Dynamic Block Sizing for Data Stream Processing Systems

Robert Birke
IBM Research - Zurich
Cloud & Computing Infrastructure

Evangelia Kalyvianaki
City University London

Walter Binder
University of Lugano
Faculty of Informatics

Lydia Y. Chen
IBM Research - Zurich
Cloud & Computing Infrastructure

***Abstract*—Real-time processing of big data is becoming one of the core operations for various areas, such as social networks and anomaly detection. Thanks to the rich information of the data, multiple queries can be executed to analyse the data and discover a variety of business values. It is very typical that a cluster infrastructure running for example Spark Streaming data stream processing system would execute multiple queries simultaneously. To enable that multiple queries can be answered by the same data concurrently, it is important to effectively allocate the CPU-cores of the underlying infrastructure amongst them, meanwhile adhering to the latency constraints of the individual queries. In this paper, we consider the problem of allocating CPU-cores in a Spark Streaming infrastructure in the context of two types of queries, namely primary and optional, that are associated with high- and low-priority analysis, respectively. We develop a controller, iBLOC, that adjusts the block sizes of streaming jobs on the fly and parallelism level of jobs, according to the input data rates and the query priorities. Our evaluation shows that we can achieve significant CPU-core savings from the primary query type such that multiple queries can run together without impairing their latency constraints, in comparison to a static block-sizing scheme.**

I. Introduction

There is an increasing demand for real-time processing on "big data" generated from various sources, such as from public sensing systems and social media, in order to analyse and extrapolate values and trends from the data on the fly as they are being generated. For example, data generated from Twitter has been shown useful for the early detection of financial movement [12] as well as for earthquake prediction [1]. Often, multiple analysis associated with different business urgencies can be derived from the same set of data [17]. Recent data-parallel processing systems, such as Comet [5], Spark Streaming [18, 15], SEEP [2], Storm, and Naiad [10] aim at processing of live data in a timely fashion such that the analysis results are continuously up-to-date. It is however challenging to use limited amount of CPU-core resources to execute different queries concurrently in a shared infrastructures, without violating latency constraint [8, 11].

Motivated by the emergence of data processing system, we target on such *data stream processing systems*, executing two types of queries, namely primary and optional, on a set of CPU-cores. The primary query has a high priority and a stringent latency requirement, whereas the optional query handles a non-critical analysis. Different types of queries are characterized by their priorities as well as their processing flow that is typically composed of a set of interconnected operators using different numbers of CPU-cores depending of the input data rates. Consider Spark Streaming as an illustrative example for the dependency of the query data flow and CPU-core consumptions. Data is first grouped into blocks executed by tasks and then multiple blocks are batched into jobs that are executed on the Spark cluster. Each task consumes a computing slot, which is configured based on the CPU-cores. Consequently, fewer and larger blocks imply a smaller number of tasks per jobs. In turn, the job latency is decided by the block sizes, the number of tasks, and the available number of CPU cores. To enable the concurrent processing of multiple query jobs without impairing the job latency, it is advantageous to control the consumption of CPU-cores via block sizes, especially during periods of low data rates. Prior work mainly focuses on optimizing single-query scenarios in batch streaming systems [9, 4]. Others, considered the optimal query placement in a shared infrastructure with known resource requirements and data stream sharing capabilities amongst the queries [8].

In this paper, we particularly address the following research question: how to best execute primary and optional queries via dynamic block sizing such that the job latency target of the primary query is met and the throughput of a optional query is maximized in the best effort. We design a controller, iBLOC, which first dynamically adjusts the block sizes via controlling block intervals defined as the time between the generation of blocks. It aims to use the minimal number of blocks/tasks per primary job and its CPU-core consumption without violating the latency target. iBLOC then opportunistically uses the remaining cores to execute the optional query, particularly during periods of low data rates. We implement iBLOC on Spark Streaming. Our evaluation results show that iBLOC is able to effectively adjust the block sizes according to the data rates and to timely activate the execution of the optional query, in comparison to a static block-sizing scheme.

Our contributions are twofold. First, we enhance the performance of both the primary and optional queries under various load conditions on the limited CPU resources. The proposed controller, iBLOC, ensures the latency requirements of the primary query and opportunistically executes multiple queries on the same data set. Second, we increase the efficiency of the CPU-core utilization by adaptively controlling the block size and the parallelism level of tasks.

The outline of this work is as follows. Section II-B provides motivation examples on the importance of dynamic block sizing. The system architecture and algorithm of iBLOC are detailed in Section II and Section III, respectively. Section IV presents the experimental set up and results of the job latency

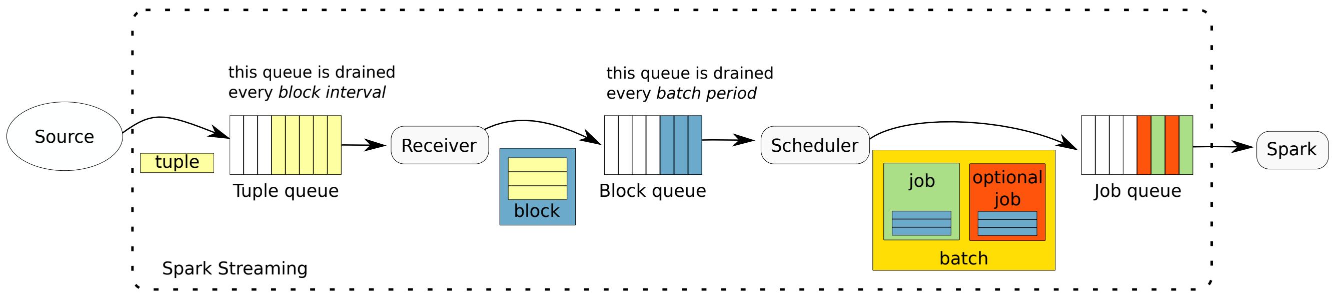


Fig. 1. Schematic of Spark Streaming: block, batch, and job generation.

and CPU-core utilization. Section V compares related work, followed by summaries and conclusions in Section VI.

II. System Architecture

In our architecture we consider Spark Streaming [18] as a typical, reference data stream processing system. Spark Streaming is a library on top of the Spark framework [14] and delivers data stream processing capabilities. In the following text, we first highlight the key components relevant to the generating processes of data, blocks, and jobs and then present a motivation example that shows the relationship between the job latency and CPU-core consumption on different block sizes.

A. Blocks, Batches, Jobs in Spark Streaming

Spark Streaming comprises two main components as shown in Figure 1: the *receiver* and the *scheduler*. The receiver accumulates incoming data from sources into *blocks* in fixed time intervals. The scheduler further batches many blocks together into *jobs* in fixed time intervals.

The flow of a query in Spark streaming is the following. First, data records, referred to as *tuples*, are generated from a source at various rates (e.g., 300 data records/ms). Every arriving tuple is stored in the *tuple queue*. The receiver generates a block by grouping together the tuples from the tuple queue every β ms, which is termed block interval. All generated blocks are stored in the *block queue*. The scheduler creates a batch by taking all blocks in the block queue every *batch period*, denoted by τ. The resulting number of blocks per batch is then given by τ/β. Afterwards, the scheduler decides how many query jobs, η, are going process this batch of the data. As we only consider primary and optional jobs, the number of jobs is thus $\eta = \{1, 2\}$. All generated jobs wait in job queue and different jobs can be processed concurrently at Spark in a first-come-first-served fashion. Here, we set the concurrency level to the maximum number of jobs per batch.

In Spark, each query job is divided into parallel *tasks*, one for each of the τ/β blocks in the batch. Each task occupies one computing slot which corresponds to one or more CPU cores. The discussion of optimal compute-unit configuration is out of the scope of this work and we set each slot equal to one CPU core throughout this work. Essentially, controlling the block intervals and batch periods decides the number of blocks as well as the parallelism level per job. In turn, the job execution time depends on the amount of data per task, the computation dependencies among tasks, and the overhead associated with scheduling each task. As pointed out by prior work [4], the job execution times exhibit a highly nonlinear relationship with the data size per block.

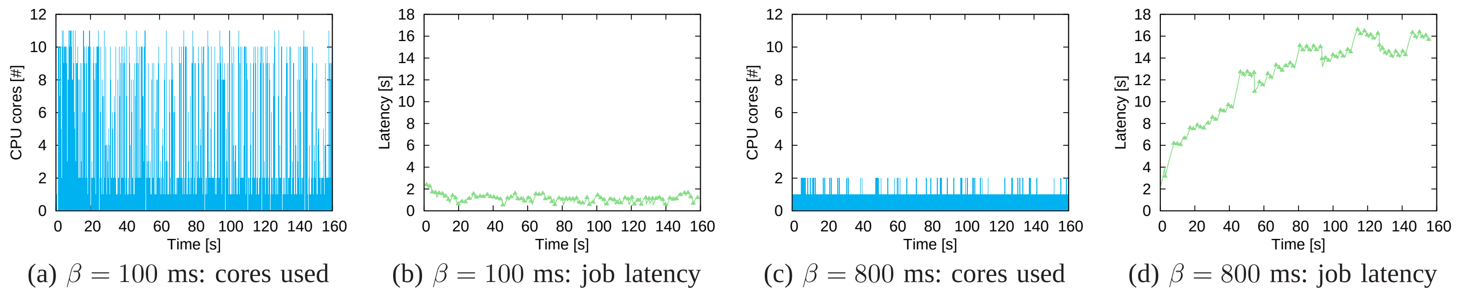
The resulting *job latency*, denoted by l, is computed as the time elapsed from the job generation and completion time. The target latency of the primary job is given by l^*. There are three key parameters affecting the job latency, i.e, the block interval β, the batch period τ, and the number of concurrent jobs η. Here, we are particularly interested in controlling β and η, while keeping τ as a constant user-specified parameter.

B. System Dynamics: a Motivating Example

We present a motivating example to illustrate the need for dynamically controlling the block intervals. Another purpose of this example is to outline the high-level design ideas of our proposed block interval controller, iBLOC. To this end, we execute a single query on the Spark streaming system, which receives data generated at a rate of 300 tuples/ms. We experiment with two constant values of block intervals, i.e., $\beta = 100$ and 800 ms. Additionally, we keep the batch period at $\tau = 1$ s. Spark Streaming is deployed on ten virtual machines each equipped with 2 virtual cores and 4 GB of memory. The details of the experiment testbed and the query can be found in Section IV-A.

For both cases, Figure 2 summarizes the CPU-core consumption and the job latency. On the one hand, when the block interval is small ($\beta = 100\ ms$, see Figure 2 (a) and (b)), the amount of data per block is small and the number of blocks/tasks per job is high. As a result, such a job occupies a large number of CPU-cores for executing tasks and the latency is low. However, there are no free CPU-cores available for other queries that might also operate on the same stream of data. On the other hand, using large block intervals ($\beta = 800\ ms$, see Figure 2 (c) and (d)), the amount of data per block is higher. The number of CPU cores occupied by tasks is thus lower and free CPU-cores are left for additional queries. Without any surprise, the system cannot cope with the incoming tuple rate and the latency explodes.

These results suggest that there could be certain block intervals able to strike a good tradeoff between job latency and CPU-core savings. As a result, one can use a smaller amount of CPU-cores, as long as the latency target of the primary job is met. The remaining spare CPU-cores can be used to execute the optional query. Furthermore, we argue that such block intervals vary according to the tuple rate. For example, during periods of low tuple rate, it can be advantageous to



(a) $\beta = 100$ ms: cores used (b) $\beta = 100$ ms: job latency (c) $\beta = 800$ ms: cores used (d) $\beta = 800$ ms: job latency

Fig. 2. Single query with two values of block intervals: showing CPU-cores consumed and job latency.

use long block intervals such that the number of tasks and occupied CPU-cores per job are low. One the contrary, during periods of high tuple rates, one shall use low block intervals to increase the parallelism level per job such that primary job latency target is met. In the following section, we consolidate these simple yet constructive ideas into the proposed controller, iBLOC.

III. iBLOC

In this section, we present the proposed solution, iBLOC, which is able to dynamically tune the number of concurrent jobs and blocks per job as a proxy for the number of parallel tasks. In particular, iBLOC controls the block interval and adjusts the number of jobs scheduled in each batch period, depending on the input tuple rates.

iBLOC consists of three main components: the Controller, the Model Estimator and the Load Classifier. Figure 3 shows an overview of how the three main components function together and interact with the Spark Streaming layer components. In the following we detail each iBLOC component.

A. Controller

The controller is responsible to dynamically compute the block interval β to adjust the number of parallel tasks so that the target job latency l^* provided by the user is met.A central idea of the controller is based on the estimation of the job latency. In particular, we propose to use a very simple way to approximate the job latency by the latency of a single task of that job, assuming there is a sufficient amount of cores/slots to run in parallel all the tasks of a job. At this point we assume parallel execution on a homogeneous infrastructure; we ignore any performance interference of co-executed tasks; and we finally ignore any data locality considerations. We will address these issues in future work. A simplistic way to estimate the task latency is to multiply the amount of tuples per block executed by a task by the runtime per tuple, where we assume the runtime to include the queueing time. As a result, we can write the job latency, l, as the product of tuple arrival rate r, block interval, and the runtime per tuple k, i.e.,

$$l = r \cdot \beta \cdot k.$$

To satisfy the target latency, l^* , iBLOC then sets the block interval as

$$\beta = \frac{l^*}{r \cdot k}. \tag{1}$$

Following Eq. 1, at the beginning of the control interval t iBLOC decides the block interval, β_t. Specifically, it uses the estimates of the tuple arrival rates, $\hat{r_t}$, and the tuple runtime, $\hat{k_t}$ for the same interval t:

$$\beta_t = \frac{l^*}{\hat{r_t} \cdot \hat{k_t}}.$$

The values $\hat{r_t}$ and $\hat{k_t}$ for each interval t are calculated by the Model Estimator component explained in Section III-B.

To minimize the amount of resources while keeping the system stable, a natural choice is $l^* = \tau$, i.e. the batch period, meaning that all tasks of a job can be finished before new ones arrive. As for the control interval length, we set it to the batch period, which is the maximum time duration before any new tasks can enter the system in a stable region.

Moreover, we apply an anti-windup unit to prevent the controller from saturating. The upper-bound of block intervals, $\bar{\beta}$, is naturally given by the batch period. As a results, we set $\bar{\beta} = \phi \cdot \tau$, where $0 < \phi \leq 1$ avoids creating batches with no blocks and still allows batches with only one block. Throughout the paper we empirically set $\phi = 0.8$. The lower-bound, $\underline{\beta}$, is given by the maximum parallelism available in the system, i.e. the number of slots, denoted by C. Here, we particularly consider the case where the number of slots equals to the number of computing cores. Note that we consider constant values of the upper bound and lower bound of block intervals. To conclude, the final control law used in iBLOC is as follows:

$$\beta_t^* = min(max(\beta_t,\ \underline{\beta}),\ \bar{\beta})$$

B. Model Estimator

The iBLOC controller needs to estimate k_t and r_t as model parameters, which, however, are unknown upon the decision time and often time-varying. To overcome this problem, we estimate these parameters online using an exponential weighted moving average:

$$\hat{r_t} = (1 - \alpha) \cdot \hat{r_{t-1}} + \alpha \cdot r_{t-1}$$

$$\hat{k_t} = (1 - \alpha) \cdot \hat{k_{t-1}} + \alpha \cdot k_{t-1}$$

where α is a parameter called *forgetting factor* and defines the weights of historic samples in the estimation. The higher the value of α, the lower the effect of historic samples.

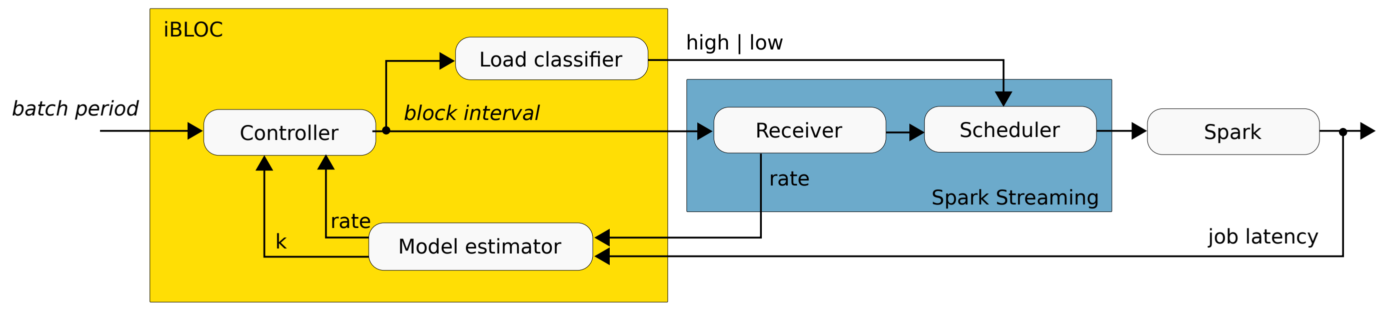


Fig. 3. iBLOC design

C. Load Classifier

In addition to the block period β^*, iBLOC also decides if the optional job shall be executed depending on the tuple rates. At the beginning of every control interval, iBLOC sends a boolean variable to the Spark Streaming scheduler, indicating the execution, or not, of the optional job. The idea here is to leverage the spare slot/core resources during low tuple rates to execute additional optional analysis on the same input data without violating the latency target of the primary one. High values of β^* imply few blocks/tasks per batch period and there could be many unused slots/cores for additional tasks. Consequently, at the beginning of interval t, iBLOC sends the signal of "low" load to the scheduler, if β^* is greater than a threshold value, which we specify as $0.9 \cdot \bar{\beta}$. Once the scheduler receives such a signal, it executes one additional optional job. Otherwise, iBLOC sends the "high" signal, so that only the main job is executed at interval t. We note that the threshold value and the number of optional jobs to start can have a significant impact on the latency of the primary job and it is our future work to optimize these variables. In summary, the load signal sent to the scheduler is computed as follows,

$$load_t = \begin{cases} low & \text{if } \beta_t^* \geq 0.9 \cdot \bar{\beta} \\ high & otherwise. \end{cases}$$

IV. Evaluation

In this section, we describe our evaluation results on Spark Streaming. Using as an example of the log analyses at a server farm, we design and combine query jobs performing different analyses and show how iBLOC copes with them. More in detail, focusing on evaluating the latency of the primary job, the cluster CPU-core consumption, and the throughput of an optional job, we show that iBLOC works on (i) jobs of different complexity and (ii) combinations of a primary plus an optional job.

A. Testbed

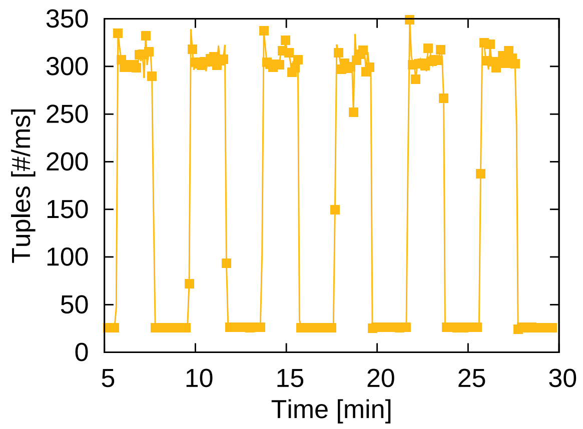
We deploy Spark version 1.3.0 on ten virtual machines (VMs) running Ubuntu 14.04. The VMs are hosted on a private cloud and each VM is equipped with two virtual cores and 4 GB of memory which corresponds to an Amazon EC2 t1.medium instance with half the memory size. One VM acts as the Spark master while the other nine as Spark workers. The Spark master coordinates the Spark cluster whereas the workers run the data processing.

As an example use case we emulate the log analysis of a server farm that monitors the overall system status. In more details we count the number of different log messages grouped by severity level, i.e., error, warning or info, and message codes. The resulting map-reduce job is similar to the popular WordCount benchmark [6]. We use a custom load generator written in the Go programming language to stream tuples emulating log entries composed of a severity level, message code, timestamp and host. The load generator runs on a separate physical machine to prevent it from being the bottleneck of the system. Preliminary tests show that the load generator can easily achieve up to 3,000 tuples/ms using less than 30% of the bandwidth available between the load generator and the Spark cluster. To emulate different data rates and stress the system, the load generator alternates between low rate periods and high rate periods.

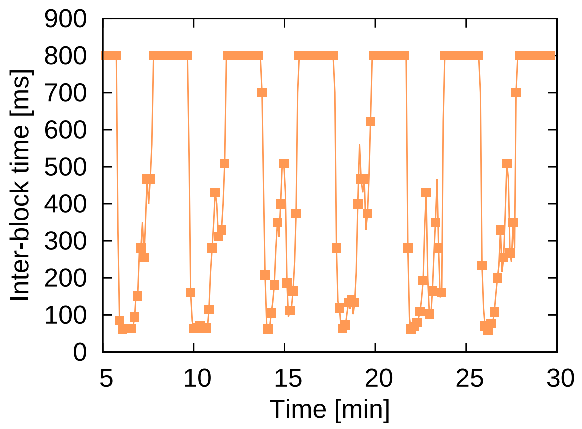
B. Job One

We start with the same scenario used in our motivating example in Section II-B using a single job which analyses systematically all the streamed log entries. Here, we enable our iBLOC controller and vary the input tuple rate to stress the controller. We set the high tuple rate to 325 tuples/ms and the low tuple rate to 25 tuples/ms and alternate between them every 120 seconds. The low tuple rate can be handled by a single CPU core without violating the target job latency l^*. Whereas, the high tuple rate saturates the Spark cluster and requires almost all the 18 cores of the Spark workers to be able to meet l^*. Figure 4 shows the job latency and the CPU cores usage as we vary the input tuple rate and the iBLOC controller sets the bock intervals.

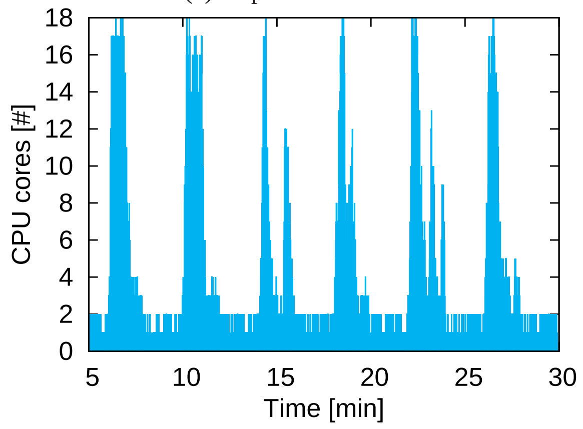
We set the batch period $\tau = 1$ s and the target primary job latency equal to the batch period $l^* = \tau$, and we use these values throughout all experiments. As expected, on the one hand, during low periods of tuple rate the controller increases the block interval (Figure 4 (b)) until it hits the anti-windup boundary, limiting the number of blocks per batch period and, consequently, the number of tasks and CPU core usage. Looking more closely at the CPU core consumption (Figure 4 (c)) we see that during periods of low tuple rates it oscillates between one and two cores. The reason is twofold. (i) ϕ is set to 0.8 and consequently the anti-windup boundary is set slightly lower than the batch period. Therefore, some batch periods see two blocks instead of one. (ii) When the receiver is scheduled, it also uses one computing slot. i.e., one CPU core. On the other hand, during high tuple rates the controller reduces the block interval increasing the number of



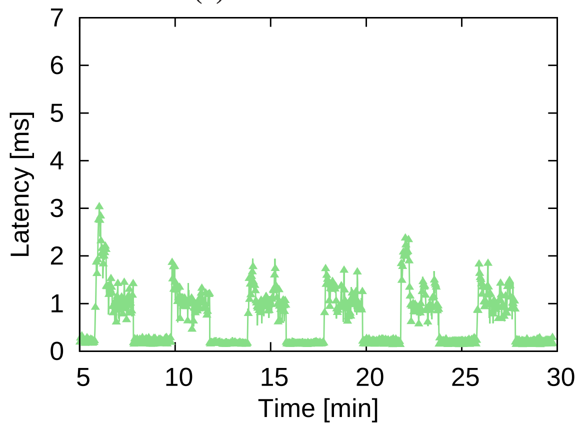
(a) Tuple arrival rate



(b) Block intervals



(c) Cores used



(d) Job latency

Fig. 4. System results with a complete log analysis which considers all messages.

blocks, and consequently tasks, per batch period to up to 18, i.e., the number of cores provided by all the available Spark workers, in fact adapting the parallelism level to the tuple rate. Without this increase, the system would be overloaded and the latency would explode, whereas here the controller is able to limit the job latency to the target value (Figure 4 (d)). Overall, the controller uses efficiently the system resources while bounding the latency to a target value. It does so by freeing up unused resources during periods of low tuple rates and claiming enough resources during periods of high load of tuple rates.

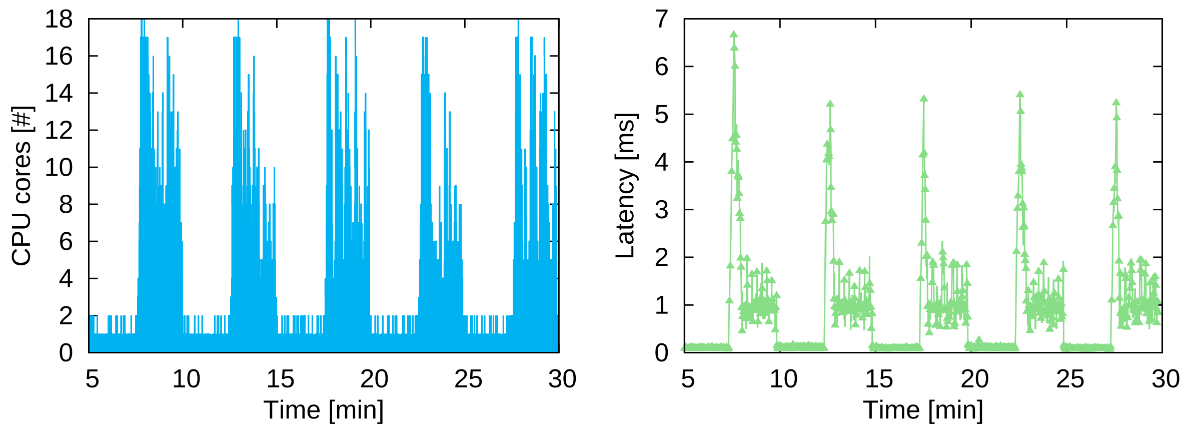
C. Job Two

Here we try to increase the supported maximum tuple rate by simplifying the performed log analysis. More in detail, we modify the job to process only the more critical error messages ignoring other messages. Using this job we were able to increase the maximum achievable tuple throughput. Hence, we load the system alternatively with 25 tuples/ms and 700 tuples/ms every 150 s. For sake of brevity, hereon we skip the arrival rate and block interval plots and only show the CPU core consumption and job latency.

Again, one can observe low number of consumed CPU cores during periods of low tuple rates and high number of consumed CPU cores during periods of high tuple rates (see Figure 5 (a)). At the same time the job latency is kept close to the target by well adapting the system resources (Figure 5 (b)). The goal of this scenario is twofold. First, we show that simplifying the job it is possible to increase the sustainable throughput. Second, we show that the controller is able to cope not only with different input rates but also with different query jobs. Moreover, this is achieved without any manual tuning of the controller itself. The controller configuration parameters depend only on the Spark cluster characteristics and not on the input data or query job.

D. Primary plus optional job

From the previous two experiments, we can draw two key observations: (i) by simplifying the job we increased the achievable system tuple throughput, but at the cost of information loss, i.e., only error messages are analysed; (ii) the system is fully utilized only at high tuple rates, leaving plenty of CPU cores at disposal during low tuple rates. To take opportunity of the unused CPU cores instead of defining one simplified job, we split our log analysis into two: a primary job and an optional job. The primary job does exactly the same analysis as in our second experiment processing only error messages, while the optional job processes all other types such as warning and information messages. The primary job is scheduled in every batch period, whereas the optional job is executed only during periods with low tuple rates. We set the tuple load to be equal to our second experiment, i.e. alternating between 25 tuples/ms and 700 tuples/ms every 150 s, since



(a) Cores used (b) Job latency

Fig. 5. System results with a partial log analysis concentrated on errors only.

during periods of high tuple load the system only runs the primary job. To highlight the execution of the secondary jobs, in addition to the CPU core consumption and primary job latency, we summarize in Figure 6 the number of tasks of the secondary job.

Starting with Figure 6 (a), one can see that the controller based on the load classifier starts secondary jobs only during periods of low tuple loads while no secondary jobs are started during periods of high tuple loads. The effect on CPU core consumption is also visible. Looking at Figure 6 (b), the CPU core consumption is roughly doubled during periods of low tuple loads. In fact instead of a single primary job, both primary and optional job run doubling the tasks executed in parallel. Finally, since optional jobs are started only during periods of low tuple loads, the system still achieves its latency target as seen in Figure 6 (c).

V. Related Work

Recent data-parallel processing systems such as Spark Streaming [18, 15], SEEP [2], Storm, and Naiad [10] process streams of data in real-time while focusing on providing a scalable and fault-tolerant substrate. An important task in such systems is the setting of the number of tuples into batches. Although the batch size can have direct influence on the performance of queries, adjusting the number of tuples per batch has not received considerable attention. The closely related work to our study is [4], which dynamically controls the batch periods and keeps the block intervals and the number of concurrent jobs constant. In contrast, our study is orthogonal on optimizing the block intervals and concurrency level of jobs such that multiple different analysis can be concurrently delivered on the same set of data.

To optimize the job latency in data stream processing systems, several different types of control knobs have been considered. Typically, admission control by data shedding is applied particularly on overloaded situations either in a probabilistic random fashion [16] or for certain types of operators, such as joins [3]. In [7] a feedback controller is presented to control the queueing time and ultimately the job latency to match user-specified targets. Another type of control knob is resource scaling based on the data load [2]. Most of work is discussed under the single application scenario, except [13, 11]. When considering multiple queries at-once, both work argue that the resource allocation to queries can significantly affect query performance. However, it is not clear how optimization techniques developed for continuous models can be effectively applied on the most recent data stream processing systems which is the focus of this study.

VI. Conclusion and Discussion

In this paper, we aim to enable the concurrent execution of primary and optional queries on the limited CPU-core resources of a data stream processing cluster, while adhering to latency constraints. We develop a controller, iBLOC, which dynamically adjusts block sizes processed by tasks and timely execute optional queries on spare CPU-cores. Specifically, iBLOC generates larger blocks and execute both types of queries during low data rates, whereas during high data rates iBLOC generates small blocks and executes the primary query. Our early evaluation results show that iBLOC is able to use the core resources to execute both queries without violating their latency targets, when compared to a scheme with constant block sizes. As part of future work, we plan to extend iBLOC to control the batch size of a fully operational cluster executing multiple queries with different latency and resource requirements. Our controller is currently designed to work with the Spark Streaming architecture. In future work we plan to evaluate our controller against other data-parallel frameworks.

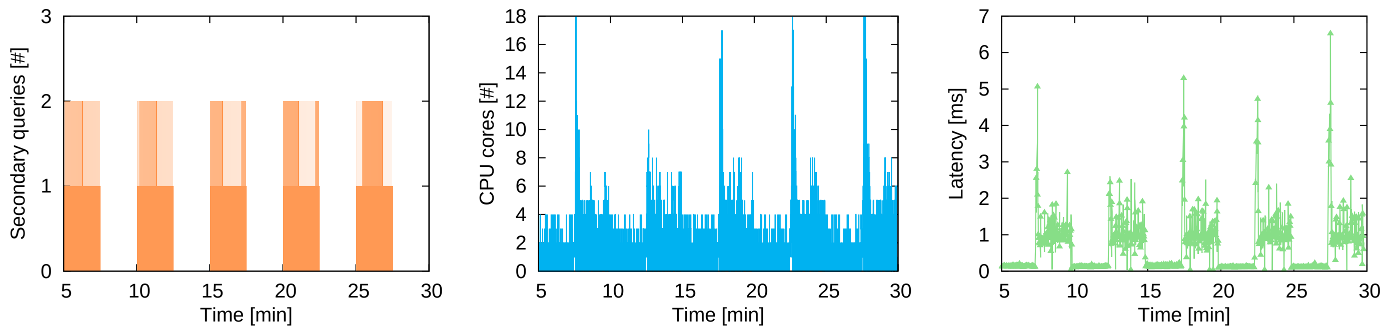
References

[1] A. F. P. (AFP). Twitters beat media in reporting china earthquake, 2008.

[2] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM SIGMOD*, pages 725–736, 2013.

[3] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *ACM SIGMOD*, pages 40–51, 2003.

[4] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *ACM SoCC*, pages 16:1–16:13, 2014.



(a) Optional tasks (b) Cores used (c) Job latency

Fig. 6. System results with both an error log analysis as primary job and log analysis of the other messages as optional job during periods of low tuple arrival rates.

[5] 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 *ACM SoCC*, pages 63–74, 2010.

[6] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshop*, pages 41–51, 2010.

[7] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch. Overload Management in Data Stream Processing Systems with Latency Guarantees. In *Feedback Computing Workshop*, 2012.

[8] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: stream query planning with reuse. In *ICDE*, pages 840–851, 2011.

[9] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. In *SIGMOD*, pages 943–954, 2014.

[10] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *ACM SOSP*, pages 439–455, 2013.

[11] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, page 49, 2006.

[12] E. J. Ruiz, V. Hristidis, C. Castillo, A. Gionis, and A. Jaimes. Correlating financial time series with microblogging activity. In *WSDM*, pages 513–522, 2012.

[13] S. Seshadri, V. Kumar, B. F. Cooper, and L. Liu. Optimizing multiple distributed stream queries using hierarchical network partitions. In *IPDPS*, pages 1–10, 2007.

[14] Spark. https://spark.apache.org.

[15] S. streaming. https://spark.apache.org/streaming/.

[16] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[17] Twitter. http://www.informationweek.com/big-data/big-data-analytics/ibm-twitter-offerings-yield-business-results-/d/d-id/1319498.

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