches (or a group), and automatically tunes the group size. A-scheduler [6] further extended this approach by dynamically changing the batch sizes using an expert fuzzy control mechanism. Wu et al. [28] studied the impact of batch size on Kafka streams that are then ingested into SPEs, and proposed a reactive batching strategy to cope with variable network conditions. Lohrmann et al. [17] proposed strategies to switch between adaptive buffer (re-)sizing and dynamic task chaining to optimize execution plans. All these approaches focus on adjusting the batch size w.r.t. the entire query or prior to ingesting the data stream into the SPE. In contrast, we focus on the batch size specific to an (aggregation) operator. AdCom is thus complementary to the above approaches.

Apart from dynamically configuring the batch size, many works have also focused on adapting the execution plan. WASP [14] uses a network-aware framework that is able to adapt the query plan to the resources available in run-time via task re-assignment, operator scaling, and query re-planning. Nasir et al. [18] proposed a hash-based algorithm to partition data that optimizes network shuffle and deals with skew workloads. While [3, 16, 20, 25] have studied eliminating redundant computations via dynamically sharing of data and/or compute, [10, 21] focused on adaptability via query complication. Eddies [2] also tackled the problem of unpredictable workloads by reordering operators at runtime.

Feedback mechanisms have been a central component in enabling adaptive stream processing. FAST [9] uses adaptive sampling methods based on a PID controller to adjust the sampling rate for processing streams. Tolosana-Calasanz et al. [23] uses a feedback mechanism based solely on proportional factor to minimize resource utilization. In AdCom, we leverage a PI controller, which is specific to optimizing local-aggregations.

6 CONCLUSION AND FUTURE WORK

Adaptability of SPEs to varying workloads is important for real-time applications. In this paper, we considered streaming aggregations, which are the backbone of many real-time applications. We have proposed AdCom, an adaptive combiner for SPEs, which improves the performance of aggregation queries under variable workloads. We have proposed a lightweight feedback mechanism that continuously monitors the network buffers, and allows AdCom to autonomously adapt to varying workloads. In our experimental evaluation using a real-world dataset, we have shown that AdCom achieves a higher sustainable throughput (up to 9×) without compromising latency, is resilient to data skew, and is resource efficient when compared to existing optimizations.

As future work, we plan to extend our work to fog and edge computing environments [30]. In particular, we plan to study how to adapt AdCom for nodes that are resource (e.g., memory) constrained. We also plan to make AdCom adaptive to hybrid environments that include unreliable compute nodes and network channels. Lastly, we plan to extend existing optimizers to create the AdCom UDF during compile time, so it can relieve the developers of this task.

REFERENCES

[1] Introducing AthenaX. 2017. Uber Engineerings Open Source Streaming Analytics Platform. https://eng.uber.com/athenax
[2] Ron Avnuret al. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*. 261–272.
[3] Shivnath Babuet al. 2004. StreaMon: An Adaptive Engine for Stream Query Processing. In *SIGMOD*. 931–932.
[4] Transaction Processing Performance C. 1988-2021. *TPC-H, a decision support benchmark.* http://www.tpc.org/tpch
[5] Paris Carbone et al. 2015. Apache flink: Stream and batch processing in a single engine. In *IEEE CS*. 28–38.
[6] D. Cheng et al. 2018. Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming. In *IEEE TPDS*. 2672–2685.
[7] Tathagata Das et al. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing. In *SOCC*. 1–13.
[8] Jeffrey Deanet al. 2008. MapReduce: Simplified Data Processing on Large Clusters. In *Commun. ACM*. 107–113.
[9] Liyue Fan et al. 2013. FAST: Differentially Private Real-Time Aggregate Monitor with Filtering and Adaptive Sampling. In *SIGMOD*. 1065–1068.
[10] Philipp M. Grulich et al. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *SIGMOD*. 2487–2503.
[11] F. Gyula et al. 2017. Scalable real-time analytics. US Patent App. 15/475,913.
[12] Joseph L Hellerstein et al. 2004. Feedback control of computing systems. In *Wiley Online Library*.
[13] Streaming Aggregation in Flink Table API. 2019. . https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/tuning/streaming_aggregation_optimization.html
[14] Albert Jonathan et al. 2020. WASP: Wide-Area Adaptive Stream Processing. In *Middleware*. 221–235.
[15] J. Karimov et al. 2018. Benchmarking Distributed Stream Data Processing Systems. In *ICDE*. 1507–1518.
[16] Jeyhun Karimov et al. 2019. AStream: Ad-hoc Shared Stream Processing. In *SIGMOD*. 607–622.
[17] Björn Lohrmann et al. 2014. Nephele streaming: stream processing under QoS constraints at scale. In *Cluster computing*. 61–78.
[18] M. A. U. Nasir et al. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *IEEE ICDE*. 137–148.
[19] Shadi A. Noghabi et al. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. In *VLDB Endowment*. 1634–1645.
[20] Do Le Quoc et al. 2018. ApproxJoin: Approximate Distributed Joins. In *SoCC*. 426–438.
[21] Filippo Schiavio et al. 2020. Dynamic Speculative Optimizations for SQL Compilation in Apache Spark. In *VLDB Endowment*. 754–767.
[22] New York City Taxiet al. 2020. . https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[23] Rafael Tolosana-Calasanz et al. 2016. Feedback-control & queueing theory-based resource management for streaming applications. In *IEEE TPDS*. 1061–1075.
[24] Ankit Toshniwal et al. 2014. Storm@Twitter. In *SIGMOD*. 147–156.
[25] Jonas Traub et al. 2019. Efficient Window Aggregation with General Stream Slicing.. In *EDBT*. 97–108.
[26] Shivaram Venkataraman et al. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *SOSP*. 374–389.
[27] Alex Woodie. 2019. *What's Behind Lyft's Choices in Big Data Tech.* https://www.datanami.com/2019/05/10/whats-behind-lyfts-choices-in-big-data-tech/
[28] H. Wu et al. 2020. A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-time. In *ISSRE*. 207–217.
[29] Matei Zaharia et al. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*. 423–438.
[30] Steffen Zeuch et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.
[31] Q. Zhang et al. 2016. Adaptive Block and Batch Sizing for Batched Stream Processing System. In *IEEE ICAC*. 35–44.