Supporting Scalable Analytics with Latency Constraints

Boduo Li, Yanlei Diao, Prashant Shenoy
University of Massachusetts Amherst, Massachusetts, USA
{boduo, yanlei, shenoy}@cs.umass.edu

ABSTRACT

Recently there has been a significant interest in building big data analytics systems that can handle both "big data" and "fast data". Our work is strongly motivated by recent real-world use cases that point to the need for a general, unified data processing framework to support analytical queries with different latency requirements. Toward this goal, we start with an analysis of existing big data systems to understand the causes of high latency. We then propose an extended architecture with mini-batches as granularity for computation and shuffling, and augment it with new model-driven resource allocation and runtime scheduling techniques to meet user latency requirements while maximizing throughput. Results from real-world workloads show that our techniques, implemented in Incremental Hadoop, reduce its latency from tens of seconds to sub-second, with 2x-5x increase in throughput. Our system also outperforms state-of-the-art distributed stream systems, Storm and Spark Streaming, by 1-2 orders of magnitude when combining latency and throughput.

1. INTRODUCTION

Recently there has been a significant interest in building big data systems that can handle not only "big data" but also "fast data" for analytics. Here "fast data" refers to high-speed real-time and near real-time data streams, such as twitter feeds, search query streams, click streams, impressions, and system logs. For instance, a breaking news reporting service that monitors the Twitter firehose requires tweet feeds to be analyzed within seconds to detect hot topics and breaking news events [24]. As another example, Google's Zeitgeist pipeline ingests a continuous input of search queries and detects anomalous queries (spiking or dipping) within seconds [2].

To respond to these new analytics needs, our work takes a step further towards building a unified data processing framework that supports both big data, by scaling to multiple machines, and fast data, by taking continuous data streams and answering analytical queries with user-specified latency constraints. In particular, we focus on a fundamental question in this study: What are the key design criteria for building such a system?

A number of systems have been developed to handle fast data, including Google MillWheel [2], Twitter Storm [37], Facebook Ptail and Puma [5], Microsoft Naiad [31] and Sonora [38], WalmartLabs' Muppet [24], IBM System S [41], Yahoo S4 [32], and academic prototypes such as Spark Streaming [39], StreamMapReduce [6], StreamCloud [16], SEEP [8] and TimeStream [35]. Despite various differences in implementations, these systems share some common features: (1) They all employ *data parallelism* to scale processing to a cluster of nodes. Data parallelism partitions a large dataset into smaller subsets, following either the storage order (physical partitioning) or a user-specified logical condition (logical partitioning), and then executes an analytic task in parallel over these subsets. (2) They also support *incremental processing*, a tuple or a small batch at a time, to move tuples quickly through a pipeline of operators. In contrast, batch processing may process all data in the first operator, then move on to process all the data in the next operator, and so on, causing significant delays of the final output.

The key questions that we ask are: (1) *Are data parallelism and incremental processing sufficient for analytical queries with stringent latency requirements?* (2) *If not, what are the additional design features that big and fast data analytics system should embrace?*

A starting point of our study is a thorough understanding of the sources of latency in existing fast data systems. Our benchmark study reveals that while incremental processing allows arriving tuples to be processed one-at-a-time, it does not guarantee the actual latency of processing a tuple in a large distributed system. (1) A key observation is that to enable streaming analytics with bounded latency of processing (e.g., 1 second) through a distributed system, it is crucial to determine the *degree of parallelism* (e.g., the number of processes per node) and *granularity of scheduling* (e.g., batching tuples every 5ms for shuffling). Otherwise, upstream and downstream operators may process data at different speeds, causing substantial data accumulation in between. This reason, as well as using a large batch size as granularity for scheduling, will cause a long wait time before tuples are processed or shuffled. The appropriate choices of those parameters vary widely among analytic tasks due to different computation needs. We refer to this problem as *job-specific resource planning*. (2) When the memory of a cluster is not large enough to process all data, the tuples spilled to disk experience high latency because their processing is often deferred to a later phase (e.g., at the end) of the job. This calls for *latency-aware scheduling* to determine which tuples to process and in what order to process them in order to keep latency low. By way of addressing the above two issues, we make the following contributions in this paper:

Model-driven Resource Planning. Given an analytical query and a user latency constraint, $\mathcal{L}$, our goal is to determine the *degree of parallelism* (the number of processes per node) and *granularity of scheduling* (batching tuples every n ms) for computation and shuffling, in order to meet the latency constraint for final results.

Tremendous engineering efforts have been invested in industry to decide the values of these parameters for critical workloads. The recent development of Hadoop Yarn [23] provides a friendly interface for users to set key system parameters, but cannot do so automatically for a given job. Recent fast data systems [2, 6, 8, 16,

Proceedings of the VLDB Endowment, Vol. 8, No. 11

24, 31, 32, 35, 37, 38, 39] do not take latency or job characteristics as input, and require the user to manually set the system parameters. In practice, enterprise businesses cannot afford the manual work to find the optimal configuration for each job.

To offer best usability, we propose a model-driven approach to automatically determining the resource allocation plan for each job. The first unique aspect of our approach is that we consider performance, including both latency and throughput, in a holistic manner. A naive approach to minimizing latency may overprovision resources, e.g., giving all resources to push one tuple at a time through the distributed system, which limits throughput severely. Instead, we formulate the per-job resource planning problem as a constrained optimization problem: given a user analytic job and latency constraint $\mathcal{L}$, find a resource allocation plan to maximize throughput while subjecting the latency of results to $\mathcal{L}$. The second feature is that we support a variety of latency models, including per-tuple latency, per-window latency, and quantiles associated with latency distributions. Then we subject any of these latency metrics to the latency constraint $\mathcal{L}$ and maximize system throughput.

Latency-Aware Scheduling. As an optimization, we further propose latency-aware scheduling at runtime to determine the set of tuples to process and the order to process them in order to maximize the number of results that meet the latency requirement, i.e., the *total utility*. Such runtime scheduling is helpful because at runtime, the workload characteristics may differ from those provided earlier to our model-driven resource planning, e.g., due to bursty inputs and change of computation costs under constrained memory. Thus, runtime selection and prioritization of tuples greatly affects the overall utility. We propose two runtime scheduling algorithms, at batch-level and tuple-level, respectively, which consider both costs and deadlines of data processing. In particular, our tuple-level scheduling algorithm has provable results on the quality of runtime schedules and efficiency of the scheduling algorithm.

Prototyping and Evaluation. All of our techniques have been implemented in an extension of Incremental Hadoop [26]. Evaluation using real-world workloads such as click stream analysis and tweet analysis show the following results: (1) Our models can capture the trend of actual latency changes when we tune system parameters, with error rates within 15% for the average latency metric, and within 20% for 0.99-quantile of latency. (2) Our model-driven approach to resource planning can reduce the average latency from 10's of seconds in Incremental Hadoop to sub-second, with 2x-5x increase in throughput. (3) For runtime scheduling, our latency-aware tuple scheduling algorithm outperforms D^{over} [22], a state-of-the-art scheduling algorithm with provable optimality in the worse case, and can dramatically improve the number of tuples meeting the latency constraint, especially under constrained memory. (4) We finally compare our system to Twitter Storm [37] and Spark Streaming [39], two state-of-the-art, commercial-grade distributed stream systems. For all workloads tested, our system, implemented as a proof-of-concept in the general Hadoop framework, offers *1-2 orders of magnitude* improvements over Storm and Sparking Streaming, when considering both latency and throughput.

2. SYSTEM DESIGN

In this section, we outline our overall design of a big and fast data analytics system, which provides a technical context for our discussion in the following sections. Since our design is based on Hadoop, which already handles big data processing at scale, we focus on architectural extensions for fast (streaming) data processing.

Background on Incremental Hadoop. Our work is built on an improvement of Hadoop, called Incremental Hadoop [26, 28]. As

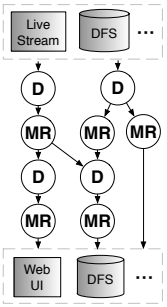


Figure 1: A query modeled as a dataflow DAG.

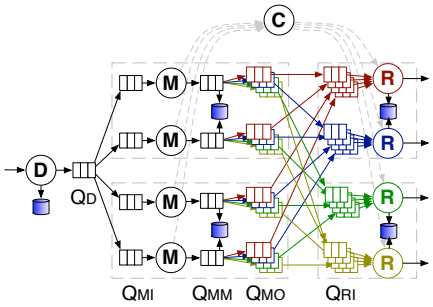


Figure 2: MapReduce architecture extended with distributor & queues of mini batches.

usual, an analytic task can be expressed as a number of rounds of `map` and `reduce` functions. Here incremental processing means that as soon as new data is loaded into the system, it can be processed through the mappers and reducers to produce timely results. Such incremental processing is enabled by a hash framework that (1) uses hashing to group data by key, whenever needed, and (2) applies an incremental algorithm called INC-hash to perform the work defined in the `reduce` function as each tuple arrives at the reducer. To do so, INC-hash maintains ⟨key, state⟩ pairs in memory. When a new tuple arrives, the corresponding state for its key is updated if the state is in memory; otherwise, the tuple is written to disk and will be processed at the end of the job. INC-hash was reported with much improved I/O cost and total running time compared to stock Hadoop and MapReduce Online [11] (also designed for incremental processing). However, this work did not report on the actual latency of each result output from the system.

2.1 Overview of A Scalable Stream System

At a high level, an analytical query in our scalable stream system is modeled as a direct acyclic graph (DAG) of computation units. A computation unit is a pair of map() and reduce(), called an MR-pair. As before, map() is an operation usually used to extract and filter tuples, and reduce() is an operation to perform analytics over a group of tuples of the same key. As for data parallelism, map() can be executed in parallel on subsets of data that are physically partitioned, while reduce() can be executed on subsets that are logically partitioned (by the user-defined key). In an MR-pair, map() can be an empty, or reduce() can be empty, but not both.[1]

More precisely, a query in our system is defined as a dataflow DAG, as shown in Fig. 1. A vertex in the DAG is either a data distributer ("D" vertex) or an MR-pair ("MR" vertex), and an edge represents a stream of tuples flowing between the vertexes. Each tuple is encoded as a triplet ⟨timestamp, key, value⟩. A distributor can take one or multiple streams of tuples from external sources or upstream MR-pairs, as well as files from a distributed file system (DFS). It feeds the received tuples to one or multiple downstream MR-pairs. An MR-pair takes an input stream from a distributor, performs computation over the stream, and outputs a stream to web UI, DFS, or one or multiple distributors.

Incremental Updates. Our system provides a low-level API to program an MR-pair. Like before, a `map` function is applied to transform each input tuple (a triplet here) to a list of output tuples.

$$\texttt{map(time,key}_1\texttt{,val}_1\texttt{)} \rightarrow \texttt{list(time,key}_2\texttt{,val}_2\texttt{)}$$

where all the arguments here indicate data types. For reduce processing, two functions are used:

[1] How to compile a query into a DAG of MR-pairs is within the purview of MapReduce query compilers such as PigLatin [15], while in this paper we focus on system support to run a given query plan with low latency.

```
init(key2) → state
update(time,key2,val2,state) →
              (state,list(time,key3,val3))
```

Reduce is stateful, and a computation state is maintained for each map output key. `init` is called to create an empty state when a new key is received from map output. `update` is triggered by each tuple received, which takes the unique state for a given key and updates it using the tuple. The `update` function can also emit a list of output tuples. For example, to identify frequent words in tweet feeds, `map` can be used to extract a list of ⟨word, count⟩ pairs from each tweet with words as keys. `init` creates a counter as the state for each new word. `update` increments the counter and, if the counter exceeds a predefined threshold, outputs a single pair ⟨word, count⟩. This programming model is similar to those in [6, 24], and has been used in a range of real-world applications.

Time Windows. Our work also provides an API for time window operations. A query defines time windows by specifying the range r and slide s. Given a system starting time t_0, the $\langle r, s\rangle$ pair defines a series of time windows, $(t_0 + i \cdot s, t_0 + r + i \cdot s]$, where $i = 0, 1, \cdots$. These windows can be **tumbling** (non-overlapping) or **sliding** (overlapping) windows. After a query is compiled, those time windows that overlap with the lifetime of the query will trigger actual processing. More specifically, the map API remains unchanged, and the reduce API consists of the following three functions:

```
init(key2,wtime) → state
update(time,key2,wtime,val2,state) → state
finalize(key2,wtime,state) → list(time,key3,val3)
```

Now the system maintains a state for each combination of key and time window. Denote a particular time window by its end time, t_w. The ⟨key, t_w⟩ pair, called a *partitioned window*, is a unique instance of windowed operation. For each partitioned window, `init` is called when a new key is seen in reduce input. `update` is triggered by each arriving tuple, which takes the state of the corresponding partitioned window and updates it with the tuple. Since time windows may overlap in time, a tuple will trigger `update` for all relevant ⟨key, t_w⟩ pairs. Finally, `finalize` is called to complete the computation of a partitioned window and generate output tuples.

Please refer to our technical report [25] for example queries, such as sessionization and windowed aggregates, implemented using our API. These queries are omitted here in the interest of space.

2.2 Extended Hadoop Architecture

We next propose necessary architectural changes of Incremental Hadoop to support stream queries with low latency. We explain these design differences using a single MR-pair as shown in Fig. 2.

Handling stream data using mini-batches and queues. The first set of changes is proposed to break stream input into mini batches and process them with low overhead, including the use of data distributors, queues, and long-living mappers.

We add a distributor (D) that can take streaming input from an external data source or an upstream MR-pair. For an external data source, the distributor tags each input tuple with a system timestamp. The distributor packs the input tuples into a mini batch every B_{in} seconds, as opposed to large batch, to reduce the delay of tuples at the distributor. It places the batch in an in-memory queue, Q_D, such that mappers can fetch data from memory without I/O overhead. The distributor can optionally materialize the mini batches to a DFS for fault tolerance. The distributor can run in multiple processes and nodes to avoid becoming the bottleneck of the system.

We modify a mapper (M) to be able to live forever, rather than terminate after processing a batch. Thus, we can avoid high mapper startup cost caused by each mini batch. We also add several in-memory queues to each mapper. A mapper requests input batches from the distributor, and places the fetched batches in the input queue Q_{MI}. It then applies the map function to each input tuple from Q_{MI}, and emits intermediate tuples, which are further packed into shuffle mini batches every B_{sh} seconds. A shuffle batch is added to the queue Q_{MM}, where it is materialized for fault tolerance, and then split into partitions and placed into the queues Q_{MO} corresponding to different reducers to send. The mapper then informs the coordinator (C) of the availability of each shuffle batch.

On the reducer side, we add an in-memory queue Q_{RI} to a reducer (R). A reducer asks the coordinator for the available shuffle batches every B_{ch} seconds, then fetches the batches, and places them in Q_{RI}. We also add a key-value store (currently implemented using BerkeleyDB) to store key-state pairs when memory is not enough.

Our system also supports out-of-order data and fault tolerance by leveraging existing techniques. Further details are left to [25].

3. RESOURCE PLANNING

We now start to address a key question raised in this study: *Are data parallelism and incremental processing sufficient for analytical queries with stringent latency requirements?* A starting point of our work is a benchmark study [25] that provides a thorough understanding of the sources of latency in Hadoop-based fast data systems. Here, we extract the results that give the most important observations, as shown in Fig. 3.

The workload tested is word counting over tweet feeds where a map function extracts the words as the *key* and emits a *value* (partial count) for each key, where a reduce function computes the total count of each key. To measure latency, we break down the lifetime of a *tuple*, a key-value pair output by the map function, into a number of phases: (1) Map Function (MF) captures the execution cost of map(); (2) Map Materialization (MM) captures the time to materialize map output for fault tolerance; (3) Shuffle Wait (SW) captures the time that map output waits to be shuffled; (4) Shuffle Transfer (ST) is the actual data transfer time; (5) Reduce Wait (RW) is the time that data sits in the reducer input buffer, waiting to be processed; (6) Reduce Update (RU) is the actual cost of reduce(), i.e., updating the computation state with each tuple. Fig. 3 shows these latency measures using both stock Hadoop and Incremental Hadoop [26], in a cluster of 10 nodes with 3 mappers and 3 reducers on each node. While Incremental Hadoop indeed provides better latency than stock Hadoop, it still experiences tens of seconds of latency in Shuffle Wait (SW) and Reduce Wait (RW). The reason is that reducers cannot keep up with mappers in processing, and hence data accumulates between them, in both the map output buffers and reduce input buffers. In addition, each phase of a tuple's lifetime contains at least seconds of latency. This is caused by the large batch size used as granularity for scheduling (32MB in this benchmark, and even larger sizes used in Hadoop such as 64-128MB).

The above observations indicate that to enable streaming analytics with bounded latency of processing (e.g., 1 second) through a distributed system, it is crucial to determine the *degree of parallelism* (e.g., the number of mappers/reducers per node) and *granularity of scheduling* (e.g., batching data items every 5 ms for shuffling). The appropriate choices of those parameters vary widely among analytic tasks due to different computation needs – using the fixed values tuned for one workload is far from ideal for other workloads, often resulting in high latency of tuples moving through the system. We refer to this problem as *job-specific resource planning*.

To offer best usability, in this section we propose a model-driven approach to automatically determining the resource allocation plan for each job. Below, we explain how our approach addresses perfor-

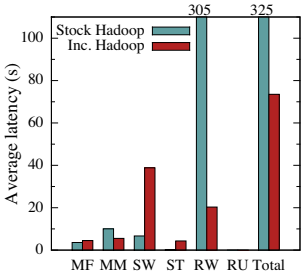


Figure 3: Tuple latency benchmark.

Table 1: System parameters in modeling.

Name (Symbol)	Description
Cluster Size S	The number of slave nodes
Map Parallelism M	The number of mappers per node
Reduce Parallelism R	The number of reducers per node
Input Batch Size B_{in}	The distributer packs input tuples into an input batch every B_{in} sec
Shuffle Batch Size B_{sh}	A mapper packs map output tuples into a shuffle batch every B_{sh} sec
Shuffle Check Period B_{ch}	A reducer checks available shuffle batches from mappers every B_{ch} sec

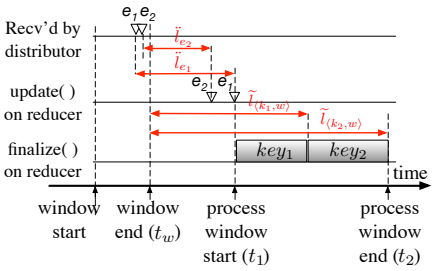


Figure 4: An example of window latencies.

mance (latency and throughput) in a holistic manner and supports a variety of latency models, including per-tuple latency, per-window latency, and any quantiles associated with these latency distributions.

3.1 Model-driven Resource Planning

Given the job of an analytical query, an estimated data input rate λ_0, and a latency constraint $\mathcal{L}$, our goal is to find the optimal resource allocation plan for the job. To do so, we start by considering the relationship among latency, throughput, and resources. A naive approach may minimize latency by giving all resources to push one tuple at a time through the distributed system, which limits throughput severely. Instead, we aim to support both latency and throughput by taking latency as constraint and maximizing throughput under this constraint. To further take resources into account, we consider a cluster of $\bar{S}$ available slave nodes. For each number of slave nodes $S \in \{1, \ldots, \bar{S}\}$, we compute the maximum input rate, $\Lambda_{\mathcal{L}}(S)$, that can be sustained by S nodes under $\mathcal{L}$, as well as the optimal setting of key system parameters, denoted as $\Theta_{\mathcal{L}}(S)$, that reaches the maximum input rate. Then, to configure the system for *"a job with input rate λ_0 and latency constraint $\mathcal{L}$"*, the smallest value of S that satisfies $\Lambda_{\mathcal{L}}(S) \geq \lambda_0$, denoted by S^*, is the minimum number of nodes to use, and $\Theta_{\mathcal{L}}(S^*)$ is the optimal configuration for the S^* nodes. Finally, our approach returns $(S^*, \Theta_{\mathcal{L}}(S^*))$ as the **resource allocation plan** for the analytic job.

In this approach, a key task is to find the maximum input rate that can be sustained (throughput), $\Lambda_{\mathcal{L}}(S)$, and the optimal setting of parameters, $\Theta_{\mathcal{L}}(S)$, for each $S \in \{1, \ldots, \bar{S}\}$. Let λ denote an input rate in number of tuples per second, $\boldsymbol{\theta}$ be a vector of key system parameters, and $\Psi_{\lambda,(S,\boldsymbol{\theta})}$ be the latency[2] under the input rate λ and resource allocation plan $(S, \boldsymbol{\theta})$. Then, we obtain $\Lambda_{\mathcal{L}}(S)$ and $\Theta_{\mathcal{L}}(S)$ by solving a constrained optimization problem for each S as follows:

$$\begin{aligned} \Lambda_{\mathcal{L}}(S) &= \max_{\boldsymbol{\theta}} \quad \lambda, \qquad \textbf{subject to} \quad \Psi_{\lambda,(S,\boldsymbol{\theta})} \leq \mathcal{L}; \\ \Theta_{\mathcal{L}}(S) &= \arg\max_{\boldsymbol{\theta}} \quad \lambda, \qquad \textbf{subject to} \quad \Psi_{\lambda,(S,\boldsymbol{\theta})} \leq \mathcal{L}. \end{aligned} \tag{1}$$

At the core of our approach is the analytical model of $\Psi_{\lambda,(S,\boldsymbol{\theta})}$, which is built on λ and $(S, \boldsymbol{\theta})$, as well as job-specific characteristics and hardware specification of the cluster. Table 1 summarizes all the parameters in $(S, \boldsymbol{\theta})$ for each MR-pair (a computation unit as defined in §2.1). More specifically, θ includes the numbers of mappers and reducers per node, and the mini-batch sizes of various queues placed in our architecture. While the hardware specification of the cluster can be obtained once for all analytical jobs, job-specific characteristics are provided by the programmer or learned at runtime by the system. (We will explain these characteristics more when presenting the detailed models.) Then with the model, $\Psi_{\lambda,(S,\boldsymbol{\theta})}$, Equation 1 can be solved by a general non-linear constrained optimization solver, such as MinConNLP in JMSL[3].

[2]The metric can be the average latency or a quantile of latency.

[3]http://www.roguewave.com/products/imsl-numerical-libraries.aspx

To develop accurate latency models, we identify the major challenges as follows: (1) **Dominant components of latency**: A thorough understanding of the system is required to identify all possible dominant components that contribute to latency. Existing models of MapReduce jobs [17, 18, 19, 14] are designed to predict the running time of a job on stored data. They are not suitable for latency analysis because they ignore key parameters such as the input rate and queue sizes, and factors such as queuing delay and wait time to create a mini-batch, which affect latency strongly. (2) **Shared resources**: Each latency component has to be modeled in a complex environment where system resources are shared by different system modules. Concurrent execution of the map, shuffle and reduce phases are necessary for incremental processing and minimizing latency of output tuples. Existing models [17, 19], however, assume that the map, shuffle and reduce phases do not run in parallel. (3) **Diverse models based on simple statistics**: To offer high usability, it is desirable to support a diverse set of latency metrics, including per-tuple latency, per-window latency, and different quantiles of these latency distributions, while using only the basic job statistics that can be easily provided by the programmer or learned at runtime. None of the existing models can support these latency metrics.

To address these challenges, we choose to model the mean and variance of per-tuple and per-window latency because they enable us to build more complex models for quantiles of latency, while allowing a clean abstraction of various data processing and system-level behaviors using appropriate statistical tools.

3.2 (μ, σ^2) of Per-Tuple Latency

We begin with the incremental updates workload as defined in §2.1. For ease of composition, we first focus on one round of MapReduce computation, i.e., one MR-pair. In incremental updates, a result is output when a map output tuple is processed by the `update` function and triggers the current state to satisfy an output criterion. Therefore, we define tuple latency as follows:

DEFINITION 3.1 (TUPLE LATENCY). *Consider a map output tuple e. Let e' be the (unique) map input tuple that generates e. The latency $\ddot{L}$ of e is the time difference from the distributor receiving e' to the `update` function completing the processing of e.*

We assume that the latencies of all tuples are independent, identically distributed (i.i.d.) random variables. Then the observed latency of each tuple can be viewed as a sample drawn from the same underlying tuple latency distribution, denoted as $f_{\ddot{L}}(l)$. Our goal is to model $E(\ddot{L})$ and $Var(\ddot{L})$.

Latency components: With a detailed analysis of the architecture in Fig. 2, we break $\ddot{L}$ into 12 distinct phases as listed in Table 2. Since these phases run sequentially, we have $\ddot{L} = \sum_{i=1}^{12} \ddot{L}_i$. We further assume that $\ddot{L}_i$ and $\ddot{L}_j$ are independent ($i \neq j$). Then

$$E(\ddot{L}) = \sum_{i=1}^{12} E(\ddot{L}_i), \qquad Var(\ddot{L}) = \sum_{i=1}^{12} Var(\ddot{L}_i).$$

Table 2: Latency breakdown of a map output tuple e (e' is the map input tuple that generates e).

Random Variable	Description	Causes of Latency
$\ddot{L}_1$	From when e' reaches the distributor to when e' is packed into an input mini batch.	**Batching**
$\ddot{L}_2$	Queuing delay seen by input batch at queue Q_D prior to network transfer.	**Queuing**
$\ddot{L}_3$	Network latency to transfer the input batch containing e'.	**Network**
$\ddot{L}_4$	Queuing delay seen by the input batch at queue Q_{MI} prior to `map` processing.	**Queuing**
$\ddot{L}_5$	CPU latency needed by `map` function to process the input batch and generate e.	**CPU**
$\ddot{L}_6$	From when e is generated to when e is packed into a shuffle mini batch.	**Batching**
$\ddot{L}_7$	Queuing delay seen by the shuffle batch at queue Q_{MM} prior to disk write.	**Queuing**
$\ddot{L}_8$	Disk latency to write out the shuffle batch containing e.	**Disk I/O**
$\ddot{L}_9$	From when the shuffle batch is added to queue Q_{MO} to when the network begins transferring the batch.	**Queuing+Heartbeat**
$\ddot{L}_{10}$	Network latency to transfer the shuffle batch containing e.	**Network**
$\ddot{L}_{11}$	Queuing delay seen by the shuffle batch at queue Q_{RI}	**Queuing**
$\ddot{L}_{12}$	CPU latency needed by the `update` function to process the shuffle batch containing e.	**CPU**

Thus, we can model $E(\ddot{L}_i)$ and $Var(\ddot{L}_i)$, the mean and variance of latency in each phase, separately.

More fundamentally, we classify these latency components into six types as shown in the last column in Table 2: (1) CPU, (2) network, (3) disk I/O, (4) queuing, (5) batching tuples, and (6) heartbeat, i.e, waiting a reducer to ask for new map output. We develop a unified approach to modeling latency types (1), (2) and (3), and show the main principles below. Then we briefly introduce the challenge and our solution to model type (4). It is straightforward to model (5) and (6). Thus we leave their details to [25].

CPU, network and disk latencies under shared resources: The latency in this category is determined by the processing time of a batch by the respective resource, i.e. CPU cycles, network bandwidth, or disk bandwidth. $E(\ddot{L}_i)$ in this category can be generally modeled as u/v, where u is the *total* resource required by a batch on average, and v is the resource available to the batch *per second*. (1) **Estimate** u: We estimate u by $m \cdot u_t$, where m is the average number of tuples per batch and u_t is the average resource required per tuple. In general, m can be computed from the data rate and the batch size. Depending on where the batch is in the MR system, the data rate needs to be revised based on the number of mappers or reducers, and input-to-output ratio, α, of the `map` function (in number of tuples) if the batch is downstream of `map`. The statistics required to estimate u, i.e., u_t and α, can be provided by the programmer from historical data or computed by the system from recent batches. (2) **Estimate** v: Due to the nature of incremental processing, the resources on a compute node are shared by many threads on the node (one thread per mapper/reducer). Hence, v is less than the total resources V available on the node. To address the issue of shared resources, we seek to estimate v using a lower bound by assuming that the other threads have higher priority–such a conservative estimate will make our predicted latency an upper bound of the actual latency, which entails still a valid resource allocation plan through constrained optimization. Our approach is to first estimate p, the fraction of the relevant resource required by all other threads on the same node. Then, we model $v = (1-p)V$. For $Var(\ddot{L}_i)$ in these types, we directly model them using the sample variance, which is empirically measured from the test runs of the workload.

CPU Latency. The latency components, $\ddot{L}_5$ and $\ddot{L}_{12}$, model the CPU processing latency of the `map` and `update` functions over a batch of tuples, respectively. Since they model the time from the start of the processing of a batch to when a specific tuple in the batch is processed, and a tuple is in a uniformly random position in the batch, $E(\ddot{L}_5)$ and $E(\ddot{L}_{12})$ can be modeled as *half of the processing time of the batch*. Then we model the average processing time l of a batch using the above approach. Here, u is the number of CPU cycles used to process a batch on average, which can be obtained by testing the average CPU cost per tuple and estimating the size of a batch. C is the number of processing cycles that the CPU has per unit time. If we know p, the fraction of CPU cycles consumed by all other threads on the same node, we can model $l = u/((1-p)\cdot C)$. Further, consider that each CPU has n cores, and a mapper (or reducer) has a single thread to run `map` (or `update`), the fraction of CPU cycles available to the batch is bounded by $1/n$. We then model the average time to process the batch as:

$$E(\ddot{L}_{cpu}) = \frac{u}{2 \cdot \min(1-p, 1/n) \cdot C}. \quad (2)$$

To model p, let c_m (c_r) be the CPU cycles required by a mapper (reducer) per unit time, which can be computed from the number of tuples processed by a mapper (reducer) in unit time. The total CPU cycles required per unit time on a node is

$$c_{total} = M \cdot c_m + R \cdot c_r.$$

When we estimate the running time of a batch in a mapper or reducer, we exclude the cost of the current thread itself from c_{total}, and model p as $(c_{total} - c_m)/C$, or $(c_{total} - c_r)/C$. Plugging p into Equation 2, we obtain the model of $E(\ddot{L}_{cpu})$.

We model *network* and *disk I/O latency* in a similar fashion.

Queuing delays have been well studied in the area of queuing theory. The challenge in our problem is to select an accurate model based on the characteristics that can be easily obtained. After surveying of a wide range of queueing models [33, 20, 21, 4], we decide to model each queue as a G/G/1 queue because the more restrictive models, such as M/M/1, make assumptions that are not true in MR systems, while the more general models, such as queuing network, further complicate our model and may require statistics hard to obtain. Queuing theory of the G/G/1 model [21, 4] states that the mean and variance of the queuing delay can be modeled based on the first, second and third moments of T_a and T_s, where T_a and T_s are the random variables for the inter-arrival time between two consecutive batches and the service time of a batch, respectively. We have modeled $E(T_s)$ for CPU-, network- and disk-type consumers, i.e. $E(\ddot{L}_i)$ in these types. $E(T_a)$ can be computed from the average number of batches added to the queue per unit time. Higher moments of T_a and T_s can be measured empirically.

Finally, the extension to multi-round MapReduce can be achieved by breaking the latency constraint $\mathcal{L}$ of the entire job to a series of constraints $\mathcal{L}_i$ of each MR-pair i, which is described more in [25].

3.3 (μ, σ^2) of Per-Window Latency

Unlike incremental update workloads, the results in the windowed workloads are computed only after a reducer has received **all** the tuples in a time window. Therefore, the per-tuple latency cannot reflect the latency of a windowed result. In this section, we define and model the latency of a windowed result. Recall from §2.1 that a window query defines a series of time windows, $(t_0 + i \cdot s, t_0 + r + i \cdot s]$, where r is the window size, s is the slide, and $i =$

$0, 1, \ldots$ For each time window, tuples can be further partitioned by the key, resulting in a set of *partitioned windows*, each of which produces a unique windowed result (if non-empty).

DEFINITION 3.2 (WINDOW LATENCY). *The latency $\widetilde{L}$ of a partitioned window, denoted as $\langle$key,window$\rangle$, is the time difference from the end-time of the window to the point that the* `finalize` *function completes the processing of $\langle$key,window$\rangle$.*

We assume that latencies of all partitioned windows are i.i.d random variables. Then, the observed latency of each partitioned window can be viewed as a sample from the same window latency distribution, denoted as $f_{\widetilde{L}}(l)$, and our goal is to model $E(\widetilde{L})$ and $Var(\widetilde{L})$. As we shall show shortly, a new major latency component of $\widetilde{L}$ is the *delay of the last tuple of the partitioned window* received by a reducer. The challenge is to choose appropriate tools to model such delay based on easy-to-obtain statistical measurements.

Latency components. Our key observation of the window processing behavior is that once the wall-clock time reaches the end-time t_w of a window, all of its partitioned windows are processed at the same time in two phases: In Phase 1, the system executes the `update` function until all tuples with timestamps earlier than t_w are processed. We denote the completion time of this phase as t_1. Then, in Phase 2, the system executes the `finalize` function over all the related non-empty partitioned windows and completes at t_2.

Fig. 4 shows the t_w, t_1 and t_2 of an example window in a reducer, with input from two mappers. Tuple e_1 is the last tuple of the window received by the reducer from mapper 1, e_2 is the last tuple from mapper 2, and they correspond to the two distinct keys in the window. At time t_1, when both e_1 and e_2 have been processed by `update` (), the reducer executes `finalize`() on the non-empty partitioned windows corresponding to key_1 and key_2 sequentially. The observed latencies of the two partitioned windows are marked as $\widetilde{L}_{\langle key_1,w\rangle}$ and $\widetilde{L}_{\langle key_2,w\rangle}$, both of which contain $(t_1 - t_w)$ and a portion of $(t_2 - t_1)$ determined by the completion time of the corresponding `finalize` execution. Denote the observed tuple latencies of e_1 and e_2 by $\ddot{L}_{e_1}$ and $\ddot{L}_{e_2}$. We estimate $(t_1 - t_w)$ with $\max(\ddot{L}_{e_1}, \ddot{L}_{e_2})$. Then, we can approximate: $\widetilde{L}_{\langle key_1,w\rangle} \approx \max(\ddot{L}_{e_1}, \ddot{L}_{e_2}) + (t_2 - t_1)/2$, and $\widetilde{L}_{\langle key_2,w\rangle} \approx \max(\ddot{L}_{e_1}, \ddot{L}_{e_2}) + (t_2 - t_1)$.

Model. Formally, the quantity $(t_1 - t_w)$ is uncertain, and we model it using a random variable U. Let t' denote the time when the `finalize` function completes the execution of a particular partitioned window ($t_1 \le t' \le t_2$). Since the quantity $(t' - t_1)$ is also uncertain, we model it using a random variable F. It is easy to get $\widetilde{L} = U + F$. Assume that U and F are independent. Then,

$$E(\widetilde{L}) = E(U) + E(F), \quad Var(\widetilde{L}) = Var(U) + Var(F).$$

We only need to model the mean and variance of U and F separately.

Mean and Variance of U. In the above example, $\widetilde{L}_{\langle key_1,w\rangle}$ and $\widetilde{L}_{\langle key_2,w\rangle}$ are two samples drawn from the distribution of $\widetilde{L}$. Based on the insights from the above example, we model

$$U = \max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}}),$$

where e_i is the last tuple of the window received by the reducer from mapper i. The moments of $\max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}})$ are studied in order statistics [12]. We have assumed that $\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}}, \ddot{L}$ are i.i.d. in §3.2. Since we observe the distribution of $\ddot{L}$ to be well centered at its mean in most workloads, we assume that $\ddot{L}$ follows a Gaussian distribution $\mathcal{N}(\ddot{\mu}, \ddot{\sigma}^2)$. Based on Fisher-Tippett-Gnedenko theorem, the following approximation can be applied [12]:

$$E(\max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}})) \approx \ddot{\mu} + a \cdot \ddot{\sigma},$$
$$Var(\max(\ddot{L}_{e_1}, \cdots, \ddot{L}_{e_{S\cdot M}})) \approx \left(a' \cdot \ddot{\sigma}\right)^2,$$

where a and a' are constants relying only in the total number of mappers $(S \cdot M)$. We can see that $\ddot{\mu} = E(\ddot{L})$ and $\ddot{\sigma}^2 = Var(\ddot{L})$ are required, which we have modeled in §3.2.

Mean and Variance of F. Assume that each time window has k non-empty partitioned windows on average, and s_k is the CPU cycles required to process a partitioned window by `finalize`(). k and s_k can be estimated by the average measured empirically. Then, we model $(t_2 - t_1) = (k \cdot s_k)/v$, where v is the CPU resources available to a reducer, which has been modeled in §3.2. As shown in the above example, for a time window, $(t' - t_1)$ of the ith partitioned window processed by `finalize`() can be modeled with $i \cdot (t_2 - t_1)/k$. Now consider a particular key. Assume F for this key in different time windows are i.i.d. We assume that, in each time window, the key has an equal probability to be processed as the 1st, 2nd,$\cdots$,kth partitioned window. Thus, F has an equal probability, $1/k$, to be $i \cdot (t_2 - t_1)/k$ with $i = 1, 2, \cdots, k$. We have $E(F) = (1 + 1/k)(t_2 - t_1)/2$ and $Var(F) = (1 - 1/k^2)(t_2 - t_1)^2/12$.

3.4 Quantiles of Latency

To simplify notation, we use L to generally refer to $\ddot{L}$ if the workload is an incremental update, or $\widetilde{L}$ if it is a windowed workload. We have modeled μ and σ^2 of L in both types of workloads. Now we model any quantile of L, based on μ and σ^2. Let $Q(x)$ denote the x-quantile of L, which has the following property:

$$Pr\left(L \le Q(x)\right) = x. \tag{3}$$

When the distribution of L is known or can be well approximated, we can model $Q(x) = F^{-1}(x)$, where F^{-1} is the inverse CDF of L, and F^{-1} can be expressed using μ and σ^2 in many cases. For example, when the distribution of L can be approximated by a normal distribution, we can model $Q(x) = \mu + \sigma \cdot \Phi^{-1}(x)$, where $\Phi^{-1}(x)$ is the inverse CDF of the standard normal distribution. When the distribution of L is not observed, we use an upper bound of the quantile to model $Q(x)$ in order to provision enough resources to meet the latency constraint. According to Cantelli's inequality,

$$Pr\left(L \le \mu + \sqrt{\frac{x}{1-x}} \cdot \sigma\right) \ge x. \tag{4}$$

According to Eq. 3 and 4, we can have $\mu + \sqrt{\frac{x}{1-x}} \cdot \sigma$ as an upper bound of $Q(x)$. Therefore, we model $Q(x) = \mu + \sqrt{\frac{x}{1-x}} \cdot \sigma$.

Finally, a complete list of job-specific characteristics and hardware specification required in our model is provided in [25].

4. LATENCY-AWARE SCHEDULING

While our model-based approach to resource planning can reduce latency, it may not always offer optimal performance at time. This occurs if the job-specific characteristics previously provided to the model change at runtime, such as the increase in processing cost due to constrained memory and bursty inputs at runtime. In this section, we propose *runtime scheduling* as an optimization, which selects and prioritizes tuples to be processed by the `update` function in each reducer in order to maximize the total utility gained from such processing. We define the scheduling problem as follows:

Problem Statement. Let us represent each map output tuple e by $(t_e, \tilde{t}_e, c_e)$, where t_e is the time when the distributor receives the input tuple that generates e, $\tilde{t}_e$ is the time when the reducer receives

e, and c_e is the time cost of processing e by the `update` function. The problem is to design an online algorithm for each reducer (i.e. the algorithm knows nothing about e before $\tilde{t}_e$) to decide an order to process tuples sequentially in order to **maximize** $\sum U(e, \hat{t}_e)$, where U is a utility function for tuple e and $\hat{t}_e$ is the time when the `update` function completes the processing of e.

We first explain how we obtain the time cost c_e for running `update()` on tuple e. In a reducer, we maintain key-state pairs in an in-memory hash table, where each arriving tuple triggers the update of the computation state for its key. When memory is insufficient to hold all key-state pairs, some of them are staged to a key-value store on local disk (using SSD for better performance). To minimize I/O, we use an existing frequency-based paging algorithm [26] to decide the keys in memory. Then to estimate c_e, we partition tuples into two groups: one group for tuples with keys in memory, and the other for tuples that are staged to disk. Which group a tuple belongs to can be determined by checking the in-memory hash table with low cost. We measure the average per-tuple cost in each group at runtime, and estimate c_e with the average value of the group.

We next present a simple yet popular utility function. Given a user-defined latency constraint $\mathcal{L}$, $U(e, \hat{t}_e) = 1$ if $\hat{t}_e \leq t_e + \mathcal{L}$ (deadline of e); $U(e, \hat{t}_e) = 0$, otherwise. That is, if the latency of a tuple is within $\mathcal{L}$, we gain utility 1. Otherwise, we gain 0.

The problem of maximizing total utility in this context is close to online scheduling problems in real-time operating systems [22, 10], where computation tasks are scheduled to maximize utility. The two major differences in our problem are: (1) a tuple represents a task so the number of tasks to schedule is much larger, and (2) processing a tuple is relatively cheap compared to processing a complex task. Therefore, existing scheduling techniques do not suit our problem due to the high time and space complexity of running the scheduling algorithm for a large number of cheap tuples. To overcome these problems, we propose two scheduling techniques below.

4.1 Batch-level Scheduling

To reduce scheduling complexity, the first method we propose is batch-level scheduling. It considers a shuffle batch received by a reducer as the scheduling unit. When a batch is scheduled, the tuples in the batch are processed sequentially. The total utility of a batch is the number of tuples in the batch n_b. Regarding the cost of the batch, a key assumption we make is that this cost can be approximated by $n_b \cdot \overline{c_e}$, where $\overline{c_e}$ is the average cost per tuple. This assumption is made to simplify scheduling, but is supported by the following intuition: While the costs of processing individual tuples may vary, when we average them over a batch, the tuple-level differences tend to cancel each other and the average cost can be quite stable, especially if we measure the average cost from recently processed batches. Therefore, we obtain two properties for each batch: (1) The *value density* of a batch, which is the utility divided by the cost, is $1/\overline{c_e}$, a constant across batches. (2) When we process a portion of tuples in the batch, we gain utilities of those tuples, even if we do not complete the batch. Given these two properties, the earliest deadline first (EDF) scheduling is known to be optimal [10].

However, batch-level scheduling processes tuples in a batch regardless of their costs, e.g., spending time to process a tuple with a high cost in a scheduled batch rather than two tuples with low costs in an unscheduled batch. Hence, it may not yield optimal utility.

4.2 Tuple-level Cost-aware Scheduling

Next we propose tuple-level cost-aware scheduling, as well as optimizations to reduce its scheduling overhead. Tuple-level scheduling does not satisfy the properties of batch-level scheduling. The value density of a tuple varies due to the varying cost, and we can gain the utility of a tuple only by fully processing it. Hence, EDF is not suitable for tuple-level scheduling. The D^{over} algorithm [22] achieves the optimal worst-case competitive ratio in this setting of online scheduling. However, D^{over} does not consider costs of tuples effectively, and thus can suffer from low utility in practice (verified in §5), despite the worst-case guarantee in theory.

Algorithm 1 Sketch of finding a largest schedulable subset.

```
largestSchedulableSubset(Γ)
1: t ← current time,    Initialize d_i, Γ_i from Γ
2: for i = 1, 2, ··· , g do
3:     while total cost of t + Γ_1 ∪ Γ_2 ∪ ··· ∪ Γ_i > d_i do
4:         remove the tuple of the largest cost from Γ_1 ∪ Γ_2 ∪ ··· ∪ Γ_i
5: return Γ_1 ∪ Γ_2 ∪ ··· ∪ Γ_g
```

We propose a new cost-aware scheduling algorithm. To better describe the algorithm, we first define a concept: A set of tuples Γ is **schedulable** at time t if there exists an order to process all tuples in Γ sequentially starting at time t and no tuple misses its deadline. Now we sketch the algorithm, which includes three functions:

- `init()` is called when a reducer is created.
- `release(`Γ'`)` is called when a shuffle batch is received by the reducer. Here the algorithm maintains a schedulable set of tuples Γ to process, which is initially empty. When a set of tuples Γ' are released to the reducer, the scheduler merges Γ' into Γ, finds a largest schedulable subset of Γ at current time, and updates Γ to the schedule subset.
- `nextToProcess()` returns the next tuple in Γ to process and is invoked by the reducer whenever the processing of the previous tuple completes. Since it is known [13] that processing a schedulable set in increasing order of deadline guarantees that no tuple misses its deadline, our algorithm generates a plan to process tuples in Γ in that order. A tuple is removed from Γ immediately when its processing starts.

Sometimes, before all the tuples in the current schedulable set Γ are processed, the reducer can receive a new shuffle batch. Then the algorithm merges the remaining tuples in Γ with the new tuples Γ', finds a new largest schedulable subset from these tuples, and hands it to the reducer for processing through `nextToProcess()`.

The key part of the algorithm is finding a largest schedulable subset from Γ, which considers both cost and deadline of each tuple. The high-level intuition is that if we enumerate all subsets of Γ, for each subset we can check whether it is a schedulable subset based on the cost and deadline; among those schedulable subsets, we want to find the subset that has the largest number of tuples. Of course, enumerating all subsets is expensive to do. We first outline how we avoid the enumeration in finding a largest schedulable subset, and then introduce an efficient implementation of the scheduling algorithm based on a tree structure. At last, we show the per-tuple scheduling time complexity.

Finding a Largest Schedulable Subset. Let g be the number of distinct deadlines of the tuples in Γ, and $d_1, d_2, \cdots, d_g$ are these distinct deadlines in increasing order. Partitioning Γ based on the deadline gives a tuple set Γ_i for each deadline d_i ($i = 1, \cdots, g$). Algorithm 1 shows the sketch: considering each deadline d_i in increasing order, we remove the tuple with the highest cost in $\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_i$ repeatedly, until all remaining tuples in $\Gamma_1 \cup \cdots \cup \Gamma_i$ can be processed before d_i starting at current time t (Lines 3-4). After all deadlines are checked, the remaining tuples in $\Gamma_1 \cup \cdots \cup \Gamma_g$ are returned (Line 5). The following proposition states the correctness of the algorithm, which is proved in [25].

PROPOSITION 4.1. *$\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_g$ returned by Algorithm 1 is a largest schedulable subset of Γ at time t.*

Table 3: Description and time complexity of basic operations over tree $\mathcal{T}$.

Operation	Description	Complexity
`insert`($\mathcal{T}$,e)	Insert a tuple e to tree $\mathcal{T}$	$O(\log m)$
`delete`($\mathcal{T}$,e)	Delete a tuple e from tree $\mathcal{T}$	$O(\log m)$
`schedulable`($\mathcal{T}$)	Return whether all tuples in $\mathcal{T}$ are schedulable now	$O(1)$
`min_unsched_grp`($\mathcal{T}$)	Return min. d_i s.t. now + total cost of $\Gamma_1 \cup \cdots \cup \Gamma_i > d_i$	$O(\log g)$
`max_cost_in_grps`($\mathcal{T}$,d_i)	Return the tuple with the maximum cost in $\Gamma_1 \cup \cdots \cup \Gamma_i$	$O(\log g)$
`next`($\mathcal{T}$)	Return a tuple with the earliest deadline in $\mathcal{T}$	$O(\log g)$

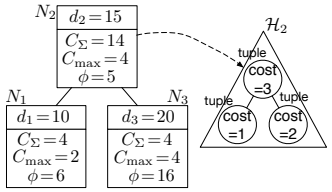


Figure 5: An example of tree $\mathcal{T}$.

A Tree-based Implementation. We now propose an efficient implementation of the tuple-level cost-aware scheduling algorithm using a tree structure $\mathcal{T}$ for organizing tuples in Γ. We implement a few basic operations over $\mathcal{T}$ with corresponding time complexity shown in Table 3, where m is the maximum number of tuples in Γ.

More specifically, $\mathcal{T}$ is a balanced binary search tree of all the distinct deadlines in Γ. Let N_i denote the tree node associated deadline d_i. Each node N_i maintains: 1) a max heap $\mathcal{H}_i$ that organizes all tuples in Γ_i based on tuple cost; 2) metadata that summarizes all tuples in the subtree $\mathcal{T}_i$ rooted at N_i, including:

- c_Σ: total cost of tuples in subtree $\mathcal{T}_i$
- $c_{\max}$: the maximum cost of a tuple in $\mathcal{T}_i$
- ϕ: the latest time to start processing all the tuples in $\mathcal{T}_i$ such that the tuples are schedulable

An example of $\mathcal{T}$ with 3 distinct deadlines 10, 15 and 20 is shown in Fig. 5, where the heap of three tuples with deadline 15 is shown as an example while the other two heaps are omitted.

The tree structure offers two key properties for supporting the operations efficiently. (1) The question, "are the tuples in the subtree $\mathcal{T}_i$ schedulable at time t," can be answered by the boolean expression "$t \le N_i.\phi$" in $O(1)$ time. (2) The metadata in each node N_i can be computed directly from the metadata of its left child N_L, right child N_R, and its associated deadline group Γ_i in $O(1)$ time as follows:

$$N_i.c_\Sigma = N_L.c_\Sigma + N_R.c_\Sigma + \Gamma_i.c_\Sigma$$
$$N_i.c_{\max} = \max(N_L.c_{\max},\ N_R.c_{\max},\ \Gamma_i.c_{\max})$$
$$N_i.\phi = \min(N_L.\phi,\ (d_i - N_L.c_\Sigma - \Gamma_i.c_\Sigma),\ (N_R.\phi - N_L.c_\Sigma - \Gamma_i.c_\Sigma))$$

where $\Gamma_i.c_\Sigma$ is the total tuple cost in Γ_i, and $\Gamma_i.c_{\max}$ is the maximum tuple cost in Γ_i. The calculation of $N_i.c_\Sigma$ and $N_i.c_{\max}$ is straightforward. We explain $N_i.\phi$ more below. Let t be a time when all the tuples in $\mathcal{T}_i$ are schedulable. Processing the tuples in $\mathcal{T}_i$ in increasing order of deadline starting at time t is known to guarantee that no tuple misses its deadline [13]. In such an execution plan, the processing of tuples in the subtree of N_L starts at t, yielding the constraint $t \le N_L.\phi$. The time when the processing of Γ_i ends, which is also when the processing of the tuples in the subtree of N_R starts, is $t + N_L.c_\Sigma + \Gamma_i.c_\Sigma$, giving the two constraints $t + N_L.c_\Sigma + \Gamma_i.c_\Sigma \le d_i$ and $t + N_L.c_\Sigma + \Gamma_i.c_\Sigma \le N_R.\phi$. Combing the three constraints, we have the above equation for $N_i.\phi$. In Fig. 5, the metadata of N_2 can be computed as described above.

Finally, based on the tree operations shown in Table 3, we implement our tuple-level scheduling algorithm as shown in Algorithm 2, where Lines 2-3 of `release()` reimplement Algorithm 1.

Time Complexity. Based on tree $\mathcal{T}$, the operations in Table 3 can be performed in at most $O(\log m)$ time, where m is the maximum number of tuples in Γ as described above. (Note $g \le m$.) We leave the implementation and time complexity analysis of the operations over $\mathcal{T}$ to [25]. As shown in Algorithm 2, each of the basic tree operations is called at most once per tuple. So, the amortized time per tuple of the scheduling algorithm is $O(\log m)$.

4.3 Optimization in Cost-aware Scheduling

Algorithm 2 Implementing the cost-aware tuple-level scheduling algorithm using a tree structure.

```
init()
 1: T ← empty tree
release(Γ')
 1: for each e ∈ Γ' do insert(T,e)
 2: while schedulable(T) ≠ true do
 3:     d ←min_unsched_grp(T),  e ←max_cost_in_grps(T,d),
        delete(T,e),  Abandon e
nextToProcess()
 1: e ← next(T)
 2: if e ≠ null then delete(T,e),  return e
```

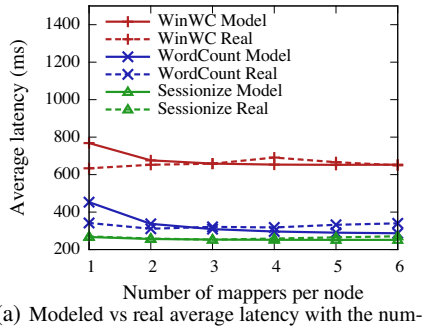
In practice, our tuple-level, cost-aware scheduling may still degrade performance due to the $O(\log m)$ complexity per tuple. We next propose several optimizations of our tuple-level scheduling to further reduce the scheduling cost. (1) We cluster tuples into a fixed number of groups based on similar costs, and estimate the cost of a tuple using the average cost of the group. For example, we described a solution to create a memory group and a disk group earlier in this section. (2) We use a configurable coarse-grained time unit for deadlines to lower the scheduling cost by reducing the size of the tree structure. The size of the time unit controls the trade-off between accuracy of deadlines and efficiency of scheduling. We observe empirically that the most utility is gained when the time unit is about one order of magnitude smaller than the latency constraint. (3) Instead of operating on each individual tuple, a set of tuples from a shuffle batch that share the same cost and deadline can be inserted, deleted, and scheduled for processing together by the scheduler. Due to a small number of cost levels and coarse-grained deadlines, a significant number of tuples can be operated together, and thus the amortized per-tuple scheduling cost is reduced.

We have also extended the runtime scheduling for time windows, the details of which are shown in our technical report [25].

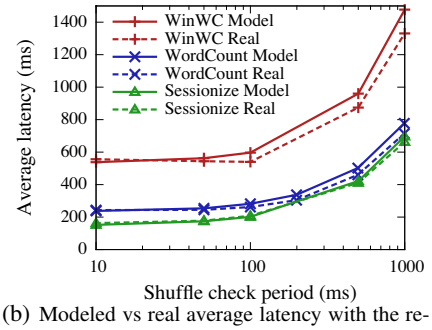
5. PERFORMANCE EVALUATION

We have implemented all of our proposed techniques in a prototype system that extends Incremental Hadoop [26]. In this section, we evaluate the efficiency of our system with the new modeling and scheduling techniques for reducing latency. We also compare to Spark Streaming [39] and Twitter Storm [37], two state-of-the-art distributed stream systems that are widely used in industry.

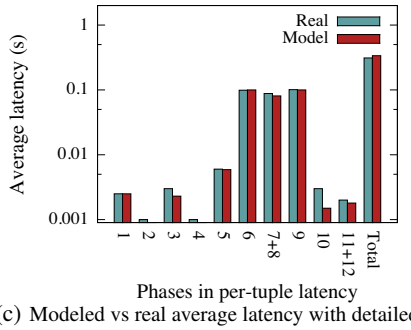
Our evaluation uses a cluster of 10 compute nodes, each equipped with an Intel Xeon X3360 Processor (4 cores), 8GB RAM, and a 2TB Western Digital RE4 HDD. We use three workloads that represent varied complexities of the reduce function (Rf): 1) *Sessionization* over a 236GB WorldCup click stream [26] (Rf: Incremental Update+UDF), for which we consider 0.5-sec latency constraint to simulate stringent performance requirements in click steam analysis; 2) *Word counting* over a 13GB Twitter dataset, which returns words whose counts exceed a threshold (Rf: Incremental Update+Aggregation), for which we consider 1-sec constraint to accommodate the computation needs to extract multiple words from each tweet; 3) *Windowed Word Counting* over the Twitter dataset us-



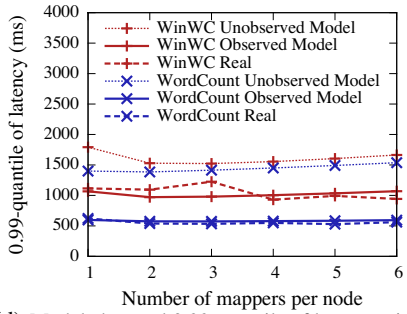
(a) Modeled vs real average latency with the number of mappers per node, M, tuned.



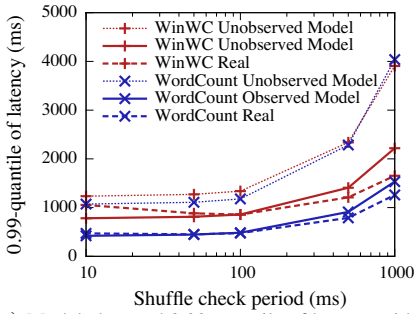
(b) Modeled vs real average latency with the reducer shuffle check period, B_{ch}, tuned.



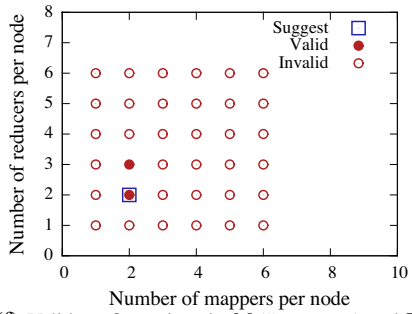
(c) Modeled vs real average latency with detailed breakdown in 12 phases in word counting.



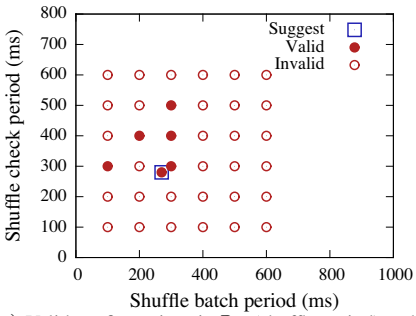
(d) Modeled vs real 0.99-quantile of latency with the number of mappers per node, M, tuned.



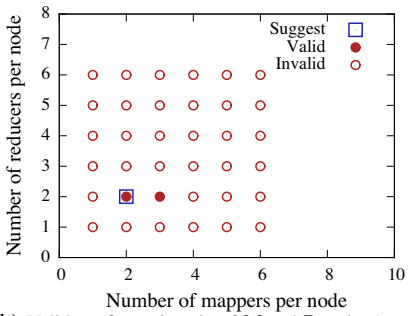
(e) Modeled vs real 0.99-quantile of latency with the reducer shuffle check period, B_{ch}, tuned.



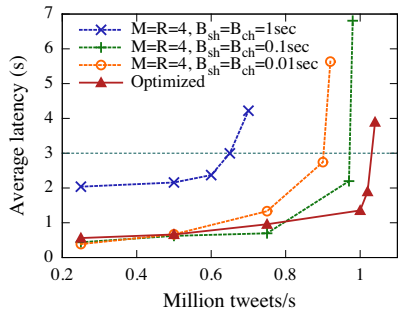
(f) Valid configurations in M (# mappers) and R (# reducers) under 1-sec constraint on avg. latency.



(g) Valid configurations in B_{sh} (shuffle period) and B_{ch} (shuffle check period) under 1-sec avg. latency.



(h) Valid configurations in of M and R under 1-sec constraint on the 0.99-quantile of latency.



(i) Comparing three "rule of thumb" configurations and the optimized one (WinWC, $\mathcal{L}$=3 sec).

Figure 6: Accuracy of our latency models and validity of the model-driven configuration optimization.

ing 30-second tumbling windows (Rf: Time Window+Aggregation), for which we consider 3-sec constraint because it has an additional cost of processing a large number of partitioned windows and needs 1-2 seconds for computation for every window in a reducer.

We simulate a stream data source at a configurable input rate with input mini-batch size B_{in}=5ms. By default, we allocate sufficient buffer space to mappers and reducers, unless stated otherwise. In the evaluation of resource planning, runtime scheduling is not applied so all input data is processed. When evaluating runtime scheduling, the configuration is optimized using the resource planning method.

5.1 Latency-Aware Configuration

We begin by motivating our resource planning with the evaluation of "rule of thumb" configurations, and then evaluate the accuracy of our latency models and their effectiveness for reducing latency.

"Rule of thumb" configurations. We first evaluate our system under a set of "rule of thumb" configurations. For the numbers of mappers and reducers, M and R are fixed to the number of CPU cores per node, which is a widely adopted heuristic in industry; for the scheduling granularities, B_{sh} and B_{ch} are tuned from 10 ms to 1 sec. We measure the average latency in all the three workloads, where the input rate for sessionization is 8 million clicks/sec, and the input rates for word counting and windowed word counting are 1 million tweets/sec. Table 4 shows the results under the "rule of thumb" configurations as well as our optimized configuration, where '-' represents that the system suffers from unbounded latency given the current input rate. It can be seen that (1) latency is sensitive to the configuration of the key parameters, (2) only a few "rule of thumb" configurations can satisfy the latency constraints, and (3) the optimized configuration of each workload can satisfy the constraint.

Model Accuracy. To evaluate accuracy, we compare our modeled latency with the latency measured in our cluster when tuning the key system parameters. The default setting is: $M = R = 2$, and $B_{sh} = B_{ch} = 0.2$ sec. We consider latency metrics ranging from the average latency to the 0.99 quantile of latency.

Regarding *average latency*, Fig. 6(a) shows the results when we tune the number of mappers, M, in three workloads, while Fig. 6(b) shows those when we tune the batch size, B_{ch}. The modeled average latency is close to real values in all the workloads. In 31 out of the 34 experiments, the relative error is below 15%. Results from other tuned parameters show similar results [25]. Note that, increasing M may help reduce latency by utilizing more CPU cores as shown in Fig. 6(a). But beyond a certain point, further increasing M does not help due to the limited number of CPU cores per node. Fig. 6(c)

Table 4: Avg. latency under the "rule of thumb" versus optimized configurations. A bold number means that it meets the latency constraint.

(M, R)	(B_{sh}, B_{ch})	**Sessionize** $\mathcal{L}$=0.5 sec	**WordCount** $\mathcal{L}$=1 sec	**WinWC** $\mathcal{L}$=3 sec
(4, 4)	(10 ms, 10 ms)	-	1.13 sec	-
(4, 4)	(10 ms, 100 ms)	**0.27 sec**	**0.72 sec**	**2.01 sec**
(4, 4)	(10 ms, 1 sec)	0.72 sec	1.07 sec	4.33 sec
(4, 4)	(100 ms, 10 ms)	-	-	-
(4, 4)	(100 ms, 100 ms)	**0.27 sec**	**0.93 sec**	-
(4, 4)	(100 ms, 1 sec)	0.71 sec	1.02 sec	**2.67 sec**
(4, 4)	(1 sec, 10 ms)	0.86 sec	-	-
(4, 4)	(1 sec, 100 ms)	0.77 sec	2.15 sec	6.82 sec
(4, 4)	(1 sec, 1 sec)	1.21 sec	-	-
(4, 2)	(380 ms, 20 ms)	**0.38 sec**		
(2, 2)	(300 ms, 400 ms)		**0.66 sec**	
(2, 2)	(290 ms, 90 ms)			**1.38 sec**

shows a detailed latency breakdown as defined in Table 2 under the default configuration in word counting. In all the phases where the measured latency is over 5ms, the relative error is below 10%.

For the *0.99-quantile of latency*, Fig. 6(d) and Fig. 6(e) show the results with varied M and B_{ch}, for word counting and windowed word counting. For readability, we omit the similar results for sessionization in these plots. To model the 0.99-quantile, we consider the cases that (1) the latency distribution L can be *observed* at runtime, for which we empirically observed it to be well approximated by normal distributions; (2) L is *not observable*, for which we model an upper bound of 0.99-quantile using Cantelli's inequality. In both workloads, the modeled latency has similar trends as the real values. When L is not observed, the modeled latency is an overestimate in all experiments, up to 2.2 times of true latency, due to the use of a data-oblivious upper bound. When L is observable, which is expected to be the common case, the model accuracy is much improved: the relative error is below 20% in 19 out of the 22 experiments.

Model Validity. We next validate the system configuration returned by our model in word counting. We consider 1-second latency constraints on average latency and 0.99-quantile of latency. We feed data at the maximum input rates according to our model, which are 1.2 million tweets/sec for the average latency, and 0.81 million tweets/sec for the 0.99-quantile. We evaluate our system under the model-suggested configuration, as well as other configurations in the system parameter space. We run three times under each configuration, and consider a configuration valid if (1) the latency metric is below 1 sec, and (2) no input tuples are dropped, in all three runs. Fig. 6(f) and 6(g) show the validity of configurations in the 2-dimensional space of M and R, and the space of B_{sh} and B_{ch}, respectively for average latency. We can see that there are only a few configurations valid at the input rate suggested by the model, marked by solid dots. The model-suggested configuration, marked by a square, is among those few valid configurations. Fig. 6(h) shows similar results for 0.99-quantile of latency in the 2-dimensional space of M and R.

Comparison to "rules of thumb". Fig. 6(i) compares three "rule of thumb" configurations and the optimized one on average latency in the windowed word counting workload. When the input rate is low, the optimized configuration provides a latency similar to the lowest latency of the "rule of thumb" configurations. As the input rate increases, the optimized configuration can sustain 5.2% to 57% higher input rate under the 3-second latency constraint.

5.2 Latency-Aware Scheduling

We next evaluate our runtime scheduling algorithms in cases when the job characteristics at runtime differ from or not covered by model-based resource planning. The metric used is the percentage of gained utility (as defined in §4) over the maximum possible utility, i.e. the percentage of tuples that satisfy the latency constraint.

We assumed that sufficient memory cannot be allocated to all processing threads. We now evaluate our scheduling algorithms in §4, by varying the available memory Y in each reducer. Fig. 7(a) shows gained utility under the 1-second latency constraint (L), using the word counting workload and an input of 1.1 million tweets per second. Here, the minimum memory needed to hold all key-state pairs in a reducer is 200MB. As Y reduces below 200MB, without scheduling the number of tuples that satisfy L drops very fast. It is because now some key-state pairs have to be staged to the key-value store on disk. Hence, the per-tuple processing cost increases, which in turn reduces the sustainable input rate to under 1.1 million per second. As tuples queue up in the system, when many of them arrive at the reducer, they have already missed the deadline. Processing them offers no utility and deprives other viable tuples of necessary resources, causing them to miss the deadline as well.

Batch-level scheduling helps by dropping some batches when they arrive at the reducer. However, among those retained batches, those tuples whose key-states are on disk are still processed, postponing other viable cheap-to-process tuples until after the deadline. The cost-aware scheduling offers the best performance, without losing much utility. This is because the tuple-level, cost-aware scheduling gives higher priority to tuples whose key-states pairs are in memory. For skewed key distribution, it further keeps most hot keys in memory and thus most tuples are processed in memory. Fig. 7(b) shows similar trends for the windowed word counting workload.

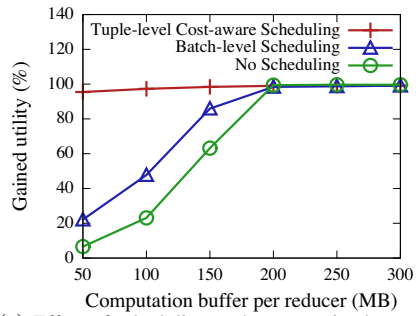
We next consider bursty inputs. For word counting with sufficient memory under 1-second latency constraint, Fig. 7(c) shows the results for (1) normal load of 1.1M tweets/sec, (2) moderate overload of 1.5M tweets/sec, and (3) high overload of 2.0M tweets/sec. In case (2), only a small fraction of tuples can meet the latency requirement without scheduling. Scheduling can drop non-viable tuples and save enough system resources to process the majority of tuples within latency constraint. In case (3), very few tuples can meet the latency constraint without scheduling (which is too low to be displayed in the figure). Scheduling helps increase the system utility, but many tuples are dropped since they have missed deadlines upon arrival at the reducer. Here scheduling works as a load-shedding mechanism, saving significant CPU cycles for non-viable tuples.

Last, we compare our cost-aware scheduling to D^{over} [22], a proven optimal algorithm in the worse case. For a detailed study with controlled per-tuple cost, we run simulation by taking the real input traces to reducers with constrained memory (50MB), fixing the cost of in-memory tuples to a value measured in the real system, but varying the cost of an on-disk tuple. Fig. 7(d) shows the results from the trace of the sessionization workload where the cost per in-memory tuple is $2\mu s$. We can see that when the cost per on-disk tuple is $10\mu s$ to $100\mu s$, our cost-aware scheduling outperforms D^{over} significantly due to the ability to prioritize in-memory tuples. In practice, SSDs are widely used for key-value stores, and a random I/O on a modern SSD takes tens of microseconds.[4] Thus, the cost per on-disk tuple may easily fall in the range between $10\mu s$ and $100\mu s$, where our cost-aware scheduling shows superior performance over D^{over}. Similar observations are made in the word counting and windowed word counting workloads as well.

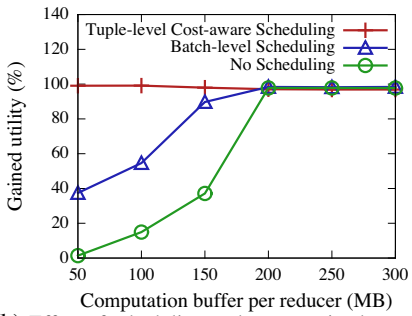
5.3 Comparison to Other Systems

Finally, we compare our system to stock Hadoop, Incremental Hadoop [26], Storm [37] and Spark Streaming [39]. In our system, we take latency as constraint, optimize resource allocation, and show the maximum input rate achieved under the latency constraint. (We do not run runtime scheduling in order to process all the data like the

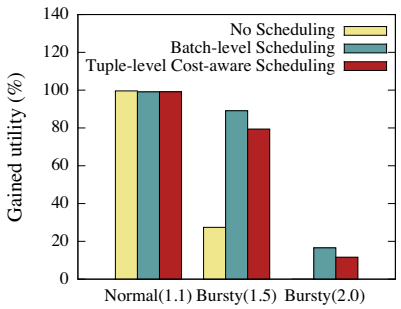
[4]For example, Samsung 850 PRO SSD can achieve 90K to 100K IOPS, which translates to 10 to 11μs per random I/O.



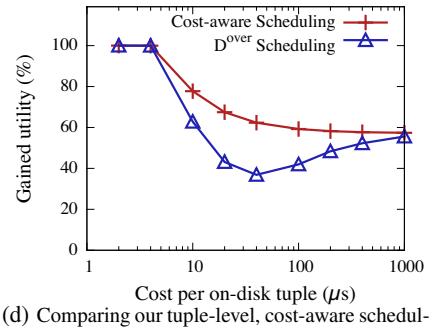
(a) Effect of scheduling under constrained memory, in the word counting workload.



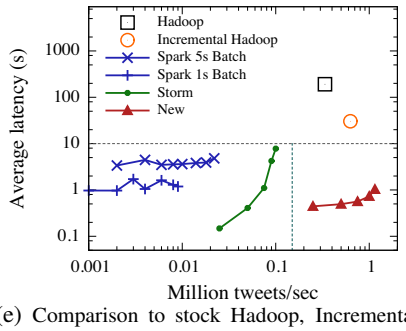
(b) Effect of scheduling under constrained memory, in the windowed word counting workload.



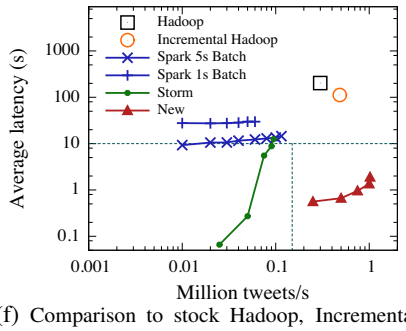
(c) Effect of scheduling under busty inputs.



(d) Comparing our tuple-level, cost-aware scheduling to the D^{over} algorithm.



(e) Comparison to stock Hadoop, Incremental Hadoop, Storm and Spark (word counting).



(f) Comparison to stock Hadoop, Incremental Hadoop, Storm and Spark (windowed counting).

Figure 7: Results of runtime scheduling, and comparison to Hadoop, Incremental Hadoop, Spark Streaming, and Storm in latency and throughput.

other systems.) In other systems, we have to manually try different configurations to explore the tradeoff of latency and throughput.

Fig. 7(e) shows the results of latency and throughput achieved for word counting. We roughly partition the plot into three regions: (1) Hadoop and Incremental Hadoop read data from the distributed file system, and hence have a fixed input rate of around 300,000 and 600,000 tweets/sec, respectively. However, both have high tuple latency of 10's to 100's of seconds, hence not suitable for low-latency tasks. (2) Storm achieves latency of less than 10 seconds when the input rate is within 100,000 tweets/sec, belonging to the low input rate, low latency region. (3) Spark Streaming requires manually setting the batch size (much larger than our mini batches) and recommends 1-5 seconds. Here it achieves stable latency of less than 10 seconds when the input rate is at most 9,000 (20,000) tweets/sec for the 1-sec (5-sec) batch size, also in the low input rate, low latency region. (4) Our system, configured for 1-second latency and even with scheduling turned *off*, can fully process every input tuple with a mean latency under 1sec for input rates up to 1.2 million tweets/sec, putting it in the high input rate, low latency region.

The comparison to Spark Streaming and Storm reveals our advantages of using both mini batches and the resource planning method. Regarding the batch size, Storm performs per-tuple-based scheduling and the high per-tuple scheduling cost prevents Storm from handling higher data inputs.[5] Spark Streaming supports batch size of several seconds, whereas our system can support batches as small as tens of milliseconds efficiently. Further resource planning allows our system to run under optimized configuration including the batch sizes, the effectiveness of which has been evaluated in §5.1. Further analysis of Storm and Spark is given in [25].

For windowed word counting, Spark Streaming stays in the low input rate region and experiences higher latency, as shown in Fig. 7(f). This is because it does not perform incremental processing for a window. Instead, it collects all the data and performs all computation after the window ends, hence delivering windowed results with high latency. Further, Spark Streaming lacks stable support of small batches: here 1-sec batches give worse latency than 5-sec batches.

Summary. Our system can reduce the average latency from 10's of seconds in Incremental Hadoop to sub-second, with 2x-5x increase in throughput. It is able to outperform Storm by 7x-28x in latency and 8-13x in throughput, and outperform Spark Streaming by 4x-27x in latency and 10x-56x in throughput.

6. RELATED WORK

Stream database systems (e.g., [1, 7, 9]) laid a foundation for stream processing including window semantics, optimized implementations, and out-of-order data processing. Our work leverages state-of-the-art techniques for windowed operations and out of order processing in the new context of the MapReduce cluster computing. Techniques for QoS including scheduling and load shedding have been studied in Aurora/Borealis [1, 7, 36]. However, scheduling in Aurora [7] considers latency only based on CPU costs and selectivity of operators in the single-machine environment, whereas our resource planning considers many more factors in a distributed system and uses constrained optimization to support both latency and throughput. Load shedding in Borealis [36] allocates resources in a distributed environment by solving a linear optimization problem, but only maximizes throughput without considering latency and only supports pipeline parallelism without data parallelism.

Distributed stream systems such as System S [41], S4 [32] and Storm [37] adopt a workflow-based programming model and leave many systems issues such as memory management and I/O operations to user code. In contrast, MapReduce systems abstract away these issues in a simple user programming model and automatically handle the memory and I/O issues in the system. CBP [27] supports stateful bulk processing for incremental analytics, but not low-latency streaming analytics (e.g., with running time of 10's to 100's of minutes). Photon [3] is designed particularly for joining data streams in real-time. Naiad [31] is a general parallel platform

[5] We also tried a version of Storm, called Trident, which can process streams in batches, but exhibits worse performance due to various added overheads.

for batch, streaming and iterative workloads, but it adopts per-tuple data transmission similar to Storm with high cost, and does not handle latency constraints. Other systems such as MillWheel [2], Sonora [38], Spark Streaming [39], StreamMapReduce [6], StreamCloud [16], SEEP [8] and TimeStream [35] address a different set of issues such as fault tolerance, load balancing and elasticity. For resource planning, they do not take latency or job characteristics as input, but require manually setting the parameters by the user. We anticipate that our modeling and scheduling techniques, once adapted, can also help reduce latency in these systems.

Cost Models. Recent work presented models to predict total running time of a MapReduce job over stored data based on analysis of CPU, I/O and network costs or based on training over profiles of previous jobs [14, 17, 18, 19, 26]. These models differ from ours as we aim to capture latency, which relates to a different set of system parameters and concerns, such as the input rate, queue sizes, and latency in each step. ParaTimer [30] is a progress indictor for MapReduce jobs, which identifies map and reduce tasks on a query's critical path. It serves a different purpose from our goal of modeling latency. Zeitler et al. [40] proposed a model of total CPU cost for distributed continuous queries, a different goal from ours.

Scheduling. Scheduling techniques to maximize utility have been used in real-time operating systems [13, 22, 10]. However, they are not suitable to our problem due to high time and space complexity. Sparrow [34] is a scheduling system minimizing response time of tasks in a parallel environment by load balancing, different from our focus on per-tuple latency experienced in a sequence of operators.

7. CONCLUSIONS AND FUTURE WORK

Towards building a unified processing framework for big and fast data, we identified the causes of high latency in today's Hadoop systems. To support low-latency, we proposed an extended architecture with mini-batches as granularity for computation and shuffling, and augmented it with new modeling and scheduling techniques to meet user-specified latency requirements while maximizing throughput. Results using real-world workloads show that our techniques, all implemented in a Hadoop-based prototype, can reduce average latency from 10's of seconds in Incremental Hadoop to sub-second, with a 2x-5x increase in throughput. Our scheduling techniques further increase the number of tuples that actually meet the latency constraint. Our system is able to outperform two state-of-the-art distributed stream systems, Storm and Sparking Streaming, by 1-2 orders of magnitude when considering both latency and throughput. In future work, we plan to evaluate our system under a wider range of real-world workloads, and extend it to gracefully handle a mix of query workloads with widely varying latency constraints.

Acknowledgements. The work has been supported in part by the NSF grants DBI-1356486, CNS-0923313, and CNS-1422245, as well as gifts from Google Research, IBM Research, and NEC Labs.

8. REFERENCES

[1] D. J. Abadi, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] T. Akidau, et al. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, pp. 734–746, 2013.

[3] R. Ananthanarayanan, et al. Photon: fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pp. 577–588, 2013.

[4] S. Asmussen. *Applied probability and queues; 2nd ed.* Stochastic Modelling and Applied Probability. Springer, New York, 2003.

[5] D. Borthakur, et al. Apache hadoop goes realtime at facebook. In *SIGMOD*, pp. 1071–1080, 2011.

[6] A. Brito, et al. Scalable and low-latency data processing with stream mapreduce. In *CloudCom*, pp. 48–58, 2011.

[7] D. Carney, et al. Operator scheduling in a data stream manager. In *VLDB*, pp. 333-353, 2003.

[8] R. Castro Fernandez, et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pp. 725–736, 2013.

[9] S. Chandrasekaran, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[10] F. Y. Chin and S. P. Fung. Improved competitive algorithms for online scheduling with partial job values. *Theoretical Computer Science*, 325(3):467–478, 2004.

[11] T. Condie, et al. Mapreduce online. In *NSDI*, pp. 21–21, 2010.

[12] H. David and H. Nagaraja. *Order Statistics*. Wiley, 2004.

[13] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIF Congress*, pp. 807–813, 1974.

[14] A. D. Ferguson, et al. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, pp. 99–112, 2012.

[15] A. Gates, et al. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[16] V. Gulisano, et al. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, 2012.

[17] H. Herodotou. Hadoop performance models. *Technical Report CS-2011-05, Duke Computer Science*, 2011.

[18] H. Herodotou, et al. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, pp. 1111–1122, 2011.

[19] V. Jalaparti, et al. Bridging the tenant-provider gap in cloud services. In *SoCC*, pp. 10:1–10:14, 2012.

[20] B. Jansson. Choosing a good appointment system-a study of queues of the type (d, m, 1). *Operations Research*, 14(2):pp. 292–312, 1966.

[21] J. F. C. Kingman. Some inequalities for the queue gi/g/1. *Biometrika*, 49(3/4):pp. 315–324, 1962.

[22] G. Koren and D. Shasha. Dover; an optimal on-line scheduling algorithm for overloaded real-time systems. In *RTSS*, 1992.

[23] V. Kumar, et al. Apache hadoop yarn: yet another resource negotiator. In *SOCC*, pp. 5:1–5:16, 2013.

[24] W. Lam, et al. Muppet: Mapreduce-style processing of fast data. *PVLDB*, 5(12):1814–1825, 2012.

[25] B. Li, et al. Supporting scalable analytics with latency constraints. Technical Report http://scalla.cs.umass.edu/papers/techreport2015.pdf.

[26] B. Li, et al. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pp. 985–996, 2011.

[27] D. Logothetis, et al. Stateful bulk processing for incremental analytics. In *SoCC*, pp. 51–62, 2010.

[28] E. Mazur, et al. Towards scalable one-pass analytics using mapreduce. In *IPDPS Workshops*, pages 1102–1111, 2011.

[29] G. Mishne, et al. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In *SIGMOD*, 2013.

[30] K. Morton, et al. Paratimer: a progress indicator for mapreduce dags. In *SIGMOD*, pp. 507–518, 2010.

[31] D. G. Murray, et al. Naiad: A timely dataflow system. In *SOSP*, pp. 439–455, 2013.

[32] L. Neumeyer, et al. S4: Distributed stream computing platform. In *ICDMW*, pp. 170–177, 2010.

[33] T. J. Ott. Simple inequalities for the d/g/1 queue. *Operations Research*, 35(4):pp. 589–597, 1987.

[34] K. Ousterhout, et al. Sparrow: distributed, low latency scheduling. In *SOSP*, pp. 69–84, 2013.

[35] Z. Qian, et al. Timestream: Reliable stream computation in the cloud. In *EuroSys*, pp. 1–14, 2013.

[36] N. Tatbul, et al. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pp. 159–170, 2007.

[37] A. Toshniwal, et al. Storm@twitter. In *SIGMOD*, pp. 147–156, 2014.

[38] F. Yang, et al. Sonora: A platform for continuous mobile-cloud computing. *Technical Report, Microsoft Research Aisa*, 2012.

[39] M. Zaharia, et al. Discretized streams: fault-tolerant streaming computation at scale. In SOSP, pp. 423–438, 2013.

[40] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.

[41] Q. Zou, et al. From a stream of relational queries to distributed stream processing. *PVLDB*, 3(1-2):1394–1405, 2010.