Optimizing Latency and Throughput Trade-offs in a Stream Processing System

Joao Carreira, Jianneng Li

Abstract

The value of stream processing systems stems largely from the timeliness of the results these systems provide. Early stream processors followed the record-at-a-time approach, servicing each data point as soon as it arrives at the system. While these systems provide good latency, their behaviors become less desirable when applications require high throughput, fault tolerance, or usage of stateful or blocking operators. More recently, systems are developed to follow the micro-batch approach, where many records are processed together as small batches, providing the missing features seemingly at the cost of latency.

Given the situation, we look into a micro-batch streaming system called Spark Streaming, and investigate how well the micro-batch architecture can handle latency-sensitive workloads. We instrumented Spark Streaming to understand where time goes and where development effort should be focused on. In this paper we make three contributions. First, we provide an analysis of the performance of Spark Streaming, showing the average time breakdown within the system. Second, we identify the performance and scalability bottlenecks of Spark Streaming and pinpoint the underlying deficiencies of the system. Last, we propose and evaluate several optimizations to reduce the system overhead and achieve lower latency while maintaining throughput.

1 Introduction

Data analytics applications such as intrusion detection, web search and monitoring, require the computation of results in a timely fashion in order to provide interactivity and responsiveness. As enterprise workflows grow increasingly complex and applications become increasingly dependent on other applications, the latency guarantees of their processing systems become a major concern. For instance, many systems today are designed to meet specific performance metrics or SLOs [14]. Applications that do not yield good performance may delay the execution of other systems and lead to loss of revenue and/or bad user experience.

Many data analytics systems [2–4, 7–12, 16, 17, 19, 20, 23] have been developed to provide easy-to-use and practical frameworks for stream processing, and they tend to follow one of the two popular approaches to act on inputs. Systems like Storm or TelegraphCQ handle streams of data by creating pipelines for record-at-a-time processing. In this environment, data flows through the system (potentially through different machines) and is continuously processed and augmented. Other systems like Spark Streaming and Trill rely on micro-batches. In these systems, records are coalesced into small groups before being processed together as batch jobs.

It is widely believed that to achieve low latency, applications should use the record-at-a-time approach, while for high throughput, the micro-batch model is more appropriate. The reason is that even though record-at-a-time systems can start processing data as soon as it is received, processing more records at a time will amortize the overheads associated with processing. However, because records are not processed right away, the latency per record is higher on average. In this paper, we are generally concerned with end-to-end latency, which the time between the application sending to data and the application receiving the output.

The trade-offs between latency and throughput become less simple as applications are scaled out to run in a distributed fashion. In a distributed environment, the record-at-a-time approach has many disadvantages when compared with the micro-batch approach. First, record-at-a-time systems are not suitable for stateful or blocking operators, as these operators by nature will unboundedly increase memory usage or stall the system. Although techniques such as punctuations [21] can be used, they place extra burden on the application. For micro-batch systems, because batches are of finite size, behaviors of stateful and blocking operators are much easier to define. Second, record-at-a-time systems require replication or upstream backup techniques to tolerate failures. Neither of these solutions are desirable, because the former requires the usage of fail-over hardware, and the latter usually leads to high recovery times. When records are batched, fault tolerance can be implemented with methods such as backing up to a database [20] or keeping track of lineage information of batches [23]. Finally, the programming model for micro-batch systems is very similar to that of a traditional batch system, making the process of developing streaming applications more familiar to developers.

Because of the advantages in the micro-batch model in distributed settings, we want to explore the possibilities of using this model to provide better end-to-end latencies. In this paper, we perform an in-depth analysis of Spark Streaming, a micro-batch streaming engine, and evaluate the system's ability to provide low-latency and high throughput stream processing. In particular, we intend to answer the question: "Are the performance limitations of Spark Streaming a consequence of its architecture, or the result of engineering decisions?" Upon evaluating the system, our answer is: both. Although it is true that the percentage of the end-to-end latency spent in useful work is low, especially for small computations, some of the overheads can be reduced relatively easily. We focus on two such areas, task overheads and data storage speed, and present our results after the optimizations.

While the focus of our analysis is in Spark Streaming, we believe that our analysis and conclusions can inform the design and development of other streaming systems using the micro-batch architecture. The reason we chose Spark Streaming is that there is a fast-growing ecosystem of frameworks such as GraphX, MLlib, and SparkSQL being developed for Spark, the engine on top of which Spark Streaming is built. As a result, our work can potentially have large impact, since developers will be able to take advantage of both the rich tools as well as low end-to-end latency without sacrificing throughput.

The paper is organized as follows: in Section 2 we provide an overview of the architecture and workflow of Spark Streaming. In Section 3 we motivate this work with a performance study of Spark Streaming across two main dimensions: end-to-end latency and throughput. In Section 4 we present optimizations aimed at solving some of the architectural deficiencies identified in the previous section. In Section 5 we present some of the lessons gathered during this work and discuss some of the architectural changes we believe are required to make Spark Streaming provide lower latencies. In Section 6 we describe how some other contributions relate and complement our work. Finally, in Section 7 we summarize our research and discuss the next steps.

2 Background

In this section, we provide a short description of Spark Streaming's architecture as well as relevant terminologies needed to understand our work.

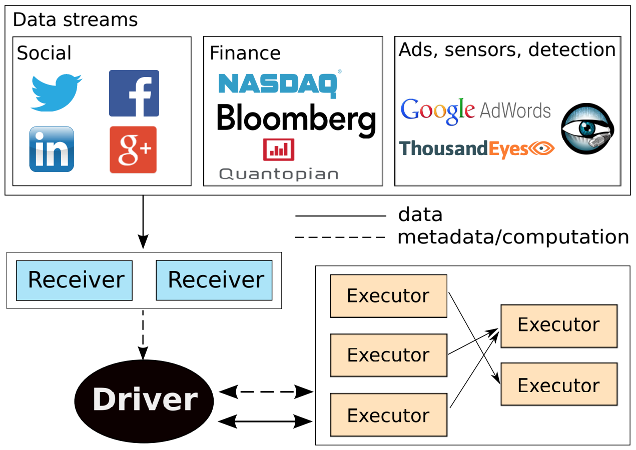

Figure 1: Diagram of a Spark Streaming work flow. A Receiver and an Executor may execute in the same node.

Spark [22] is a general-purpose engine for large-scale cluster computing. It operates on Resilient Distributed Datasets (RDDs), an abstraction that represents read-only collections of objects partitioned across a set of machines. RDDs can be created from raw data or other RDDs, and keep track of the coarse-grained operations (e.g., map, reduce) performed on its underlying data as lineage information, so that partitions can be recomputed if they are lost due to failures. Another feature of RDDs is that they can be persisted in memory, allowing efficient iterative computations.

Spark Streaming, the focus of our work, is a stream processor built on top of Spark. It implements an abstraction called discretized streams (D-Streams), which takes advantage of RDDs and their lineage properties to divide potentially infinite streams of data into finite chunks and provide an interface to perform computation over it. The abstraction allows a micro-batch approach to streaming data, yielding high throughput, scalability and fault-tolerance.

The execution of a Spark Streaming job works as depicted in Figure 1. First, data is generated at a source (e.g., tweets from Twitter). As the data is pushed into Spark Streaming, it is received by a Receiver, which in turn stashes it in memory. After every set amount of time called the block interval, the Receiver takes the data stashed in memory and generates a block with it.

Once a block is generated, the Receiver informs a Spark Streaming's central component called Driver about this block. The Driver is responsible for holding metadata about reported blocks from all Receivers. After a separate, larger fixed amount of time called the batch interval or batch window, the Driver takes all blocks that have been communicated by the Receivers and that have yet to be processed and generates a batch job with them. Generated jobs are passed to the Scheduler (not shown in diagram) to be scheduled and run on machines available in the cluster.

Every batch job is divided into one or more stages. Each stage can be a map stage or reduce stage. Stages have to be processed in sequence, so the more stages required for a job, i.e. the more reduce operations used by the application, the longer a job will take to execute.

Each stage operates on a RDD. The RDDs are partitioned by the blocks generated during the batch interval, and can be processed in parallel. The computation on a partition is called a task. Tasks are sent from the Scheduler to run in stateless environments called Executors. Executors are usually deployed in the same nodes as Receivers; in fact, Receivers are implemented as long-running tasks.

At the heart of the micro-batch approach taken by Spark is the trade-off between the amount of records Spark can process per unit of time and the time the system takes to return results to the user. On one hand, waiting for more records to generate bigger batches allows Spark Streaming to amortize its overheads. On the other hand, the more time the system waits for data, the less responsive the system becomes.

Internally, Spark Streaming makes extensive use of the producer-consumer design pattern, where separate threads offload work to each other through queues. For example, as blocks are generated at the Receiver, they are pushed to a queue by a producer thread. The corresponding consumer thread constantly polls the queue, and stores the dequeued block as well as sends metadata about the block to the Driver. As well, communications between different components of the system is done through the Actor model, where components ("actors") talk to each other by sending messages to each other's message queues.

3 Motivation

To better understand the performance and limitations of the Spark Streaming architecture, we conducted a benchmark study of the system. We added instrumentation code that allowed us to track the timestamps of a subset of the input data as it flows through the system.

A synthetic workload was used for our performance testing on Spark Streaming. This workload consists of an application that listens for a stream of text records. The records have sizes between 15-25 bytes, and each holds a unique ID and a timestamp of time when the data was generated. For each micro-batch, the application computes the difference between the current time and the time specified by the record, and use it to define the end-to-end latency of that record.

This workload only requires one stage per batch, since no aggregations are needed. As well, the work done by each task is very lightweight, since it only parses numbers and perform subtractions. Despite the near-trivial computation, we believe that this application does is comparable with computations carried out on streaming systems. Due to latency constraints, fast streaming applications cannot be overly complex. They are likely to use filtering and simple aggregations, which are operations that can scale linearly with the resources available. Also, often work can be completed more quickly with smart pipelining or different levels of parallelism, both of which are out of the scope of this paper. Our goal is to look at the code path that every workload uses, and identify and minimize the bottlenecks.

To run this benchmark, we first developed a record generator that can generate the type of workload described previously. Our record generator takes a record size and a data rate as input, listens on a socket for connections. For each incoming connection, it continuously generates as many records as needed to reach the specified data rate per second, until the connection or the generator is killed.

We deployed the generator and the Spark Streaming application on a dedicated cluster of machines, each equipped with a 16-core Intel Xeon CPU E5-1680 v2 @ 3.00GHz and 64GB of RAM. Each Receiver was run on a separate machine, and connected to a data generator for input.

The results of our experiment are shown in Figure 2. This graph displays the average end-to-end latency obtained when running Spark Streaming with five different batch window values and two different throughput levels. Changing the batch window configuration allows us to tune the responsiveness of the system: a lower value means that each records on average spends less time in the Receiver, waiting to be put into a batch and subsequently processed by a task. Varying the number of records generated by the stream source (throughput) allows us to understand how Spark behaves when it has to do more or less work per unit of time and how that affects latency.

As expected, as we instruct Spark Streaming to spawn tasks more frequently – smaller batch window – the average end-to-end latency time decreases. However, at some point decreasing the batch window has a negative effect on the resulting latency.

We also find that as we increase the throughput, the end-to-end latency increases. When analyzing the reason, we determined that Spark Streaming's Receiver can be a source of slowdown. For instance, we found that one Receiver is not able to receive more than roughly 30 MB/s, or 1.5M records/s. This has to do

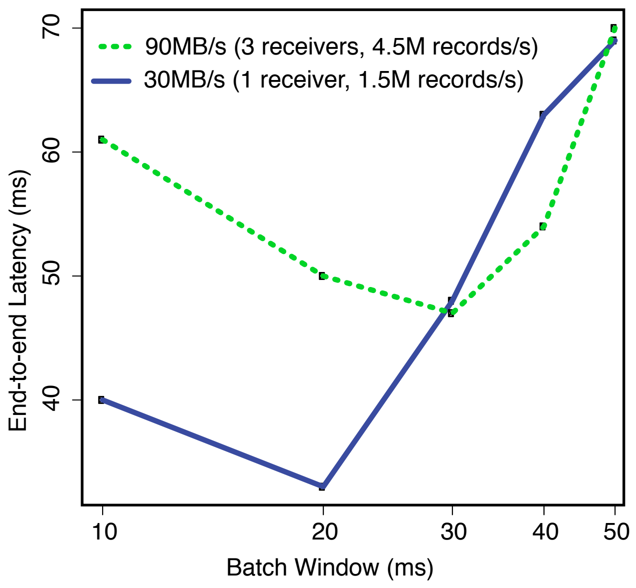

Figure 2: Average end-to-end latency for different batch window sizes, different throughputs, and a fixed block interval of 2ms. For batch windows smaller than 10ms the average end-to-end latency tens to infinity.

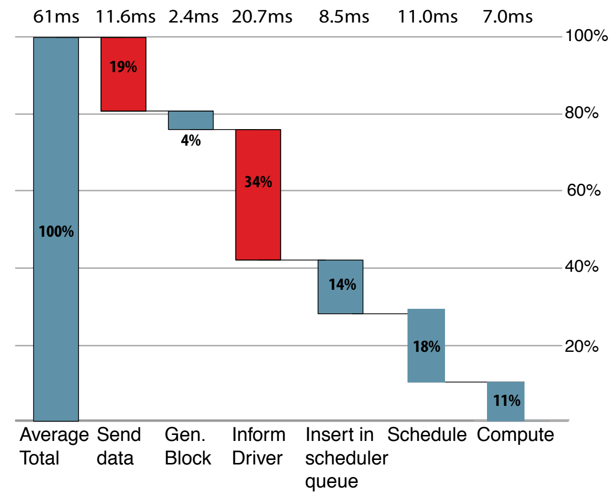

Figure 3: Execution time breakdown for a batch window of 10ms at 90MB/s. The timestamp of the data is collected when 1) the data generator generates the record, 2) the Receiver receives the record, 3) the record is put inside a block, 4) the block is stored and communicated to the Driver, 5) the Driver puts the block metadata inside a batch, 6) the batch is scheduled, and 7) the record is processed by a task.

with the fact that the Receiver receives and stores data in a sequential fashion within a single-threaded loop.

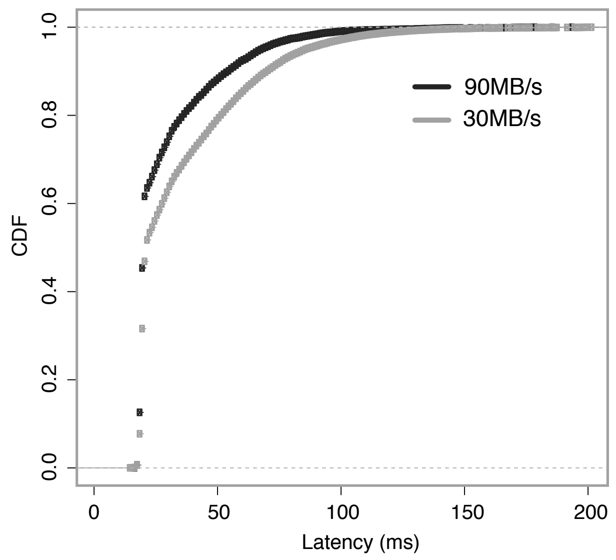

Figure 4: CDF of latencies for a batch window of 10ms at 90MB/s and 30MB/s.

As a result, for batch windows less than 10ms we found the system to be unstable. Because Spark Streaming is not able to process records as quickly as the rate of arrival of new records, the end-to-end latency increases indefinitely.

Next, to understand where time is spent, we decomposed the execution time of Spark Streaming to different phases, based on our instrumentation. We chose the top left point of Figure 2, i.e. batch window of 10ms and throughput of 90MB/s, and show the breakdown in Figure 3. This graph provides several insights. First, at most 11% of the execution time is useful time. Second, roughly 20% of the time is spent waiting for data to be transmitted from the source of the data to the Receiver. This time has to do with the single-threaded loop design of reading tuples from the network. Third, more than one third of the time is spent between generating a block of data and informing the Driver about blocks received. This time stems from the fact that the Receiver reserves one single thread to continuously store blocks and send the respective metadata to the Driver. The approach does not seem to scale to workloads with a high number of blocks generated per unit of time. Lastly, as expected, the time spent to schedule a micro-batch record is 11ms. Given a batch window of 10ms, each record will have to wait roughly 10ms to be scheduled.

Figure 4 shows a CDF of the end-to-end latencies for different throughput levels. The graph shows that a considerably-sized tail of records is served with high latency, implying high variance in end-to-end latency.

One caveat is that the record we choose to track is always the first record in the first block of every batch. This means the record is one of the first records arrived for the batch, and among the ones that waited the longest. Therefore, the median end-to-end latencies for the shown data points should be the displayed latency minus half of block interval as well as batch interval.

Overall, we believe these results motivate the need for an architecture that can scale and adapt to bursts of data in order to consistently provide acceptable latencies.

4 Implementation

From Figure 3, we gained an understanding of how different components of the Spark Streaming architecture contribute to the end-to-end latency. In this section, we focus on optimizing two areas: the overhead of running tasks, and the time it takes for a Receiver to store a block and inform the Driver.

4.1 Task Overheads

According to Figure 3, around 11%, or 7.0ms is spent on average executing a task. However, since our synthetic benchmark performed near-trivial computation, most of the time spent in running the task is due to the overheads in running a task.

To breakdown the process of running desks, we modified the Receiver so that it generated a block regardless of the number of records received. We then ran the system on empty input, using an application that required a single stage. The application is launched with 1 Receiver, running on the same machine as the Driver. Since there was no input, the computation itself was effectively a no-op. After profiling this workflow, we found that from the Driver's perspective, the average time of running a task, i.e. the time between the task is scheduled and the time between the result is received, was 5.0ms. The difference of 2.0ms was likely due to the fact that there was only 1 Receiver, running on the same machine as the Driver, under a very light load.

As we looked deeper into the 5.0ms, we found that approximately 3.6ms out of this time was in deserializing the task. These numbers suggest that if we can reduce the task deserialization time, there will be a considerable improvement to the overheads of running tasks.

Having discovered that a significant portion of the time running small tasks is spent in deserialization, we further measured the time it took for individual components of deserialization to complete. Figure 5 summarizes the process of task deserialization on the

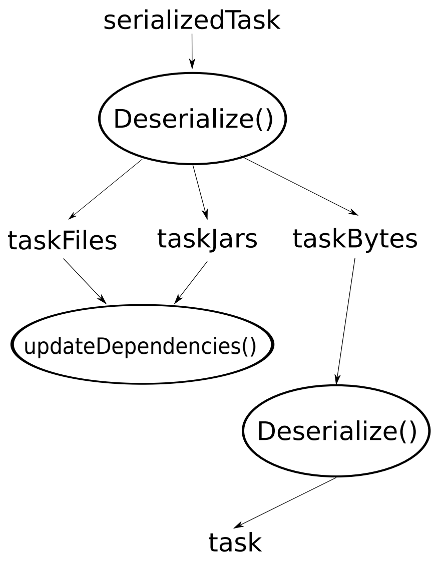

Figure 5: How a task is deserialized on the Executor. Examples of new dependencies can be new libraries needed to run the current task.

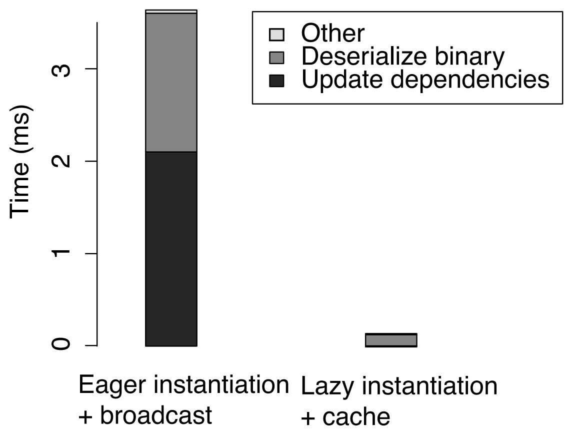

Figure 6: Breakdown of average time spent deserializing a task before and after adding lazy instantiation of configuration object and caching task binaries.

Executor. The Executor receives a task in the form of an array of bytes called `serializedTask`. This array is deserialized into a tuple of `taskFiles`, `taskJars`, and `taskBytes`, the first two of which are passed into a method called `updateDependencies()`, while the latter is further deserialized into a `task` object. The `task` object contains information such as the function to run, the RDD to use, and the partition of the RDD to operate on.

The left part of of Figure 6 shows a time breakdown of deserialization. As can be seen in the graph, the majority of the time in this case is in updating dependencies and deserializing `taskBytes`. We next examine each of these two components in more detail.

4.1.1 Update Dependencies

In `updateDependencies()`, Spark creates a configuration object, and use it when dependencies need to be updated, for example to download additional libraries used by the task. However, this object is created regardless of whether new dependencies are introduced, and this objection creation is very costly in CPU cycles. Since a streaming application rarely introduces new dependencies once it starts to run, this method is incurring unnecessary costs. To solve this problem, we changed the object to be lazily instantiated, so that no cycles are wasted creating the configuration object unless new dependencies are introduced.

4.1.2 Deserialize Binary

In order to reduce the amount of duplicate data transferred, Spark wraps the function and the RDD of a task into a broadcast variable, and serializes the broadcast variable as a part of `taskBinary`. This way, those information are not sent a part of the task, but is pulled by the Executor when it reads the value of the broadcast variable. The advantage of broadcast variable is that once it is pulled by the Executor, the value of the variable is cached in its memory. Therefore, if multiple tasks in the Executor use the same function and RDD, only the first task will need to pull the information from the Driver, and the rest can read it from the cache, reducing network traffic.

This approach, however, is still too wasteful in a streaming environment. Similar to the rarity of new dependencies, the types of RDDs and functions to operate on them change little from batch to batch. With the current approach, one round trip is still required per batch to fetch the function and RDD, even though information could be derived from a previous batch. As the batch interval shrinks, this inefficiency becomes more apparent.

To eliminate this round trip, we experimented with caching of information on the functions and RDDs. The cache is implemented by keeping track of previously broadcasted information on the Driver, and resend the previous broadcast variable if possible rather than creating a new one every time. Broadcast variables are automatically cached on the Executor side, so this methodology removes the extraneous communication with the Driver.

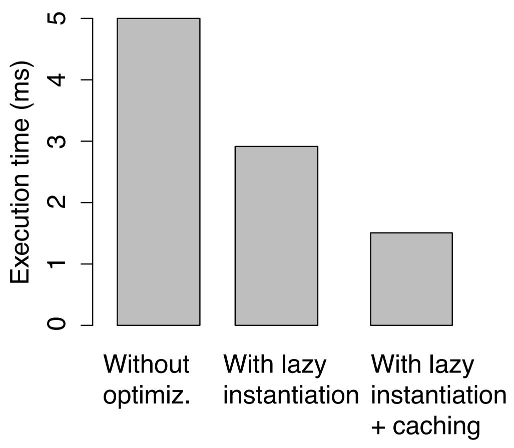

Figure 7: Average time spent in running no-op tasks before and after the optimizations in deserialization.

4.2 Block Storage

As mentioned in Section 2, Spark Streaming heavily uses the producer-consumer pattern, both through its usage of queues with pushing and polling threads as well as the Actor model. This pattern has many advantages, namely modularity, and the ability to explicitly shows bottlenecks in the pipeline. If a downstream processing is slow, a pushback mechanism can be used, where the upstream is blocked from inserting more items into the queue until the queue has free space.

We looked more closely at the section of the system responsible for storing blocks and sending their metadata to the Driver, and found the pushback to be the culprit. More specifically, one thread coalesces records into blocks, and puts them into a queue. Another thread polls the queue, and performs the storage and Driver communication. The consumer in this queue cannot keep up with the producer, introducing delays. To solve this problem, we spawned more threads to act as consumers.

4.3 Evaluation

The improvements of the two optimizations in task deserialization are also shown in the right half of Figure 6. After the changes, deserialization time for a task decreased from 3.6ms to 0.2ms. The impact of the two changes on overall task runtime is reflected in Figure 7.

To show the effect of reducing task runtimes, we performed a micro-benchmark running a single stage with many tasks. The results are shown in Figure 8. For 10K and 100K tasks, lazy instantiation of the configuration object achieves a speedup of 2-3 times.

To examine how the number of consumers storing

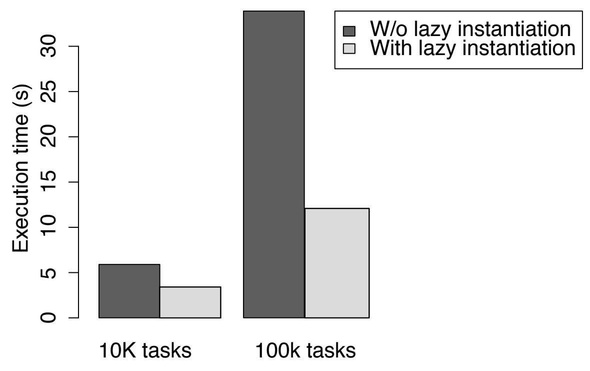

Figure 8: Amount of time spent running a single Spark stage consisting of many tasks without and with lazy instantiation of configuration object.

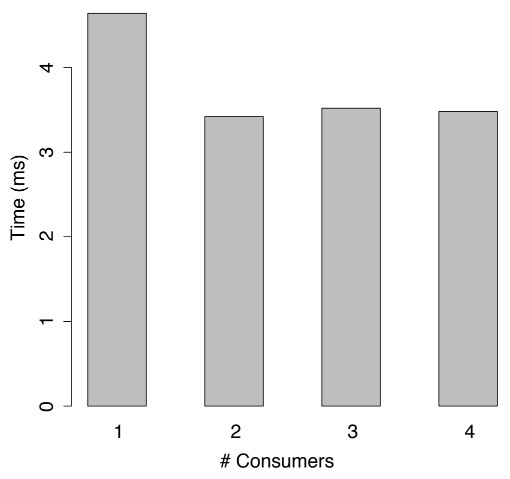

Figure 9: The average time taken to store blocks and send the Driver their metadata, with respect to the number of consumer threads polling generated blocks from the queue. The Receiver and Driver are on the same machine, with data rate of 60MB/s, batch interval of 10ms, and block interval of 2ms.

blocks changes latency, we ran a benchmark with the Receiver and Driver on the same machine, on a data rate of 60MB/s, batch interval of 10ms, and block interval of 2ms. The average time taken to store a block and inform the Driver with respect to the number of consumer threads are shown in Figure 9. As we can see, although two consumers perform better than one consumer, more than two consumers in this case does not necessarily improve the latency.

4.4 Limitations

While caching task binaries sound simple in theory, they are more complicated to implement in practice. For example, in our implementation, the Driver caches the serialized format on the function and RDD, and use it to test equality. For serializations across batches to match, we prevented fields such as IDs that are unique across objects to be serialized. Even though this change does not affect correctness, it will complicate other components such as logging that use those information.

Furthermore, our caching technique only works for batches with single stages. When there are multiple stages, i.e. when shuffles or aggregations are involved, the lineage information of the RDD tends to change from batch to batch, making it difficult to hit a previously broadcasted version.

Finally, the largest limitation to our findings is arguably the fact that we only benchmarked the before and after results on a single machine. This limitation is both due to time constraints as well as volatility of the distributed environment. For example, since the clocks are synchronized across machines, the differences between timestamps generated by different machines are less useful, especially when they are in the order of milliseconds.

5 Lessons and Discussion

Having conducted various benchmarks of Spark Streaming for this research, we have found various parameters that can be used to tune the performance of the system. These parameters include the number of Receivers, the batch window size, the size of input records, and whether the input data is serialized. In this section we will describe in detail about their impacts on the end-to-end latency or throughput. We believe these parameters reflect configuration parameters common to micro-batch streaming systems and thus these lessons are generally applicable.

Number of Receivers In our synthetic benchmark using 20-byte records, a single Receiver is able to take in at most around 30MB/s, or 1.5M records/s, even after we modified the code to immediate drop instead of storing the data. While the throughput does not seem very high, 1.5M records is 2.5 times the 600K records per second per node recorded by the original Spark Streaming paper. This insight tells us that when evaluating throughput of a system, both the volume of data as well as the size of input data should be taken into consideration.

To overcome the bottleneck in data intake, the solution can be as simple as increasing the number of Receivers. It is also possible to obtain the performance gain by setting the number of Receivers larger than the number of physical machines.

Size and Number of Input Records As explained by the previous subsection, the size and number of input records affect the system's performance in terms of throughput. As the size of the records increases, the throughput of the system in volume will also increase linearly to a point, since more data is processed per record. Conversely, as the size of input decreases, the Receiver eventually reaches a bottleneck in the number of records that it can handle (around 1.5M/s).

Size of Batch Interval While the micro-batch approach increases throughput at the expense of latency by coalescing input data before processing, the size of the batch interval does not form a linear relationship with the throughput. In fact, the best latency is obtained when the batch interval is set to slightly larger than the time it takes to computationally process a batch. This difference accounts for overheads in task spawning, scheduling, and noise. In practice, instead of using trial and error, it would be best if the system can dynamically adjust the duration of the batch interval based on inputs. Dynamic batch sizes in addition will have the ability to adjust to sudden changes in data incoming rates.

Input Serialization In Section 3, we evaluated the system using plain text input records. However, in real world applications, applications may choose to serialize their input data for better network latency. Serializing input data means they have to be deserialized during computation, so this decision concerns the trade-off between CPU and network I/O.

In the case of Spark Streaming, if the input data is serialized, it has the option to be directly stored inside the system as a block, bypassing the block interval and logic to convert stored records into blocks. Not surprisingly, this approach increases the rate at which Spark Streaming can intake data significantly. For the same benchmark with 20-byte records, Spark Streaming can sustain a throughput of 80MB/s, or 16M records per second using a single Receiver.

The disadvantage of sending data directly as blocks is that they need to be coalesced at the application, i.e. the application needs to group input records together and send them as a block to Spark Streaming. This number should be relatively large: when Spark Stream receives blocks only containing single records, its throughput dropped to around 375KB/s, or 75K records/s.

Others There are also a number of areas which we did not explore, either due to time or resource constraints. First, likely due to the reason that we only had access to 16 machines, scheduling and network communication were never bottlenecks. Second, as we are targeting very low latency streaming workloads, we did not benchmark in detail the performance of the system running applications that involve shuffles and aggregations, i.e. multiple stages per batch. Finally, we did not look into the implications of low latency in the Spark Streaming architecture when fault tolerance becomes a concern.

6 Related Work

High Throughput Trill [9] is a recent query processor for analytics that uses a tempo-relational model, enabling it to support a wide range of latencies for both online and offline data. It achieves high performance by using a streaming batched-columnar data representation, to provide data locality and reduce data access time. The original Spark Streaming [23] paper also highlights its high throughput, along with other desirable features including fault-tolerance, straggler mitigation, fast recovery, and scalability. Our work builds on top of this belief in high throughput, without removing any of the features in the system.

Adaptive Batch Sizes TelegraphCQ [10] is a system designed for a volatile environment, and can make per-tuple and per-operator routing decisions to balance load. [13] studies the effect of batch sizes and other parameters on the throughput and end-to-end latency of the system, and proposes an algorithm based on Fixed-Point Iteration to automatically adapt batch sizes as the circumstance varies. Although there is currently no such mechanism implemented in Spark Streaming, adaptive batch sizes can be very useful in determining the right value for the best trade-offs.

Faster computation Some past work has focused on reducing time to compute results. Systems like BlinkDB [1] perform computations on samples of the data and return prematurely with error bars, while ideas like online aggregation [15] show current results as the query goes to completion. We see these techniques as complementary to our work, as they can be used to reduce the overall time of streaming computations. Others systems such as Incoop [6] and Slider [5] perform computation incrementally in order to avoid repeating previous computations.

Scheduling Extensive research has been done on how to improve the scheduling of tasks in MapReduce frameworks. Systems like Sparrow [18] focus specifically on scenarios where the number of tasks to be scheduled is very large. Even though we haven't identified the scheduler of Spark Streaming as an immediate bottleneck, we believe that methods like decentralized scheduling can be useful. As tasks become shorter, the scheduler may be forced to schedule more tasks per unit of time.

7 Conclusion and Future Work

This paper presents an analysis of low-latency stream processing in Spark Streaming, a space that we have found the system as well as the underlying micro-batch approach lacking in practice. We analyzed the average time breakdown of data processing, identified the bottlenecks, and implemented a number of optimizations for the system as well as proposed several tuning suggestions for the architecture.

Much of this work is still in progress. We have encountered numerous challenges along the way, including time constraint, unfamiliar code base, and inability to reproduce results across even minor versions of Spark Streaming. Nevertheless, we are encouraged by our discoveries, and we plan to continue working on this problem and evaluating the solutions that we have proposed.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[2] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[3] M. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Kirilov, A. Ananthanarayan, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. D. Nicola, X. Wang, D. Maier, I. Santos, O. Nano, and S. Grell. Microsoft cep server and online behavioral targeting. In *International Conference on Very Large Data Bases (VLDB), Lyon, France*, August 2009.

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

[5] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental sliding window analytics. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 61–72, New York, NY, USA, 2014. ACM.

[6] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[7] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 48–58, Washington, DC, USA, 2011. IEEE Computer Society.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[9] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. The trill incremental analytics engine. Technical Report MSR-TR-2014-54, April 2014.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[11] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, volume 10, page 20, 2010.

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

[14] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[16] L. Hu, K. Schwan, H. Amur, and X. Chen. Elf: Efficient lightweight fast stream processing at scale. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 25–36, Philadelphia, PA, June 2014. USENIX Association.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[19] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[20] 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 *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[21] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):555–568, 2003.

[22] 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 *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[23] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.