Drizzle: Fast and Adaptable Stream Processing at Scale

Shivaram Venkataraman[*]
UC Berkeley

Aurojit Panda
UC Berkeley

Kay Ousterhout
UC Berkeley

Michael Armbrust
Databricks

Ali Ghodsi
Databricks, UC Berkeley

Michael J. Franklin
University of Chicago

Benjamin Recht
UC Berkeley

Ion Stoica
Databricks, UC Berkeley

ABSTRACT

Large scale streaming systems aim to provide high throughput and low latency. They are often used to run mission-critical applications, and must be available 24x7. Thus such systems need to adapt to failures and inherent changes in workloads, with minimal impact on latency and throughput. Unfortunately, existing solutions require operators to choose between achieving low latency during normal operation and incurring minimal impact during adaptation. Continuous operator streaming systems, such as Naiad and Flink, provide low latency during normal execution but incur high overheads during adaptation (e.g., recovery), while micro-batch systems, such as Spark Streaming and FlumeJava, adapt rapidly at the cost of high latency during normal operations.

Our key observation is that while streaming workloads require millisecond-level processing, workload and cluster properties change less frequently. Based on this, we develop Drizzle, a system that decouples the processing interval from the coordination interval used for fault tolerance and adaptability. Our experiments on a 128 node EC2 cluster show that on the Yahoo Streaming Benchmark, Drizzle can achieve end-to-end record processing latencies of less than 100ms and can get 2–3x lower latency than Spark. Drizzle also exhibits better adaptability, and can recover from failures 4x faster than Flink while having up to 13x lower latency during recovery.

[*]Correspondence to: shivaram@cs.berkeley.edu

CCS CONCEPTS

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; *Distributed architectures*;

KEYWORDS

Stream Processing, Reliability, Performance

ACM Reference Format:
Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 16 pages.
https://doi.org/10.1145/3132747.3132750

1 INTRODUCTION

Recent trends [48, 52] in data analytics indicate the widespread adoption of stream processing workloads [3, 59] that require low-latency and high-throughput execution. Examples of such workloads include real time object recognition [68] and internet quality of experience prediction [37]. Systems designed for stream processing [14, 67] often process millions of events per second per machine and aim to provide sub-second processing latencies.

Stream processing systems are deployed to process data 24x7 [41, 50], and therefore in addition to meeting performance requirements, these systems must also be able to handle changes in the cluster, workload or incoming data. This requires that they gracefully react to software, machine or disk failures, that can happen once every few hours in large clusters [27], straggler tasks, which can slow down jobs by 6–8x [4, 6] and varying workload patterns [45, 50], which can result in more than 10x difference in load between peak and non-peak durations. To handle such changes data processing, systems have to *adapt*, i.e, dynamically change nodes on which operators are executed, and update the execution plan while ensuring consistent results. These adaptions occur frequently, and as a result systems need to handle them during normal execution, without sacrificing throughput or latency.

Unfortunately existing solutions for stream processing have either focused on providing low-latency during normal operation or on ensuring that adaptation does not affect latency. Systems like Naiad [48] and Apache Flink [14, 54] use a continuous operator streaming model that provides low latency during normal execution. However recovering from failures (or adapting to changes) in such systems is expensive, as even in cases where a single machine fails, the state for all operators must be reset to the last checkpoint, and computation must resume from that point. In contrast, micro-batch based systems like Spark Streaming [67] and FlumeJava [16] process data in batches using the bulk-synchronous processing (BSP) model. This design makes fault recovery (and other adaptations) more efficient as it is possible to reuse partial results and perform parallel recovery [67]. However such systems typically impose a barrier across all nodes after every batch, resulting in high latency [63].

Furthermore, aspects of *both* solutions are required to meet throughput and latency goals. While continuous operator systems can achieve lower latency by immediately processing an input record, processing too few records at a time does not allow the use of techniques such as vectorizations [10] and query optimization [55] which require batching records. On the other hand, while batching in BSP systems naturally enables the use of these techniques, the coordination required at barriers induces additional processing time. As a result, reducing batch size to lower latency often results in unacceptable overhead.

In this paper we propose an alternate design based on the observation that while streaming workloads require millisecond-level processing latency, workload and cluster properties change at a much slower rate (several seconds or minutes) [27, 45]. As a result, it is possible to *decouple the processing interval from the coordination interval used for fault tolerance, adaptability*. This means that we can process tuples every few milliseconds, while coordinating to respond to workload and cluster changes every few seconds.

We implement this model in Drizzle. Our design makes use of a micro-batch processing model with a centralized scheduler and introduces a number of techniques to improve performance. To avoid the centralized scheduling bottleneck, we introduce *group scheduling* (§3.1), where multiple batches (or a group) are scheduled at once. This decouples the granularity of data processing from scheduling decisions and amortizes the costs of task serialization and launch. One key challenge is in launching tasks before their input dependencies have been computed. We solve this using *pre-scheduling* (§3.2), where we proactively queue tasks to be run on worker machines, and rely on workers to trigger tasks when their input dependencies are met. To achieve better throughput, we utilize query optimization techniques [10] while processing small batches of inputs. In combination these techniques help Drizzle to achieve the latency and throughput requirements while remaining adaptable.

Choosing an appropriate group size is important to ensure that Drizzle achieves the desired properties. To simplify selecting a group size, we implement an automatic group-size tuning mechanism that adjusts the granularity of scheduling given a performance target.

We build Drizzle on Apache Spark and integrate Spark Streaming [67] with Drizzle. Using micro-benchmarks on a 128 node EC2 cluster we show that group scheduling and pre-scheduling are effective at reducing coordination overheads by up to 5.5x compared to Apache Spark. With these improvements we show that on Yahoo's stream processing benchmark [63], Drizzle can achieve end-to-end record processing latencies of less than 100ms and is up to 3.5x faster when compared with Spark. Furthermore, as a result of the optimizations enabled by Drizzle, we achieve up to 3x better latency compared to Flink when throughput is held constant, and between 2-3x better throughput at fixed latency. Finally, in terms of fault tolerance, our experiments show that Drizzle recovers around 4x faster from failures than Flink while having up to 13x lower latency during recovery.

2 BACKGROUND

We begin by providing some background about properties that are required for large scale stream processing, following which we describe and compare two computation models used by existing stream processing systems.

2.1 Desirable Properties of Streaming

To illustrate the requirements of production stream jobs, we briefly discuss the design of a prediction service at a video analytics company.

Case Study: Video Quality Prediction. The prediction service is run as a streaming application that computes a number of aggregates based on heartbeats from clients and then queries a machine learning model [37] to determine the optimal parameters for bitrate, CDN location, etc. The output from the streaming job needs to be computed approximately every 100ms to reflect the most recent state of the network to the video frontends. Further the number of heartbeats can range in thousands to millions of updates per second. If the system cannot meet the 100ms deadline (either due to failures or other changes), the use of stale prediction results can lead to a user perceivable degradation in service due to say a wrong choice of bitrate. As a result the prediction service must ensure that it can rapidly recover from failures, and minimize violations of the target latency. Finally, due to the diurnal pattern, the number of viewers and heartbeats

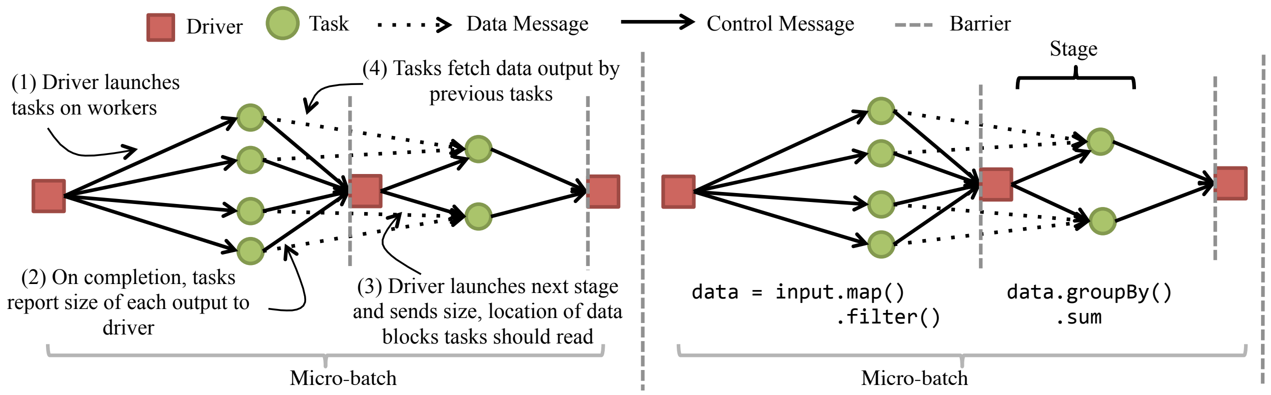

Driver
Task
Data Message
Control Message
Barrier
(1) Driver launches tasks on workers
(4) Tasks fetch data output by previous tasks
(2) On completion, tasks report size of each output to driver
(3) Driver launches next stage and sends size, location of data blocks tasks should read
Micro-batch
Stage
data = input.map()
.filter()
data.groupBy()
.sum
Micro-batch

Figure 1: Execution of a streaming job when using the batch processing model. We show two micro-batches of execution here. The left-hand side shows the various steps used to coordinate execution. The query being executed in shown on the right hand side

varies over the course of a day with specific events (e.g., Superbowl) leading to large increase in load. Thus the system also needs to adapt the number of machines used based on demand. Based on this example, we next discuss the main properties [56] required by large scale streaming systems.

High Throughput. There has been a rapid increase in the amount of data being generated and collected for analysis, and high-throughput processing is essential to keeping up with data streams. For example, in a recent blog post [23] LinkedIn reported generating over 1 trillion messages per day using Kafka, similarly Twitter [58] reports that their timeseries databases need to handle 2.8 billion writes per minute. Keeping up with these ingest rates requires using distributed stream processing systems whose throughput matches or exceeds the incoming data rate.

Low Latency. We define the latency of a stream processing system as the time that elapses between receiving a new record and producing output that accounts for this record. For example, in an anomaly detection workload which predicts trends [2] using a 1-second tumbling window, the processing latency is the time taken to process all events in a one second window and produce the necessary output. Processing latency impacts the system's responsiveness and high latency can both limit the applications that can be supported by a system and user perceived responsiveness. As a result, distributed streaming systems need to provide low latency, and should be designed to ensure that latencies remain low even when the system is scaled across machines.

Adaptability. Unlike traditional analytic queries, streaming jobs process live data and are long lived. As a result the system needs to be able to adapt to changes in both workload and cluster properties. For example, recent papers from Twitter [26, 41] observe that failing hardware, load changes or misbehaving user code occur frequently in production clusters. Additionally it is important to ensure that adaptation does not lead to correctness or prolonged latency spikes [46].

Consistency. A central challenge in distributed stream processing is ensuring consistency for results produced over time. Consistency could be defined in terms of application-level semantics or lower-level message delivery semantics. For example if we have two counters, one tracking the number of requests and another tracking the number of responses, it is important to ensure that requests are accounted for before responses. This can be achieved by guaranteeing *prefix integrity*, where every output produced is equivalent to processing a well-specified prefix of the input stream. Additionally, lower-level message delivery semantics like exactly once delivery are useful for programmers implementing fault tolerant streaming applications. In this work we aim to maintain the consistency semantics offered by existing systems and focus on performance and adaptability.

2.2 Computation Models for Streaming

BSP for Streaming Systems. The bulk-synchronous parallel (BSP) model has influenced many data processing frameworks. In this model, the computation consists of a phase whereby all parallel nodes in the system perform some local computation, followed by a blocking *barrier* that enables all nodes to communicate with each other, after which the process repeats itself. The MapReduce [24] paradigm adheres to this model, whereby a *map* phase can do arbitrary local computations, followed by a barrier in the form of an all-to-all shuffle, after which the *reduce* phase can proceed with each reducer reading the output of relevant mappers (often all of them). Systems such as Dryad [34, 64], Spark [66], and FlumeJava [16] extend the MapReduce model to allow combining many phases of map and reduce after each other,

Property	Micro-batch Model	Continuous Operator Model
Latency	seconds	milliseconds
Consistency	exactly-once, prefix integrity [22]	exactly once [13]
Fault Recovery	sync checkpoints, parallel recovery	sync / async checkpoints, replay
Adaptability	at micro-batch boundaries	checkpoint and restart

Table 1: Comparison of the micro-batch and continuous operator models for different properties useful for streaming systems

and also include specialized operators, e.g. filter, sum, group-by, join. Thus, the computation is a directed acyclic graph (DAG) of operators and is partitioned into different *stages* with a barrier between each of them. Within each stage, many map functions can be fused together as shown in Figure 1. Further, many operators (e.g., sum, reduce) can be efficiently implemented [8] by pre-combining data in the map stage and thus reducing the amount of data transferred.

Streaming systems, such as Spark Streaming [67], Google Dataflow [3] with FlumeJava, adopt the aforementioned BSP model. They implement streaming by creating a *micro-batch* of duration T seconds. During the micro-batch, data is collected from the streaming source, processed through the entire DAG of operators and is followed by a barrier that outputs all the data to the streaming sink, e.g. Kafka. Thus, there is a barrier at the end of each micro-batch, as well as within the micro-batch if the DAG consists of multiple stages, e.g. if it has a group-by operator.

In the micro-batch model, the duration T constitutes a lower bound for record processing latency. Unfortunately, T cannot be set to adequately small values due to how barriers are implemented in all these systems. Consider a simple job consisting of a map phase followed by a reduce phase (Figure 1). A centralized driver schedules all the map tasks to take turns running on free resources in the cluster. Each map task then outputs records for each reducer based on some partition strategy, such as hashing or sorting. Each task then informs the centralized driver of the allocation of output records to the different reducers. The driver can then schedule the reduce tasks on available cluster resources, and pass this metadata to each reduce task, which then fetches the relevant records from all the different map outputs. Thus, each barrier in a micro-batch requires communicating back and forth with the driver. Hence, setting T too low will result in a substantial communication overhead, whereby the communication with the driver eventually dominates the processing time. In most systems, T is limited to 0.5 seconds or more [63].

The use of barriers greatly simplifies fault-tolerance and scaling in BSP systems. First, the scheduler is notified at the end of each stage, and can reschedule tasks as necessary. This allows the scheduler to change the degree of parallelism provided to the job at the end of each stage, and effectively make use of any additional resources available to the job. Furthermore, fault tolerance in these systems is typically implemented by taking a snapshot when at a barrier. This snapshot can either be physical, *i.e.,* record the output from each task in a stage; or logical, *i.e.,* record the computational dependencies for some data. Task failures can be trivially recovered from using these snapshots since the scheduler can reschedule the task and have it read (or reconstruct) inputs from the previous stage's snapshot. Further since every task in the micro-batch model has deterministic inputs, fault recovery can be accelerated by running tasks from different micro-batches or different operators in parallel [67]. For example, consider a job with one map and one reduce stage in each microbatch as shown in Figure 1. If a machine fails, we might lose some map outputs from each timestep. With parallel recovery, map tasks from different timesteps can be run in parallel during recovery.

Continuous Operator Streaming. An alternate computation model that is used in systems specialized for low latency workloads is the dataflow [38] computation model with long running or *continuous operators*. Dataflow models have been used to build database systems [29], streaming databases [1, 17] and have been extended to support distributed execution in systems like Naiad [48], StreamScope [42] and Flink [54]. In such systems, user programs are similarly converted to a DAG of operators, and each operator is placed on a processor as a long running task. As data is processed, operators update local state and messages are directly transferred from between operators. Barriers are inserted only when required by specific operators. Thus, unlike BSP-based systems, there is no scheduling or communication overhead with a centralized driver. Unlike BSP-based systems, which require a barrier at the end of a micro-batch, continuous operator systems do not impose any such barriers.

To handle machine failures, dataflow systems typically use distributed checkpointing algorithms [18] to create consistent snapshots periodically. The execution model is flexible and can accommodate either asynchronous [13] checkpoints (in systems like Flink) or synchronous checkpoints (in systems like Naiad). Recent work on Falkirk Wheel [33] provides a more detailed description comparing these two approaches and also describes how the amount of state that is checkpointed in timely dataflow can be minimized by checkpointing at the end of a processing epoch. However checkpoint replay during recovery can be more expensive in this model. In both synchronous and asynchronous approaches, whenever a node fails, all the nodes are rolled back to the last consistent checkpoint and records are then replayed from this point. As the continuous operators cannot be easily split into smaller components this precludes parallelizing recovery across timesteps (as in the BSP model) and each continuous operator is recovered serially.

2.3 Comparing Computation Models

Table 1 summarizes the difference between the models. As mentioned, BSP-based systems suffer from poor latency due to scheduling and communication overheads which lower-bound the micro-batch length. If the micro-batch duration T is set lower than that, the system will fall behind and become unstable. Continuous operator systems do not have this disadvantage, as no barrier-related scheduling and communication overhead is necessary.

On the other hand, BSP-based systems can naturally adapt at barrier boundaries to recover from failures or add/remove nodes. Continuous operator systems would have to roll-back to a checkpoint and replay from that point on. Both models can guarantee exactly-one semantics.

Finally the execution model in BSP-systems also makes it easy to achieve high throughput by applying optimized execution techniques [10] within each micro-batch. This is especially advantageous for aggregation-based workloads where combining updates across records in a batch can significantly reduce the amount of data transferred over the network. Supporting such optimizations in continuous operator systems and requires explicit buffering or batching.

3 DESIGN

Next we detail the design of Drizzle. Drizzle builds on existing BSP-based streaming systems, and we begin by showing how the BSP model can be changed to dramatically reduce average scheduling and communication overheads. In designing Drizzle, we chose to extend the BSP model since it allows us to inherit existing support for parallel recovery and optimizations for high-throughput batch processing. We believe one could go in the other direction, that is start with a continuous operator system, modify it to add support for tasks with periodic centralized coordination and get similar benefits.

Our high level approach to removing the overheads in the BSP-based streaming model is to decouple the size of the micro-batch being processed from the interval at which coordination takes place. This decoupling will allow us to reduce the size of a micro-batch to achieve sub-second processing latency, while ensuring that coordination, which helps the system adapt to failures and cluster membership changes, takes place every few seconds. We focus on the two sources of coordination that exists in BSP systems (Figure 1). First we look at the centralized coordination that exists *between* micro-batches and how we can remove this with group scheduling. Following that, we discuss how we can remove the barrier *within* a micro-batch using pre-scheduling.

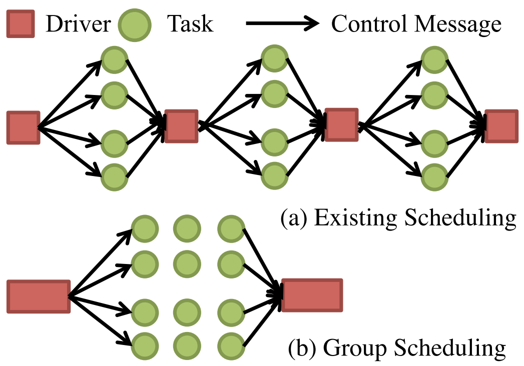


Figure 2: Group scheduling amortizes the scheduling overheads across multiple micro-batches of a streaming job.

3.1 Group Scheduling

BSP frameworks like Spark, FlumeJava [16] or Scope [15] use centralized schedulers that implement many complex scheduling techniques; these include: algorithms to account for locality [65], straggler mitigation [6], fair sharing [35] etc. Scheduling a single stage in these systems proceeds as follows: first, a centralized scheduler computes which worker each task should be assigned to, taking in the account locality and other constraints. Following this tasks are serialized and sent to workers using an RPC. The complexity of the scheduling algorithms used coupled with computational and network limitations of the single machine running the scheduler imposes a fundamental limit on how fast tasks can be scheduled.

The limitation a single centralized scheduler has been identified before in efforts like Sparrow [52], Apollo [11]. However, these systems restrict themselves to cases where scheduled tasks are independent of each other, *e.g.,* Sparrow forwards each scheduling request to one of many distributed schedulers which do not coordinate among themselves and hence cannot account for dependencies between requests. In Drizzle, we focus on micro-batch streaming jobs where there are dependencies between batches and thus coordination is required when scheduling parts of a single streaming job.

To alleviate centralized scheduling overheads, we study the execution DAG of streaming jobs. We observe that in stream processing jobs, the computation DAG used to process micro-batches is largely static, and changes infrequently. Based on this observation, we propose reusing *scheduling decisions* across micro-batches. Reusing scheduling decisions means that we can schedule tasks for *multiple micro-batches* (or a group) at once (Figure 2) and thus amortize the centralized scheduling overheads. To see how this can help, consider a streaming job used to compute moving averages. Assuming the data sources remain the same, the locality preferences computed for every micro-batch will be same. If the cluster configuration also remains static, the same worker to task mapping will be computed for every micro-batch. Thus we can run scheduling algorithms once and reuse its decisions. Similarly, we can reduce network overhead of RPCs by combining tasks from multiple micro-batches into a single message.

When using group scheduling, one needs to be careful in choosing how many micro-batches are scheduled at a time. We discuss how the group size affects the performance and adaptability properties in §3.3 and present techniques for automatically choosing this size in §3.4.

3.2 Pre-Scheduling Shuffles

While the previous section described how we can eliminate the barrier between micro-batches, as described in Section 2 (Figure 1), existing BSP systems also impose a barrier within a micro-batch to coordinate data transfer for shuffle operations. We next discuss how we can eliminate these barriers as well and thus eliminate all coordination within a group.

In a shuffle operation we have a set of upstream tasks (or map tasks) that produce output and a set of downstream tasks (or reduce tasks) that receive the outputs and run the reduction function. In existing BSP systems like Spark or Hadoop, upstream tasks typically write their output to local disk and notify the centralized driver of the allocation of output records to the different reducers. The driver then applies task placement techniques [19] to minimize network overheads and creates downstream tasks that pull data from the upstream tasks. Thus in this case the metadata is communicated through the centralized driver and then following a barrier, the data transfer happens using a pull based mechanism.

To remove this barrier, we *pre-schedule downstream tasks* before the upstream tasks (Figure 3) in Drizzle. We perform scheduling so downstream tasks are launched first; upstream tasks are then scheduled with metadata that tells them which machines running the downstream tasks need to be notified on completion. Thus, data is directly transferred between workers without any centralized coordination. This approach has two benefits. First, it scales better with the number of workers as it avoids centralized metadata management. Second, it removes the barrier, where succeeding stages are launched only when all the tasks in the preceding stage complete.

We implement pre-scheduling by adding a local scheduler on each worker machine that manages pre-scheduled tasks. When pre-scheduled tasks are first launched, these tasks are marked as inactive and do not use any resources. The local scheduler on the worker machine tracks the data dependencies that need to be satisfied. When an upstream task finishes, it materializes the output on local disk, notifies the corresponding downstream workers and asynchronously notifies the centralized scheduler. The local scheduler at the downstream task then updates the list of outstanding dependencies. When all the data dependencies for an inactive task have been met, the local scheduler makes the task active and runs it. When the task is run, it fetches the files materialized by the upstream tasks and continues processing. Thus we implement a push-metadata, pull-based data approach that minimizes the time to trigger tasks while allowing the downstream tasks to control when the data is transferred.

3.3 Adaptability in Drizzle

Group scheduling and shuffle pre-scheduling eliminate barriers both within and across micro-batches and ensure that barriers occur only once every group. However in doing so, we incur overheads when adapting to changes and we discuss how this affects fault tolerance, elasticity and workload changes below. This overhead largely does not affect record processing latency, which continues to happen within a group.

Fault tolerance. Similar to existing BSP systems we create synchronous checkpoints at regular intervals in Drizzle. The checkpoints can be taken at the end of any micro-batch and the end of a group of micro-batches presents one natural boundary. We use heartbeats from the workers to the centralized scheduler to detect machine failures. Upon noticing a failure, the scheduler resubmits tasks that were running on the failed machines. By default these recovery tasks begin execution from the latest checkpoint available. As the computation for each micro-batch is deterministic we further speed up the recovery process with two techniques. First, recovery tasks are executed in parallel [67] across many machines. Second, we also reuse any intermediate data that was created by map stages run in earlier micro-batches. This is implemented with lineage tracking, a feature that is already present in existing BSP systems.

Using pre-scheduling means that there are some additional cases we need to handle during fault recovery in Drizzle. For reduce tasks that are run on a new machine, the centralized scheduler pre-populates the list of data dependencies that have been completed before. This list is maintained based on the asynchronous updates from upstream tasks. Similarly the scheduler also updates the active upstream (map) tasks to send outputs for succeeding micro-batches to the new machines. In both cases, if the tasks encounter a failure in either sending or fetching outputs they forward the failure to the centralized

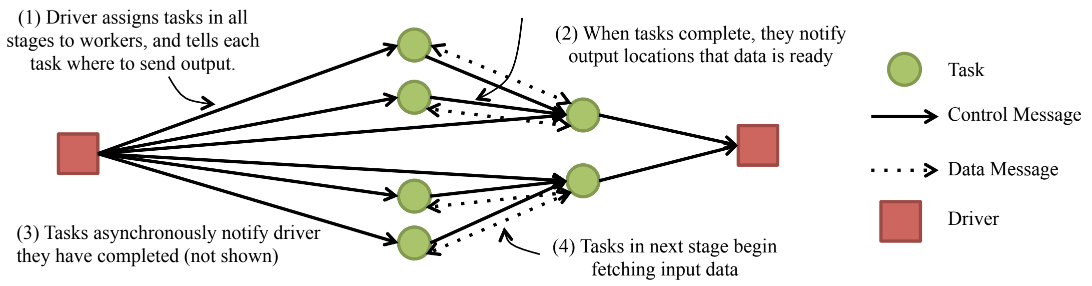

(1) Driver assigns tasks in all stages to workers, and tells each task where to send output.
(2) When tasks complete, they notify output locations that data is ready
(3) Tasks asynchronously notify driver they have completed (not shown)
(4) Tasks in next stage begin fetching input data
Task
Control Message
Data Message
Driver

Figure 3: Using pre-scheduling, execution of a micro-batch that has two stages: the first with 4 tasks; the next with 2 tasks. The driver launches all stages at the beginning (with information about where output data should be sent to) so that executors can exchange data without contacting the driver.

scheduler. Thus we find having a centralized scheduler simplifies design and helps ensure that there is a single source that workers can rely on to make progress.

Elasticity. In addition to handling nodes being removed, we can also handle nodes being added to improve performance. To do this we integrate with existing cluster managers such as YARN [7] and Mesos [32] and the application layer can choose policies [20] on when to request or relinquish resources. At the end of a group boundary, Drizzle updates the list of available resources and adjusts the tasks to be scheduled for the next group. Thus in this case, using a larger group size could lead to larger delays in responding to cluster changes.

3.4 Automatically selecting group size

Intuitively, the group size controls the performance to coordination trade-off in Drizzle. The total runtime of the job can be split between time spent in coordination and time spent in data-processing. In the absence of failures, the job's running time is minimized by avoiding coordination , while more frequent coordination enables better adaptability. These two objectives are thus at odds with each other. In Drizzle, we explore the tradeoff between these objectives by bounding coordination overheads while maximizing adaptability. Thus, we aim to choose a group size that is the smallest possible while having a fixed bound on coordination overheads.

We implement an adaptive group-size tuning algorithm that is inspired by TCP congestion control [12]. During the execution of a group we use counters to track the amount of time spent in various parts of the system. Using these counters we are then able to determine what fraction of the end-to-end execution time was spent in scheduling and other coordination vs. time spent on the workers executing tasks. The ratio of time spent in scheduling to the overall execution gives us the scheduling overhead and we aim to maintain this overhead within user specified lower and upper bounds.

When the overhead goes above the upper bound, we multiplicatively increase the group size to ensure that the overhead decreases rapidly. Once the overhead goes below the lower bound, we additively decrease the group size to improve adaptivity. This is analogous to applying AIMD policy to determine the coordination frequency for a job. AIMD is widely used in TCP, and has been shown to provably converge in general [36]. We use exponentially averaged scheduling overhead measurements when tuning group size. This ensures that we are stable despite transient latency spikes from garbage collection and other similar events.

Aggregate	Percentage of Queries
Count	60.55
First/Last	25.9
Sum/Min/Max	8.640
User Defined Function	0.002
Other	4.908

Table 2: Breakdown of aggregations used in a workload containing over 900,000 SQL and streaming queries.

The adaptive group-size tuning algorithms presents a simple approach that handles the most common scenarios we find in large scale deployments. However the scheme requires users to provide upper and lower bounds for the coordination overhead and these bounds could be hard to determine in a new execution environment. In the future we plan to study techniques that can measure various aspects of the environment to automatically determine the scheduling efficiency.

3.5 Data-plane Optimizations

The previous sections describe the design of the control-plane used in Drizzle, next we describe data plane optimizations enabled by Drizzle. To show the practical importance of batching in the data plane, we start by analyzing the query workload for a popular cloud hosted data analytics provider. We use this analysis to motivate the need for efficient support for aggregations and describe how batching can provide better throughput and latency for such workloads.

Workload analysis. We analyze over 900,000 SQL queries and streaming queries executed on a popular cloud-based data analytics platform. These queries were executed by different users on a variety of data sets. We parsed these queries to determine the set of frequently used operators:

we found that around 25% of queries used one or more aggregation functions. While our dataset did not explicitly distinguish between streaming and ad-hoc SQL queries, we believe that streaming queries which update dashboards naturally require aggregations across time. This platform supports two kinds of aggregation functions: aggregations that can use partial merge operations (e.g., `sum`) where the merge operation can be distributed across workers and complete aggregations (e.g., `median`) which require data to be collected and processed on a single machine. We found that over 95% of aggregation queries only made use of aggregates supporting partial merges. A complete breakdown is shown in Table 2. In summary, our analysis shows that supporting efficient computation of partial aggregates like `count`, `sum` is important for achieving good performance.

Optimization within a batch. In order to support aggregates efficiently, batching the computation of aggregates can provide significant performance benefits. These benefits come from two sources: using vectorized operations on CPUs [10] and by minimizing network traffic from partial merge operations [8]. For example to compute a count of how many ad-clicks were generated for each publisher, we can aggregate the counts per publisher on each machine during a `map` and combine them during the `reduce` phase. We incorporate optimizations within a batch in Drizzle and measure the benefits from this in Section §5.4.

Optimization across batches and queries. Using Drizzle's architecture also enables optimizations across batches. This is typically useful in case the query plan needs to be changed due to changes in the data distribution. To enable such optimizations, during every micro-batch, a number of metrics about the execution are collected. These metrics are aggregated at the end of a group and passed on to a query optimizer [8, 47] to determine if an alternate query plan would perform better. Finally, a micro-batch based architecture also enables reuse of intermediate results *across streaming queries*. This could be useful in scenarios where a number of streaming queries are run on the same dataset and we plan to investigate the benefits from this in the future.

3.6 Discussion

We next discuss other design approaches to group scheduling and extensions to pre-scheduling that can further improve performance.

Other design approaches. An alternative design approach we considered was to treat the existing scheduler in BSP systems as a black-box and pipeline scheduling of one micro-batch with task execution of the previous micro-batch. That is, while the first micro-batch executes, the centralized driver schedules one or more of the succeeding micro-batches. With pipelined scheduling, if the execution time for a micro-batch is t_{exec} and scheduling overhead is t_{sched}, then the overall running time for b micro-batches is $b \times max t_{exec}, t_{sched}$. The baseline would take $b \times t_{exec} + t_{sched}$. We found that this approach is insufficient for larger cluster sizes, where the value of t_{sched} can be greater than t_{exec}.

Another design approach we considered was to model task scheduling as leases [30] that could be revoked if the centralized scheduler wished to make any changes to the execution plan. By adjusting the lease duration we can similarly control the coordination overheads. However using this approach would require reimplementing the task-based execution model used by BSP-style systems and we found that group scheduling offered similar behavior while providing easier integration with existing systems.

Improving Pre-Scheduling. While using pre-scheduling in Drizzle, the reduce tasks usually need to wait for a notification from all upstream map tasks before starting execution. We can reduce the number of inputs a task waits for if we have sufficient semantic information to determine the communication structure for a stage. For example, if we are aggregating information using binary tree reduction, each reduce task only requires the output from two map tasks run in the previous stage. In general inferring the communication structure of the DAG that is going to be generated is a hard problem because of user-defined map and hash partitioning functions. For some high-level operators like `treeReduce` or `broadcast` the DAG structure is predictable. We have implemented support for using the DAG structure for `treeReduce` in Drizzle and plan to investigate other operations in the future.

Quality of Scheduling. Using group and pre-scheduling requires some minor modifications to the scheduling algorithm used by the underlying systems. The main effect this introduces for scheduling quality is that the scheduler's decision algorithm is only executed once for each group of tasks. This coarser scheduling granularity can impact algorithms like fair sharing, but this impact is bounded by the size of a group, which in our experience is limited to a few seconds at most. Further, while using pre-scheduling, the scheduler is unaware of the size of the data produced from the upstream tasks and thus techniques to optimize data transfer [19] cannot be applied. Similarly database-style optimizations that perform dynamic rebalancing [39] of tasks usually depend on data statistics and cannot be used within a group. However for streaming applications we see that the data characteristics change over the course of seconds to minutes and thus we can still apply these techniques using previously collected data.

4 IMPLEMENTATION

We have implemented Drizzle by extending Apache Spark 2.0.0. Our implementation required around 4000 lines of Scala code changes to the Spark scheduler and execution engine. We next describe some of the additional performance improvements we made in our implementation and also describe how we integrated Drizzle with Spark Streaming.

Spark Improvements. The existing Spark scheduler implementation uses two threads: one to compute the stage dependencies and locality preferences, and the other to manage task queuing, serializing, and launching tasks on executors. We observed that when several stages are pre-scheduled together, task serialization and launch is often a bottleneck. In our implementation we separated serializing and launching tasks to a dedicated thread and optionally use multiple threads if there are many stages that can be scheduled in parallel. We also optimized locality lookup for pre-scheduled tasks and these optimizations help reduce the overheads when scheduling many stages in advance. However there are other sources of performance improvements we have not yet implemented in Drizzle. For example, while iterative jobs often share the same closure across iterations we do not amortize the closure serialization across iterations. This requires analysis of the Scala byte code that is part of the closure and we plan to explore this in the future.

Spark Streaming. The Spark Streaming architecture consists of a `JobGenerator` that creates a Spark RDD and a closure that operates on the RDD when processing a micro-batch. Every micro-batch in Spark Streaming is associated with an execution timestamp and therefore each generated RDD has an associated timestamp. In Drizzle, we extend the `JobGenerator` to submit a number of RDDs at the same time, where each generated RDD has its appropriate timestamp. For example, when Drizzle is configured to use group size of 3, and the starting timestamp is t, we would generate RDDs with timestamps t, $t+1$ and $t+2$. One of the challenges in this case lies in how to handle input sources like Kafka [40], HDFS etc. In the existing Spark architecture, the metadata management of which keys or files to process in a micro-batch is done by the centralized driver. To handle this without centralized coordination, we modified the input sources to compute the metadata on the workers as a part of the tasks that read input. Finally, we note these changes are restricted to the Spark Streaming implementation and user applications do not need modification to work with Drizzle.

5 EVALUATION

We next evaluate the performance of Drizzle. First, we use a series of microbenchmarks to measure the scheduling performance of Drizzle and breakdown the time taken at each step in the scheduler. Next we measure the impact of using Drizzle with an industrial streaming benchmark [63]. We compare Drizzle to Apache Spark, a BSP style-system and Apache Flink, a continuous operator stream processing system. Finally, we evaluate the adaptability of Drizzle to a number of scenarios including failures, elastic scaling and also evaluate the efficacy of our group size tuning algorithm.

Our evaluation shows that

- On microbenchmarks, we find that group scheduling and pre-scheduling are effective at reducing coordination overheads by up to 5.5x.
- Using optimizations within a micro-batch, Drizzle is able to achieve less than 100ms latency on the Yahoo Streaming benchmark and is up to 3x faster than Flink and Spark.
- Drizzle recovers from failures around 4x faster than Flink and has up to 13x lower latency during recovery.

5.1 Setup

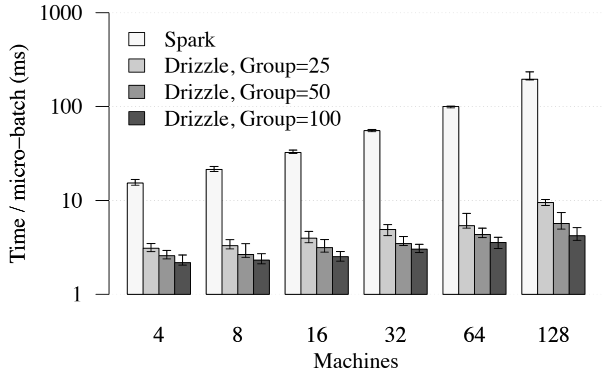
We ran our experiments on 128 `r3.xlarge` instances in Amazon EC2. Each machine has 4 cores, 30.5 GB of memory and 80 GB of SSD storage. We configured Drizzle to use 4 slots for executing tasks on each machine. For all our experiments we warm up the JVM before taking measurements. We use Apache Spark v2.0.0 and Apache Flink v1.1.1 as baselines for our experiments. All three systems are run on the JVM and in our evaluation we use the same JVM heap size and garbage collection flags across all of them.

5.2 Micro benchmarks

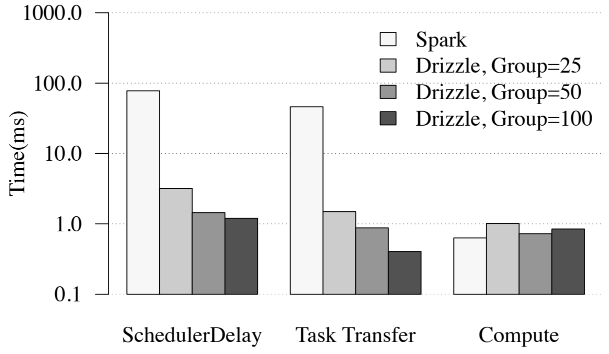
In this section we present micro-benchmarks that evaluate the benefits of group scheduling and pre-scheduling. We run ten trials for each of our micro-benchmarks and report the median, 5th and 95th percentiles.

5.2.1 Group Scheduling. We first evaluate the benefits of group scheduling in Drizzle, and its impact in reducing scheduling overheads with growing cluster size. We use a weak scaling experiment where the amount of computation per task is fixed but the size of the cluster (and hence number of tasks) grow. For this experiment, we set the number of tasks to be equal to the number of cores in the cluster. In an ideal parallel system the total time taken should remain constant. We use a simple workload where each task computes the sum of random numbers and the computation time for each micro-batch is less than 1ms. Note that there are no shuffle operations in this benchmark. We measure the average time taken per micro-batch while running 100 micro-batches and we scale the cluster size from 4–128 machines.

As discussed in §2, we see that (Figure 4(a)) in Spark task scheduling overheads grow as we increase cluster and are around 200ms when using 128 machines. Drizzle is able to

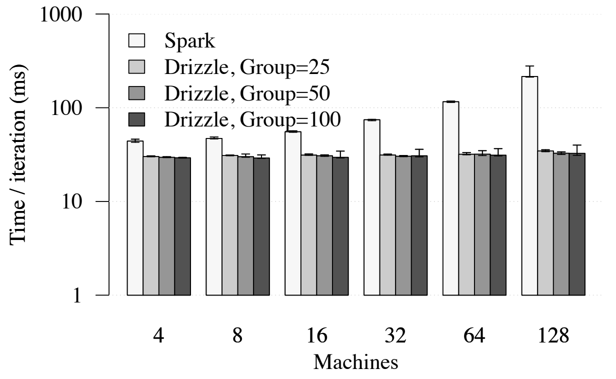


(a) **Time taken for a single stage job with 100 micro-batches while varying the group size. We see that with group size of 100, Drizzle takes less than 5ms per micro-batch.**

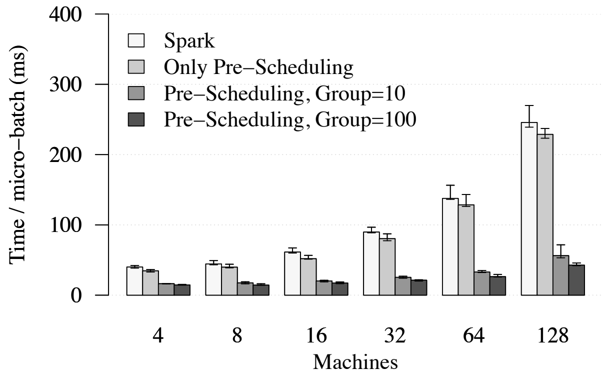


(b) **Breakdown of time taken by a task in the single stage micro-benchmark when using 128 machines. We see that Drizzle can lower the scheduling overheads.**

Figure 4: Micro-benchmarks for performance improvements from group scheduling



(a) **Time taken for a single stage job with** $100x$ **more data while running 100 micro-batches and varying the group size. We see that with group size greater than 25, the benefits diminish.**



(b) **Time taken per micro-batch for a streaming job with shuffles. We break down the gains between pre-scheduling and group scheduling.**

Figure 5: Micro-benchmarks for performance improvements from group scheduling and pre-scheduling

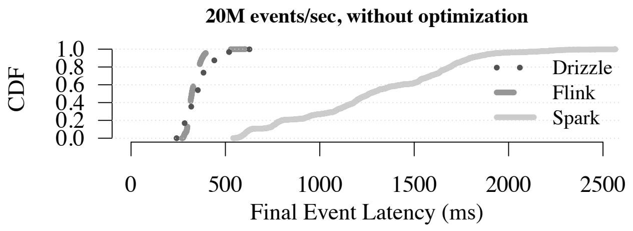
amortize these overheads leading to a $7-46\times$ speedup across cluster sizes. With a group size of 100 and a cluster of 128 machines, Drizzle's scheduler overheads are less than 5ms compared to around 195ms for Spark.

We provide a breakdown of the source of these improvements in Figure 4(b). We do so by analyzing the average time taken by each task for scheduling, task transfer (including serialization, deserialization, and network transfer of the task) and computation. We find that when this benchmark is run on Spark, scheduling and task transfer times dominate, while Drizzle is able to amortize both of these operations using group scheduling.

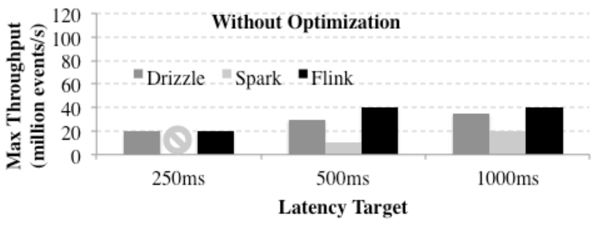
However the benefits for a workload also depends on the amount of computation being performed in each iteration. To measure this effect we increased the amount of data in each partition by 100x and results are shown in Figure 5(a). In this case, using a group size of 25 captures most of the benefits and as the running time is dominated by computation we see no additional benefits from larger group sizes.

5.2.2 Pre-Scheduling Shuffles. To measure benefits from pre-scheduling we use the same setup as in the previous subsection but add a shuffle stage to every micro-batch with 16 reduce tasks. We compare the time taken per micro-batch while running 100 micro-batches in Figure 5(b). Drizzle achieves between 2.7x to 5.5x speedup over Spark as we vary cluster size.

Figure 5(b) also shows the benefits of just using pre-scheduling (i.e., group size = 1). In this case, we still have barriers between the micro-batches and only eliminate barriers within a single micro-batch. We see that the benefits from just pre-scheduling are limited to only 20ms when using 128 machines. However for group scheduling to be effective we need to pre-schedule all of the tasks in the DAG and thus pre-scheduling is essential.



(a) **CDF of event processing latencies when using `groupBy` operations in the micro-batch model. Drizzle matches the latencies of Flink and is around 3.6x faster than Spark.**



(b) **Maximum throughput achievable at a given latency target by Drizzle, Spark and Flink. Spark is unable to sustain latency of 250ms while Drizzle and Flink achieve similar throughput.**

Figure 6: Latency and throughput comparison of Drizzle with Spark and Flink on the Yahoo Streaming benchmark.

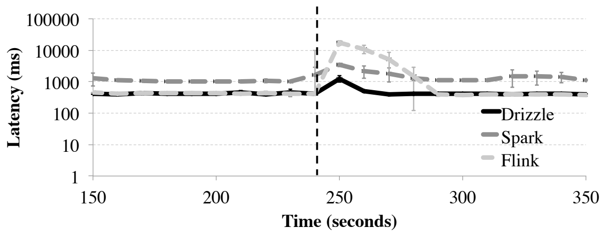


Figure 7: Event Latency for the Yahoo Streaming Benchmark when handling failures. At 240 seconds, we kill one machine in the cluster. Drizzle has lower latency during recovery and recovers faster.

Finally, we see that the time per micro-batch of the two-stage job (45ms for 128 machines) is significantly higher than the time per micro-batch of the one-stage job in the previous section (5 ms). While part of this overhead is from messages used to trigger the reduce tasks, we also observe that the time to fetch and process the shuffle data in the reduce task grows as the number of map tasks increase. To reduce this overhead, we plan to investigate techniques that reduce connection initialization time and improve data serialization/deserialization [44].

5.3 Streaming workloads

We demonstrate Drizzle's benefits for streaming applications using the Yahoo! streaming benchmark [63] which mimics running analytics on a stream of ad-impressions. A producer inserts JSON records into a stream. The benchmark then groups events into 10-second windows per ad-campaign and measures how long it takes for all events in the window to be processed after the window has ended. For example if a window ended at time a and the *last event* from the window was processed at time b, the processing latency for this window is said to be $b - a$.

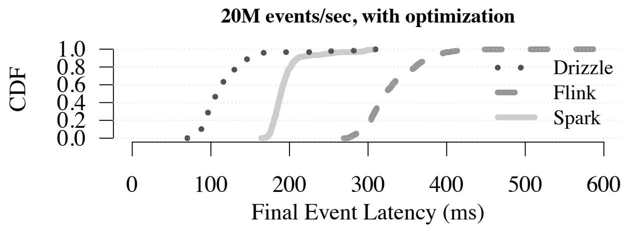
When implemented using the micro-batch model in Spark and Drizzle, this workload consists of two stages per micro-batch: a map-stage that reads the input JSONs and buckets events into a window and a reduce stage that aggregates events in the same window. For the Flink implementation we use the optimized version [21] which creates windows in Flink using the event timestamp that is present in the JSON. Similar to the micro-batch model we have two operators here, a map operator that parses events and a window operator that collects events from the same window and triggers an update every 10 seconds.

For our evaluation, we created an input stream that inserts JSON events and measure the event processing latency. We use the first five minutes to warm up the system and report results from the next five minutes of execution. We tuned each system to minimize latency while meeting throughput requirements. In Spark this required tuning the micro-batch size, while in Flink we tuned the buffer flush duration.

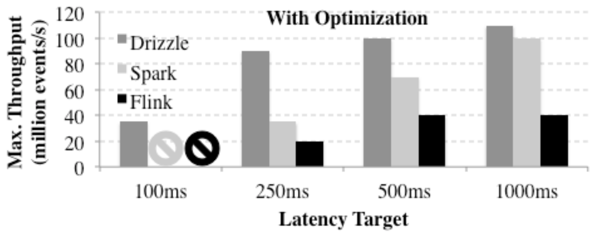
Latency. The CDF of processing latencies for 20M events/second on 128 machines is shown in Figure 6(a). We see Drizzle is able to achieve a median latency of around 350ms and matches the latency achieved by Flink, a continuous operator streaming system. We also verified that the latency we get from Flink match previously reported numbers [21, 63] on the same benchmark. We also see that by reducing scheduling overheads, Drizzle gets around 3.6x lower median latency than Spark.

We next analyze how latency improvements change across workloads. To do this we run a video streaming analytics benchmark which processes heartbeats sent by video streaming clients. The heartbeats are structured as JSON events and contain metadata about the clients, and the type of event being recorded. The heartbeats are grouped together by their session identifier and, and the application uses all the heartbeats for a session, to updates the session summary for each session. This session summary is used by other applications to power dashboards and CDN predictions as described in Section 2.

We show a comparison of Drizzle on both the Yahoo benchmark and the video streaming benchmark in Figure 9. From the figure we see that Drizzle has a similar median latency of around 400ms for this workload but that the 95th percentile latency goes to around 780ms from 480ms. The main reason



(a) **CDF of event processing latencies when using micro-batch optimizations with the Yahoo Streaming Benchmark. Drizzle is 2x faster than Spark and 3x faster than Flink.**



(b) **Maximum throughput achievable at a given latency target by Drizzle, Spark and Flink when using micro-batch optimizations. Spark and Flink fail to meet the 100ms latency target and Drizzle's throughput increases by 2–3x by optimizing execution within a micro-batch.**

Figure 8: Effect of micro-batch optimization in Drizzle in terms of latency and throughput.

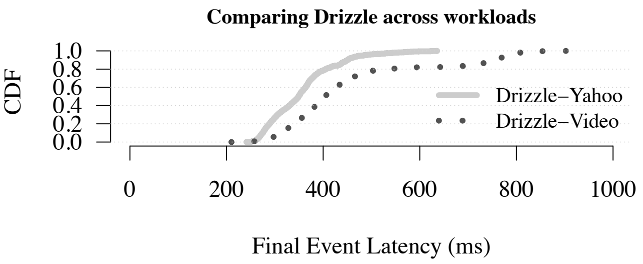


Figure 9: Comparing Drizzle across the Yahoo Streaming Benchmark and a video streaming workload.

for this is that the heartbeats used in video analytics are bigger in size when compared to the ad-campaign events in the Yahoo benchmark and hence the workload involves more data being shuffled. Further the workload also has some inherent skew where some sessions have more events when compared to others. This workload skew results in the 95th percentile latency being much higher than the Yahoo benchmark.

Throughput. We next compare the maximum throughput that can be achieved given a latency target. We use the Yahoo Streaming benchmark for this and for Spark and Drizzle we set the latency target by adjusting the micro-batch size and measure the maximum throughput that can be sustained in a stable fashion. For continuous operator systems like Flink there is no direct way to configure a latency target. Thus, we measure the latency achieved as we increase the throughput and report the maximum throughput achieved within our latency bound. The results from this experiment are shown in Figure 6(b). From the figure we can see that while Spark crashes at very low latency target of 250ms, Drizzle and Flink both get around 20M events/s. At higher latency targets we find that Drizzle gets 1.5-3x more throughput than Spark and that this number reduces as the latency target grows. This is because the overheads from scheduling become less important at higher latency targets and thus the benefits from Drizzle become less relevant.

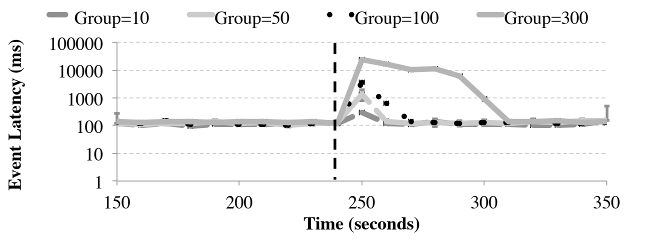
Fault Tolerance. We use the same Yahoo streaming benchmark as above and benchmark the fault tolerance properties of all three systems. In this experiment we kill one machine in the cluster after 240 seconds and measure the event processing latency as before. Figure 7 shows the latency measured for each window across time. We plot the mean and standard deviation from five runs for each system. We see that using the micro-batch model in Spark has good relative adaptivity where the latency during failure is around 3x normal processing latency and that only 1 window is affected. Drizzle has similar behavior where the latency during failure increases from around 350ms to around 1s. Similar to Spark, Drizzle's latency also returns to normal after one window.

On the other hand Flink experiences severe slowdown during failure and the latency spikes to around 18s. Most of this slow down is due to the additional coordination required to stop and restart all the operators in the topology and to restore execution from the latest checkpoint. We also see that having such a huge delay means that the system is unable to catch up with the input stream and that it takes around 40s (or 4 windows) before latency returns to normal range.

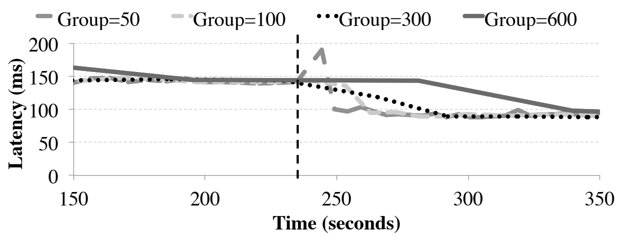
5.4 Micro-batch Optimizations

As discussed in §3.5, one of the advantages of using the micro-batch model for streaming is that it enables additional optimizations to be used *within a batch*. In the case of the Yahoo streaming benchmark, as the output only requires the number of events in a window we can reduce the amount of data shuffled by aggregating counters for each window on the map side. We implemented this optimization in Drizzle and Spark by using the `reduceBy` operator instead of the `groupBy` operator and study the latency, throughput improvements bought about by this change.

Latency. Figure 8(a) shows the CDF of the processing latency when the optimization is enabled. Since Flink creates windows after the keys are partitioned, we could not directly apply the same optimization. In this case we see that using



(a) **Event Latency for the Yahoo Streaming Benchmark when handling failures. At 240 seconds, we kill one machine in the cluster and we see that having a smaller group allows us to react faster to this.**



(b) **Average time taken per micro-batch as we change the number of machines used. At 240 seconds, we increase the number of machines from 64 to 128. Having a smaller group allows us to react faster to this.**

Figure 10: Effect of varying group size in Drizzle.

optimization leads to Drizzle getting around 94ms median latency and is 2x faster than Spark and 3x faster than Flink.

Throughput. Similarly we again measure the maximum throughput that can be achieved given a latency target in Figure 8(b). We see that using optimization within a batch can lead to significant wins in throughput as well. Drizzle is able to sustain around 35M events/second with a 100ms latency target, a target that both Spark and Flink are unable to meet for different reasons: Spark due to scheduling overheads and Flink due to lack of batch optimizations. Similar to the previous comparison we see that the benefits from Drizzle become less pronounced at larger latency targets and that given a 1s target both Spark and Drizzle achieve similar throughput of 100M events/second. We use the optimized version for Drizzle and Spark in the following sections of the evaluation.

In summary, we find that by combining the batch-oriented data processing with the coarse grained scheduling in Drizzle we are able to achieve better performance than existing BSP systems like Spark and continuous operator systems like Flink. We also see that Drizzle also recovers faster from failures when compared to Flink and maintains low latency during recovery.

5.5 Adaptivity in Drizzle

We next evaluate the importance of group size in Drizzle and specifically how it affects adapativity in terms of fault tolerance and elasticity. Following that we show how our auto-tuning algorithm can find the optimal group size.

5.5.1 Fault tolerance with Group Scheduling. To measure the importance of group size for fault recovery in Drizzle, we use the same Yahoo workload as the previous section and we vary the group size. In this experiment we create checkpoints at the end of every group. We measure processing latency across windows and the median processing latency from five runs is shown in Figure 10(a).

We can see that using a larger group size can lead to higher latencies and longer recovery times. This is primarily because of two reasons. First, since we only create checkpoints at the end of every group having a larger group size means that more records would need to be replayed. Further, when machines fail pre-scheduled tasks need to be updated in order to reflect the new task locations and this process takes longer when there are larger number of tasks. In the future we plan to investigate if checkpoint intervals can be decoupled from group and better treatment of failures in pre-scheduling.

5.5.2 Handling Elasticity. To measure how Drizzle enables elasticity we consider a scenario where we start with the Yahoo Streaming benchmark but only use 64 machines in the cluster. We add 64 machines to the cluster after 4 minutes and measure how long it takes for the system to react and use the new machines. To measure elasticity we monitor the average latency to execute a micro-batch and results from varying the group size are shown in Figure 10(b).

We see that using a larger group size can delay the time taken to adapt to cluster changes. In this case, using a group size of 50 the latency drops from 150ms to 100ms within 10 seconds. On the other hand, using group size of 600 takes 100 seconds to react. Finally, as seen in the figure, elasticity can also lead to some performance degradation when the new machines are first used. This is because some of the input data needs to be moved from machines that were being used to the new machines.

5.5.3 Group Size Tuning. To evaluate our group size tuning algorithm, we use the same Yahoo streaming benchmark but change the micro-batch size. Intuitively the scheduling overheads are inversely proportional to the micro-batch size, as small micro-batches imply there are more tasks to schedule. We run the experiment with the scheduling overhead target of 5% to 10% and start with a group size of 2. The progress of the tuning algorithm is shown in Figure 11 for micro-batch size of 100ms and 250ms.

We see that for a smaller micro-batch the overheads are high initially with the small group size and hence the algorithm increases the group size to 64. Following that as the overhead goes below 5% the group size is additively decreased to 49. In the case of the 250ms micro-batch we see

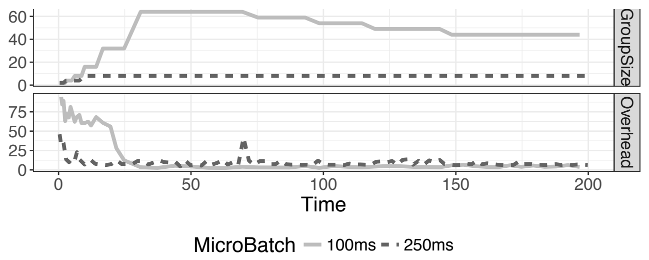


Figure 11: Behavior of group size auto tuning with the Yahoo Streaming benchmark when using two different micro-batch sizes. The optimal group size is lower for the larger micro-batch.

that a group size of 8 is good enough to maintain a low scheduling overhead.

6 RELATED WORK

Stream Processing. Stream processing on a single machine has been studied by database systems including Aurora [1] and TelegraphCQ [17]. As the input event rate for many data streams is high, recent work has looked at large-scale, distributed stream processing frameworks, *e.g.,* Spark Streaming [67], Storm [57], Flink [54], Google Dataflow [3], etc., to scale stream processing across a cluster. In addition to distributed checkpointing, recent work on StreamScope [42] has proposed using reliable vertices and reliable channels for fault tolerance. In Drizzle, we focus on re-using existing fault tolerance semantics from BSP systems and improving performance to achieve low latency.

BSP Performance. Recent work including Nimbus [43] and Thrill [9] has focused on implementing high-performance BSP systems. Both systems claim that the choice of runtime (*i.e.,* JVM) has a major effect on performance, and choose to implement their execution engines in C++. Furthermore, Nimbus similar to our work finds that the scheduler is a bottleneck for iterative jobs and uses scheduling templates. However, during execution Nimbus uses mutable state and focuses on HPC applications while we focus on improving adaptivity by using deterministic micro-batches for streaming jobs in Drizzle. On the other hand Thrill focuses on query optimization in the data plane; our work is therefore orthogonal to Thrill.

Cluster Scheduling. Systems like Borg [61], YARN [7] and Mesos [32] schedule jobs from different frameworks and implement resource sharing policies [28]. Prior work [51] has also identified the benefits of shorter task durations and this has led to the development of distributed job schedulers such as Sparrow [52], Apollo [11], etc. These frameworks assume that the jobs are independent of each other and hence perform distributed scheduling *across* jobs. In Drizzle, we target micro-batch streaming jobs where there are dependencies between batches and thus a single streaming job cannot be easily split across distributed schedulers.

To improve performance within a job, techniques for improving data locality [5, 66], mitigating stragglers [6, 62], re-optimizing queries [39] and accelerating network transfers [19, 31] have been proposed. In Drizzle we study how we can use these techniques for streaming jobs without incurring significant overheads. Prior work [60] has also looked at the benefits of removing the barrier across shuffle stages to improve performance. In addition to removing the barrier, pre-scheduling in Drizzle also helps remove the centralized co-ordination for data transfers. Finally, Ciel [49] proposed a data flow framework that can distribute execution based on data-dependent decisions. In contrast, we exploit the predictable structure of streaming queries in Drizzle to reduce scheduling overheads.

7 CONCLUSION AND FUTURE WORK

In this paper we presented techniques to optimize scheduling and other control plane operations for stream processing workloads, and showed that these techniques enable low latency, high throughput and adaptability. In the future we plan to investigate additional techniques that can be used to improve the performance of the data plane including the use of compute accelerators or specialized network hardware [25] for low latency RPCs. We also plan to extend Drizzle and integrate it with other execution engines (e.g., Impala, Greenplum) and thus develop an architecture for general purpose low latency scheduling.

While we have focused on stream processing in this paper, other workloads including iterative machine learning algorithms [53] can also also benefit from the control plane optimizations in Drizzle. In the future we plan to study how we can support low latency iterative algorithms and how parameter servers can be effectively integrated.

Big data stream processing systems have positioned themselves as either BSP or continuous operators. In Drizzle, we present a new design that decouples the data processing from the fault tolerance, adaptability and show that we can develop a streaming system that combines the best features from both models. This allows Drizzle to achieve high throughput with very low latency not only during normal operation, but also during adaptation.

Acknowledgements: We would like to thank the anonymous reviewers and our shepherd, Marco Serafini, for their insightful comments that improved this paper. We would also like to thank Ganesh Ananthanarayanan, Evan Sparks and Matei Zaharia for their feedback on earlier drafts of this paper. We also thank Jibin Zhan and Yan Li for discussions on video analytics applications. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award

CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware. BR is generously supported by ONR award N00014-17-1-2401, NSF award CCF-1359814, ONR awards N00014-14-1-0024 and N00014-17-1-2191, a Sloan Research Fellowship, a Google Faculty Award, and research grants from Amazon.

REFERENCES

[1] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: a new model and architecture for data stream management. *VLDB* (2003).

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

[3] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., FernÃąndez-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *VLDB* (2015), 1792–1803.

[4] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective straggler mitigation: Attack of the clones. In *NSDI* (2013).

[5] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. Pacman: Coordinated memory caching for parallel jobs. In *NSDI* (2012).

[6] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the outliers in Map-Reduce clusters using Mantri. In *OSDI* (2010).

[7] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[8] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark SQL: Relational data processing in Spark. In *SIGMOD* (2015).

[9] Bingmann, T., Axtmann, M., Jöbstl, E., Lamm, S., Nguyen, H. C., Noe, A., Schlag, S., Stumpp, M., Sturm, T., and Sanders, P. Thrill: High-performance algorithmic distributed batch data processing with c++. *CoRR abs/1608.05634* (2016).

[10] Boncz, P. A., Zukowski, M., and Nes, N. Monetdb/x100: Hyper-pipelining query execution. In *CIDR* (2005), vol. 5, pp. 225–237.

[11] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI* (2014).

[12] Brakmo, L. S., and Peterson, L. L. TCP Vegas: End to End congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications 13*, 8 (Oct. 1995), 1465–1480.

[13] Carbone, P., Fóra, G., Ewen, S., Haridi, S., and Tzoumas, K. Lightweight asynchronous snapshots for distributed dataflows. *CoRR abs/1506.08603* (2015).

[14] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* (2015).

[15] Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: easy and efficient parallel processing of massive data sets. *VLDB* (2008), 1265–1276.

[16] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R., and Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI* (2010).

[17] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. TelegraphCQ: continuous dataflow processing. In *SIGMOD* (2003), ACM.

[18] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3*, 1 (1985), 63–75.

[19] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing data transfers in computer clusters with orchestra. In *SIGCOMM* (2011).

[20] Das, T., Zhong, Y., Stoica, I., and Shenker, S. Adaptive stream processing using dynamic batch sizing. In *SOCC* (2014).

[21] Extending the Yahoo! Streaming Benchmark. http://data-artisans.com/extending-the-yahoo-streaming-benchmark.

[22] Structured Streaming In Apache Spark: A new high-level API for streaming. https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html.

[23] Datanami. Kafka Tops 1 Trillion Messages Per Day at LinkedIn. https://goo.gl/cY7VOz.

[24] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1 (2008).

[25] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP* (2015).

[26] Floratou, A., Agrawal, A., Graham, B., Rao, S., and Ramasamy, K. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment 10*, 12 (2017), 1825–1836.

[27] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. Availability in globally distributed storage systems. In *OSDI* (2010), pp. 61–74.

[28] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI* (2011).

[29] Graefe, G. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD* (1990), pp. 102–111.

[30] Gray, C., and Cheriton, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP* (1989), pp. 202–210.

[31] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., and Crowcroft, J. Queues don't matter when you can jump them! In *NSDI* (2015).

[32] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[33] Isard, M., and Abadi, M. Falkirk wheel: Rollback recovery for dataflow systems. *arXiv preprint arXiv:1503.08877* (2015).

[34] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys* (2007).

[35] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair scheduling for distributed computing clusters. In *SOSP* (2009).

[36] Jacobson, V. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review 18*, 4 (1988), 314–329.

[37] Jiang, J., Sekar, V., Milner, H., Shepherd, D., Stoica, I., and Zhang, H. CFA: A Practical Prediction System for Video QoE Optimization. In *NSDI* (2016), pp. 137–150.

[38] Johnston, W. M., Hanna, J., and Millar, R. J. Advances in

dataflow programming languages. *ACM Computing Surveys (CSUR) 36*, 1 (2004), 1–34.

[39] KE, Q., ISARD, M., AND YU, Y. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Eurosys* (2013), pp. 15–28.

[40] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A distributed messaging system for log processing. In *NetDB* (2011).

[41] KULKARNI, S., BHAGAT, N., FU, M., KEDIGEHALLI, V., KELLOGG, C., MITTAL, S., PATEL, J. M., RAMASAMY, K., AND TANEJA, S. Twitter heron: Stream processing at scale. In *SIGMOD* (2015), pp. 239–250.

[42] LIN, W., QIAN, Z., XU, J., YANG, S., ZHOU, J., AND ZHOU, L. Streamscope: continuous reliable distributed processing of big data streams. In *NSDI* (2016), pp. 439–453.

[43] MASHAYEKHI, O., QU, H., SHAH, C., AND LEVIS, P. Scalable, fast cloud computing with execution templates. *CoRR abs/1606.01972* (2016).

[44] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (2015).

[45] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power management of online data-intensive services. In *ISCA* (2011).

[46] SLA for Stream Analytics. https://azure.microsoft.com/en-us/support/legal/sla/stream-analytics/v1_0/.

[47] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. In *CIDR* (2003).

[48] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *SOSP* (2013), pp. 439–455.

[49] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI* (2011), pp. 113–126.

[50] Stream-processing with Mantis. http://techblog.netflix.com/2016/03/stream-processing-with-mantis.html.

[51] OUSTERHOUT, K., PANDA, A., ROSEN, J., VENKATARAMAN, S., XIN, R., RATNASAMY, S., SHENKER, S., AND STOICA, I. The case for tiny tasks in compute clusters. In *HotOS* (2013).

[52] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *SOSP* (2013), pp. 69–84.

[53] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems* (2011), pp. 693–701.

[54] SCHELTER, S., EWEN, S., TZOUMAS, K., AND MARKL, V. All roads lead to rome: optimistic recovery for distributed iterative data processing. In *CIKM* (2013).

[55] SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. Access path selection in a relational database management system. In *SIGMOD* (1979), pp. 23–34.

[56] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *SIGMOD Record 34*, 4 (Dec. 2005), 42–47.

[57] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., ET AL. Storm at twitter. In *SIGMOD* (2014).

[58] Observability at Twitter: technical overview. https://goo.gl/wAHi2I.

[59] Apache Spark, Preparing for the Next Wave of Reactive Big Data. http://goo.gl/FqEh94.

[60] VERMA, A., CHO, B., ZEA, N., GUPTA, I., AND CAMPBELL, R. H. Breaking the mapreduce stage barrier. *Cluster computing 16*, 1 (2013), 191–206.

[61] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Eurosys* (2015).

[62] YADWADKAR, N. J., ANANTHANARAYANAN, G., AND KATZ, R. Wrangler: Predictable and faster jobs using fewer resources. In *SOCC* (2014).

[63] Benchmarking Streaming Computation Engines at Yahoo! https://yahooeng.tumblr.com/post/135321837876.

[64] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008).

[65] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Eurosys* (2010).

[66] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[67] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP* (2013).

[68] ZHANG, T., CHOWDHERY, A., BAHL, P. V., JAMIESON, K., AND BANERJEE, S. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 426–438.