A Platform for Scalable One-Pass Analytics using MapReduce

Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, Prashant Shenoy
Department of Computer Science
University of Massachusetts, Amherst, Massachusetts, USA
{boduo, mazur, yanlei, mcgregor, shenoy}@cs.umass.edu

ABSTRACT

Today's one-pass analytics applications tend to be data-intensive in nature and require the ability to process high volumes of data efficiently. MapReduce is a popular programming model for processing large datasets using a cluster of machines. However, the traditional MapReduce model is not well-suited for one-pass analytics, since it is geared towards batch processing and requires the data set to be fully loaded into the cluster before running analytical queries. This paper examines, from a systems standpoint, what architectural design changes are necessary to bring the benefits of the MapReduce model to incremental one-pass analytics. Our empirical and theoretical analyses of Hadoop-based MapReduce systems show that the widely-used sort-merge implementation for partitioning and parallel processing poses a fundamental barrier to incremental one-pass analytics, despite various optimizations. To address these limitations, we propose a new data analysis platform that employs hash techniques to enable fast in-memory processing, and a new frequent key based technique to extend such processing to workloads that require a large key-state space. Evaluation of our Hadoop-based prototype using real-world workloads shows that our new platform significantly improves the progress of map tasks, allows the reduce progress to keep up with the map progress, with up to 3 orders of magnitude reduction of internal data spills, and enables results to be returned continuously during the job.

1. INTRODUCTION

Today, real-time analytics on large, continuously-updated datasets has become essential to meet many enterprise business needs. Like traditional warehouse applications, real-time analytics using incremental one-pass processing tends to be data-intensive in nature and requires the ability to collect and analyze enormous datasets efficiently. At the same time, MapReduce has emerged as a popular model for parallel processing of large datasets using a commodity cluster of machines. The key benefits of this model are that it harnesses compute and I/O parallelism on commodity hardware and can easily scale as the datasets grow in size. However, the MapReduce model is not well-suited for incremental one-pass analytics since it is primarily designed for batch processing of queries on large datasets. Furthermore, MapReduce implementations require the entire data set to be loaded into the cluster before running analytical queries, thereby incurring long latencies and making them unsuitable for producing incremental results.

In this paper, we take a step towards bringing the many benefits of the MapReduce model to incremental one-pass analytics. In the new model, the MapReduce system *reads input data only once, performs incremental processing as more data is read, and utilizes system resources efficiently to achieve high performance and scalability*. Our goal is to design a platform to support such scalable, incremental one-pass analytics. This platform can be used to support interactive data analysis, which may involve online aggregation with early approximate answers, and, in the future, stream query processing, which provides near real-time insights as new data arrives.

We argue that, in order to support incremental one-pass analytics, a MapReduce system should avoid any blocking operations and also computational and I/O bottlenecks that prevent data from "smoothly" flowing through map and reduce phases on the processing pipeline. We further argue that, from a performance standpoint, the system needs to perform *fast in-memory processing* of a MapReduce query program for all, or most, of the data. In the event that some subset of data has to be staged to disks, the I/O cost of such disk operations must be minimized.

Our recent benchmarking study evaluated existing MapReduce platforms including Hadoop and MapReduce Online (which performs pipelining of intermediate data [5]). Our results revealed that the main mechanism for parallel processing used in these systems, based on a sort-merge technique, is subject to significant CPU and I/O bottlenecks as well as blocking: In particular, we found that the sort step is CPU-intensive, whereas the merge step is potentially blocking and can incur significant I/O costs due to intermediate data. Furthermore, MapReduce Online's pipelining functionality only redistributes workloads between the map and reduce tasks, and is not effective for reducing blocking or I/O overhead.

Building on these benchmarking results, in this paper we perform an in-depth analysis of Hadoop, using a theoretically sound analytical model to explain the empirical results. Given the complexity of the Hadoop software and its myriad of configuration parameters, we seek to understand whether the above performance limitations are inherent to Hadoop or whether tuning of key system parameters can overcome these drawbacks, from the standpoint of incremental one-pass analytics. Our key results are two-fold: We show that our analytical model can be used to choose appropriate values of Hadoop parameters, thereby reducing I/O and startup costs. However, both theoretical and empirical analyses show that the sort-merge implementation, used to support partitioning and parallel processing, poses a fundamental barrier to incremental one-pass analytics. Despite a range of optimizations, I/O and CPU bottlenecks as well as blocking persists, and the reduce progress falls significantly behind the map progress.

We next propose a new data analysis platform, based on MapReduce, that is geared for incremental one-pass analytics. Based on the insights from our experimental and analytical evaluation of current platforms, we design two key mechanisms into MapReduce:

Our first mechanism replaces the sort-merge implementation in MapReduce with a purely hash-based framework, which is designed to address the computational and I/O bottlenecks as well as blocking behavior of sort-merge. We devise two hash techniques to suit

different user reduce functions, depending on whether the reduce function permits incremental processing. Besides eliminating the sorting cost from the map tasks, these hash techniques enable fast in-memory processing of the reduce function when the memory reaches a sufficient size as determined by the workload and algorithm.

Our second mechanism further brings the benefits of fast in-memory processing to workloads that require a large key-state space that far exceeds available memory. We propose an efficient technique to identify frequent keys and then update their states using a full in-memory processing path, both saving I/Os and enabling early answers for these keys. Less frequent keys trigger I/Os to stage data to disk but have limited impact on the overall efficiency.

We have built a prototype of our incremental one-pass analytics platform on Hadoop 0.20.1. Using a range of workloads in click stream analysis and web document analysis, our results show that our hash techniques significantly improve the progress of the map tasks, due to the elimination of sorting, and given sufficient memory, enable fast in-memory processing of the reduce function. For challenging workloads that require a large key-state space, our frequent-key mechanism significantly reduces I/Os and enables the reduce progress to keep up with the map progress, thereby realizing incremental processing. For instance, for sessionization over a click stream, the reducers output user sessions as data is read and finish as soon as all mappers finish reading the data in 34.5 minutes, triggering only 0.1GB internal data spill to disk in the job. In contrast, the original Hadoop system returns all the results towards the end of the 81 minute job, writing 370GB internal data spill to disk.

2. BACKGROUND

To provide a technical context for the discussion in this paper, we begin with background on MapReduce systems.

2.1 The MapReduce Model

At the API level, the MapReduce *programming model* simply includes two functions: The `map` function transforms input data into ⟨key, value⟩ pairs, and the `reduce` function is applied to each list of values that correspond to the same key. This programming model abstracts away complex distributed systems issues, thereby providing users with rapid utilization of computing resources.

To achieve parallelism, the MapReduce system essentially implements "*group data by key, then apply the reduce function to each group*". This *computation model*, referred to as MapReduce group-by, permits parallelism because both the extraction of ⟨key, value⟩ pairs and the application of the reduce function to each group can be performed in parallel on many nodes. The system code of MapReduce implements this computation model (and other functionality such as load balancing and fault tolerance).

The MapReduce program of an analytical query includes both the map and reduce functions compiled from the query (e.g., using a MapReduce-based query compiler [15]) and the MapReduce system's code for parallelism.

2.2 Common MapReduce Implementations

Hadoop. We first consider Hadoop, the most popular open-source implementation of MapReduce. Hadoop uses block-level scheduling and a sort-merge technique [20] to implement the group-by functionality for parallel processing (Google's MapReduce system is reported to use a similar implementation [6], but further details are lacking due to the use of proprietary code).

The Hadoop Distributed File System (HDFS) handles the reading of job input data and writing of job output data. The unit of data storage in HDFS is a 64MB block by default and can be set to other values during configuration. These blocks serve as the task granularity for MapReduce jobs.

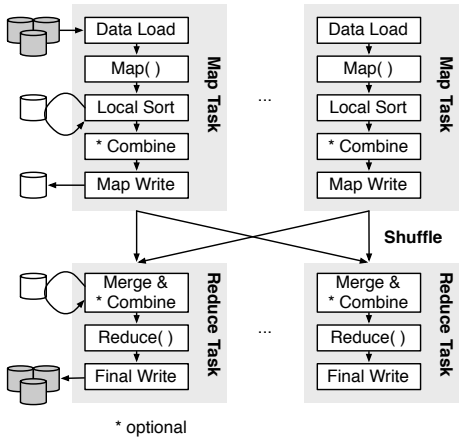


Figure 1: Architecture of the Hadoop implementation of MapReduce.

Given a query job, several map tasks (mappers) and reduce tasks (reducers) are started to run concurrently on each node. As Fig. 1 shows, each mapper reads a chunk of input data, applies the map function to extract ⟨key, value⟩ pairs, then assigns these data items to partitions that correspond to different reducers, and finally sorts the data items in each partition by the key. Hadoop currently performs a *sort* on the compound ⟨partition, key⟩ to achieve both partitioning and sorting in each partition. Given the relatively small block size, a properly-tuned buffer will allow such sorting to complete in memory. Then the sorted map output is written to disk for fault tolerance. A mapper completes after the write finishes.

Map output is then shuffled to the reducers. To do so, reducers periodically poll a centralized service asking about completed mappers and once notified, requests data directly from the completed mappers. In most cases, this data transfer happens soon after a mapper completes and so this data is available in the mapper's memory.

Over time, a reducer collects pieces of sorted output from many completed mappers. Unlike before, this data cannot be assumed to fit in memory for large workloads. As the reducer's buffer fills up, these sorted pieces of data are merged and written to a file on disk. A background thread merges these on-disk files progressively whenever the number of such files exceeds a threshold (in a so-called *multi-pass merge* phase). When a reducer has collected all of the map output, it will proceed to complete the multi-pass merge so that the number of on-disk files becomes less than the threshold. Then it will perform a final merge to produce all ⟨key, value⟩ pairs in sorted order of the key. As the final merge proceeds, the reducer applies the reduce function to each group of values that share the same key, and writes the reduce output back to HDFS.

Additionally, if the reduce function is commutative and associative, as shown in Fig. 1, a `combine` function is applied after the map function to perform partial aggregation. It can be further applied in each reducer when its input data buffer fills up.

MapReduce Online. We next consider an advanced system, MapReduce Online, that implements a Hadoop Online Prototype (HOP) with pipelining of data [5]. This prototype has two unique features: First, as each mapper produces output, it can push data eagerly to the reducers, with the granularity of transmission controlled by a parameter. Second, an adaptive mechanism is used to balance the work between the mappers and reducers. A potential benefit of HOP is that with pipelining, reducers receive map output earlier and can begin multi-pass merge earlier, thereby reducing the time required for the multi-pass merge after all mappers finish.

Table 1: Workloads and their running time in the benchmark.

	Click Streams			Web Documents
Setting	*Sessionization*	*Page frequency*	*Per-user count*	*Inverted Index*
Input data	256GB	508GB	256GB	427GB
Map output data	269GB	1.8GB	2.6 GB	150GB
Reduce spill data	370GB	0.2GB	1.4 GB	150GB
Intermediate/input	250%	0.4%	1.0%	70%
Output data	256GB	0.02GB	0.6GB	103GB
Map tasks	3,773	7,580	3,773	6,803
Reduce tasks	40	40	40	40
Completion time	76 min.	40 min.	24 min.	118 min.

3. BENCHMARKING AND ANALYSIS

The requirements for scalable one-pass analytics, namely, *incremental processing* and *fast in-memory processing* whenever possible, require the entire MapReduce program of a query to be non-blocking and have low CPU and I/O overhead. In this section, we examine whether current MapReduce systems satisfy these requirements.

3.1 Experimental Setup

We consider two applications in benchmarking: click stream analysis which represents workloads for stream processing, and web document analysis which represents workloads for one-pass analysis over stored data. The workloads tested are summarized in Table 1. (In ongoing work, we are extending our benchmark to Twitter feed analysis and complex queries such as top-k and graph queries.) [1]

In click stream analysis, an important task is sessionization, which reorders click logs into individual user sessions. Its MapReduce program employs the map function to extract the url and user id from each click log, then groups click logs by user id, and implements the sessionization algorithm in the reduce function. A key feature of this task is a large amount of intermediate data due to the reorganization of all click logs by user id. Another task in click stream analysis is page frequency counting. As a simple variant on the canonical word counting problem, it counts the number of visits to each url. A similar task counts the number of clicks that each user has made. For such counting problems, a combine function can be applied to significantly reduce the amount of intermediate data. For this application, we use the click logs from the World Cup 1998 website [2] and replicate it to larger sizes as needed.

The second application is web document analysis. A key task is inverted index construction, in which a large collection of web documents (or newly crawled web documents) is parsed and an inverted index on the occurrences of each word in those documents is created. In its MapReduce program, the map function extracts (word, (doc id, position)) pairs and the reduce function builds a list of document ids and positions for each word. The intermediate data is typically smaller than the document collection itself, but still of a substantial size. Other useful tasks in this application involve word frequency analysis, which are similar to page frequency analysis mentioned above, hence omitted in Table 1. For this application, we use the 427GB GOV2 document collection created from an early 2004 crawl of government websites.[3]

Our test cluster contains ten compute nodes and one head node. It runs CentOS 5.4, Sun Java 1.6u16, and Hadoop 0.20.1. Each compute node has 4 2.83GHz Intel Xeon cores, 8GB RAM, a 250GB Western Digital RE3 HDD, and a 64GB Intel X25-E SSD. The Hadoop configuration used the default setting and 4 reducers per node unless stated otherwise. The JVM heap size was 1GB, and map and reduce buffers were about 140MB and 500MB, respectively. All I/O operations used the disk as the default storage device. We ran the NameNode and JobTracker daemons on the head node and ran DataNode and TaskTracker daemons on each of the 10 compute nodes. The HDFS block size was 64MB. HDFS replication was turned down to 1 from the default 3.

A variety of tools are used for profiling, all of which have been packaged into a single program for simplicity. This program launches standard utilities such as `iostat` and `ps`, and logs the output to a file. We use the logged information to track metrics such as disk utilization and system CPU utilization. Hadoop-specific plots such as the task history were created by a publicly available parser.

3.2 Result Analysis

Table 1 shows the running time of the workloads as well as the sizes of input, output, and intermediate data in our benchmark. Due to space constraints, our analysis below focuses on the sessionization workload that involves the largest amount of intermediate data. We comment on the results of other workloads in the discussion whenever appropriate. Fig. 2 (a) shows the task timeline for the sessionization workload, i.e., the number of tasks for the four main operations in its MapReduce job: *map* (including sorting), *shuffle*, *merge* (the multi-pass part), and *reduce* (including the final scan to produce a single sorted run). As can be seen, time is roughly evenly split between the map and reduce phases, with a substantial merge phase in between. Also note that some periodic background merges take place even before all map tasks complete. When the intermediate data is reduced as in other workloads, first the merge phase shrinks and then the reduce phase also shrinks.

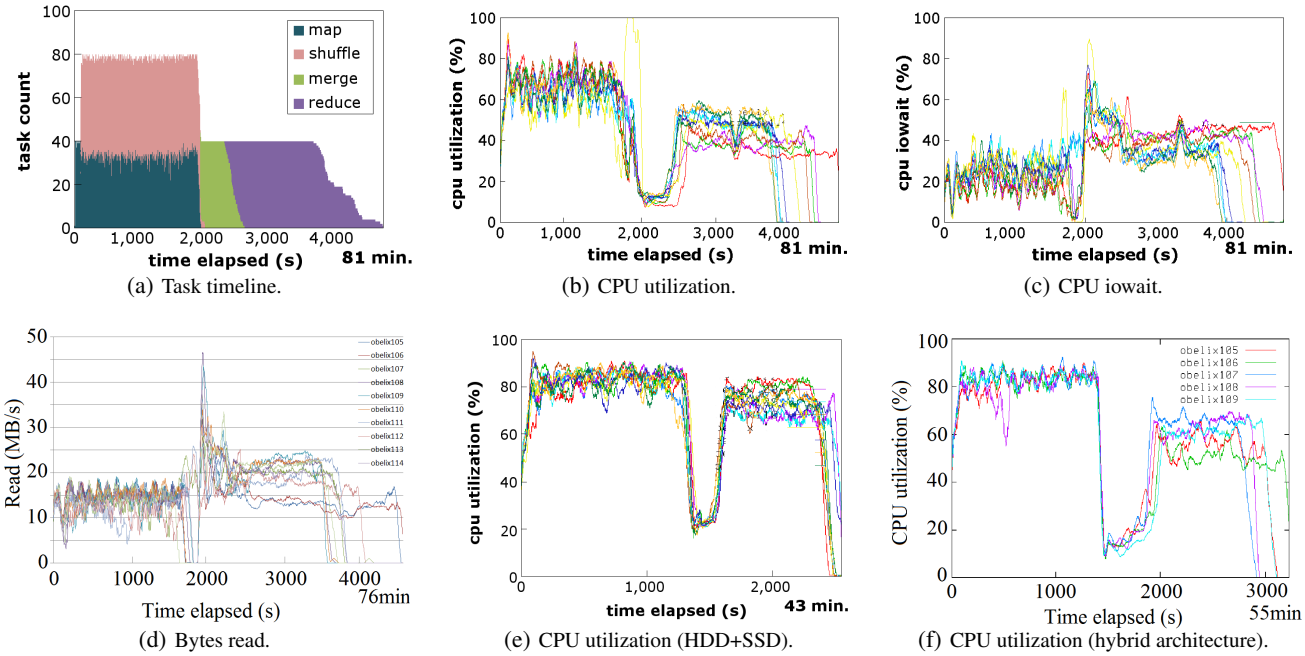
1. Cost of Parsing. A potential CPU bottleneck can be parsing line-oriented flat text files into the data types that map functions expect. To investigate this possibility, we prepared two different formats of the same data to use as input for the sessionization workload. The first format is the original line-oriented text files, leaving the task of extracting user ids to a regular expression in the map function. The second format is the same data preprocessed into Hadoop's `SequenceFile` binary format, allowing the map function to immediately operate on the data without having to do any parsing. We ran the sessionization workload on these two inputs and observed almost no difference in either running time or CPU utilization between the jobs. We therefore concluded that input parsing is a negligible overall cost.

2. Cost of Map Output. A potential I/O bottleneck can be the writes of map output to disk using synchronous I/O, required for fault tolerance in MapReduce. In our benchmark, we observed that although each map task did block while performing this write, it did not take up a large portion of a map task's lifetime. In the

[1] An existing benchmark [16] mostly contains simple aggregate queries over stored data. Our benchmark includes more complex tasks required in real-world applications, many of which are performed on data streams.

[2] http://ita.ee.lbl.gov/html/contrib/WorldCup.html

[3] http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm



(a) Task timeline. (b) CPU utilization. (c) CPU iowait.

(d) Bytes read. (e) CPU utilization (HDD+SSD). (f) CPU utilization (hybrid architecture).

Figure 2: Experimental results using the sessionization workload.

Table 2: Average CPU cycles per node, measured by CPU seconds, in the map phase (256GB worldcup dataset).

	Sessionization	*Per-user count*
Map function (%)	566 sec. (61%)	440 sec. (52%)
Sorting (%)	369 sec. (39%)	406 sec. (48%)

sessionization workload with a large amount of map output data, these writes took 1.3 seconds on average, while the average map task running time took 21.6 seconds. This 6% time did not make a significant contribution to a map task's running time relative to other parts. Furthermore, the recent MapReduce Online system [5] proposes to pipeline map output to the reducers and persists the data using asynchronous I/O. Hence, it can be used as a solution if the map output may be observed as an I/O bottleneck elsewhere.

3. Overhead of Sorting. Recall from §2 that when a map task finishes processing its input block, the key-value pairs must be partitioned according to different reducers and key-value pairs in each partition must be sorted to facilitate the merge in reducers. Hadoop accomplishes this task by performing a sort on the map output buffer on the compound of (partition, key).

First, we observe from Fig. 2 (b) that CPUs are busy in the map phase. It is important to note that the map function in the sessionization workload is relatively CPU light: it parses each click log into user id, timestamp, url, etc., and emits a key-value pair where the key is the user id and the value contains other attributes. The rest of cost in the map phase is attributed to sorting of the map output buffer. To quantify the costs of the map function and sorting, we performed detailed profiling of CPU cyles consumed by each, as shown in Table 2. In the sessionization workload, roughly 61% of CPU cycles were consumed by the map function while 39% was by sorting. In the per-user click counting workload, the map function simply emits pairs in the form of (user id, "1"), and up to 48% of CPU cycles were consumed by sorting these pairs. We further note that if we expedite click log parsing in the map function using the recent proposal of mutable parsing [10], the overhead of sorting will be even more prominent in the map phase.

Conclusion: Sorting of map output can introduce a significant CPU overhead, due to the use of the sort-merge implementation of the group-by operation in MapReduce.

4. Overhead of Merging. As map tasks complete and their output files are shuffled to the reducers, each reducer writes these files to disk (since there is not enough memory to hold all of them) and performs multi-pass merge: as soon as the number of on-disk files reaches F, it merges these files to a larger file and writes it back to disk. Such a merge will be triggered next time when the reducer sees F files on disk. This process continues until all map tasks have completed and the reducer has brought the number of on-disk files down to F. It completes by merging these on-disk files and feeding sorted data directly into the reduce function.

In the sessionization workload, the overhead of multi-pass merge is particularly noticeable when most map tasks have completed. In the CPU utilization plot in Fig. 2 (b), there is an extended period (from time 1800 to 2400) where the CPUs are mostly idle. While CPUs could be idle due to both disk I/O and network I/O, the CPU iowait graph in Fig. 2 (c) shows that it is largely due to outstanding disk I/O requests, and the graph in Fig. 2 (d) shows a large number of bytes read from disk in the same period. All of these observations match the merge activities shown between the map and reduce phases in the task timeline plot in Fig. 2 (a).

Overall, multi-pass merge is a blocking operation. The reduce function cannot be applied until this operation completes with all the data arranged into a single sorted run. This blocking effect causes low CPU utilization when most map tasks complete and prevents any answer from being returned by reducers for an extended period.

Moreover, the multi-pass merge operation is also I/O intensive. Our profiling tool shows that in sessionization, the reducers read and write 370GB data in the multi-pass merge operation while the input data has only 256 GB, as shown in Table 1. The inverted index workload incurs a somewhat reduced but still substantial I/O cost of 150GB data in this operation. As shown in Fig. 3, the blocking merge phase is present in this workload as well. Progress is stopped until local intermediate data is merged on each node. In

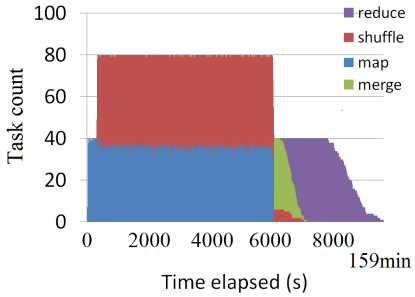


Figure 3: Task timeline using the inverted index construction workload.

simpler workloads, such as counting the number of clicks per user, there is an effective combine function to reduce the intermediate data size. However, it is interesting to observe from Table 1 that even if there is ample memory to perform in-memory processing, the multi-pass merge still causes I/O, e.g., 1.4GB spill from the reducers. This is because when the memory fills up, each reducer applies the combine function to the data in memory but still writes the data to disk waiting for all future data to produce a single sorted run.

Conclusion: The multi-pass merge operation is blocking. It is I/O intensive for workloads with large amounts of intermediate data. It may still cause I/O even if there is enough memory to hold all intermediate data.

3.3 Architectural Improvements

We next explore some architectural choices to investigate whether these changes can eliminate the blocking effect and the I/O bottleneck observed in the previous benchmark. The motivation is that when given a substantial amount of intermediate data, the disk utilization stays high for most time of a MapReduce job (e.g., over 90% in the sessionization workload). This is because the disk on each node not only serves the input data from HDFS and writes the final output to HDFS, but also handles intermediate data including the map output and the reduce spill in the multi-pass merge. Given a mix of requests from different MapReduce operations, the disk is often maxed out and subject to random I/Os.

Separate storage devices. One architectural improvement is to employ multiple devices per node for storage, thereby reducing disk contention in MapReduce operations. In this experiment, in addition to the existing hard disk, we add a solid state drive (64GB Intel SSD) to each node in the cluster. We use the hard disk to handle the input and output with HDFS and use the smaller, but faster, SSD to hold all intermediate data. This way, reading input data from HDFS and managing the intermediate data can proceed in parallel. In addition, the writes of map output and the reads/writes for multi-pass merge can benefit from the fast random access offered by the SSD.

We show the CPU utilization (among many other measurements) of the sessionization workload in Fig. 2 (e). The main observations include the following. Extra storage devices help reduce the total running time, from 76 minutes to 43 minutes for sessionization. Detailed profiling shows that roughly 2/3 of the performance benefit comes from having an extra storage device, and about 1/3 of it comes from the SSD characteristics themselves. However, there is still a significant period where the CPU utilization is low, demonstrating that the multi-pass merge continues to be blocking and involving intensive I/Os.

A separate distributed storage system. An alternative way to address the disk contention problem is to use separate systems to host the distributed storage and MapReduce computation. This is analogous to Amazon's Elastic MapReduce where the S3 system handles distributed storage and the EC2 system handles MapReduce computation with its local disks reserved for the use of intermediate data. This comes at the price of data locality though; tasks will no longer be able to be scheduled on the same nodes where data resides and so this architecture will incur additional network overhead. In our experiment, we simulate two subsystems by allocating 5 nodes to host the distributed storage and 5 nodes to serve as compute nodes for MapReduce. We reduce the input data size accordingly to keep the running time comparable to before.

Similar to the previous experiment, the separation of the distributed storage system helps reduce the running time of sessionization from 76 minutes to 55 minutes (which, however, does not have the benefits of SSDs). More importantly, the CPU utilization plot in Fig. 2 (f) shows that the issues of blocking and intensive I/O remain, which agrees with the previous experiment.

Conclusion: Architectural improvements can help reduce contention in storage device usage and decrease overall running time. However, they do not eliminate the blocking effect or the I/O bottleneck observed about the sort-merge implementation of MapReduce.

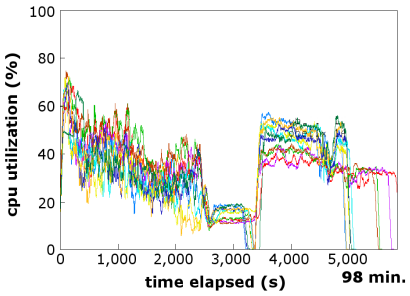
3.4 MapReduce Online

We finally consider a recent system called MapReduce Online that implements a Hadoop Online Prototype (HOP) with pipelining of data [5]. This prototype has two distinct features: First, as each map task produces output, it can push data eagerly to the reducers. The granularity of such data transmission is controlled by a parameter. Second, an adaptive control mechanism is in place to balance work between mappers and reducers. For instance, if the reducers become overloaded, the mappers will write the output to local disks and wait until reducers are able to keep up again. A potential benefit of HOP is that with pipelining, reducers receive map output earlier and can begin multi-pass merge earlier, thereby reducing the time required for the merge work after all mappers finish.

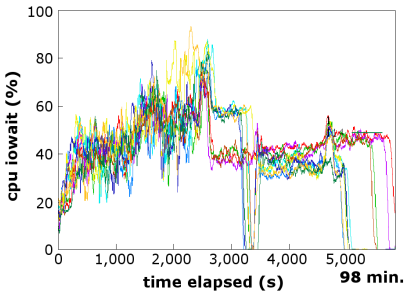
However, it is important to note that HOP adds pipelining to an overall blocking implementation of MapReduce based on sort-merge. As is known in the database literature, the sort-merge implementation of group by is an inherently blocking operation. HOP has a minor extension to periodically output snapshots (e.g., when reducers have received 25%, 50%, 75%, ..., of the data). This is done by repeating the merge operation for each snapshot. This is not real incremental computation desired in stream processing, and may incur a significant I/O overhead in doing so. Furthermore, such pipelining does not reduce CPU and I/O overhead but only redistributes workloads between mappers and reducers.

Fig. 4 shows some initial results of MapReduce Online using the sessionization workload. The most important observation is that the CPU utilization plot shows a similar pattern of low values in the middle of the job. While CPU can be idle due to both I/O wait and network wait (given the different communication model used in MapReduce Online), the CPU iowait graph again shows a spike in the middle of the job. Hence, our previous observations of blocking and I/O activity due to multi-pass merge still hold here.

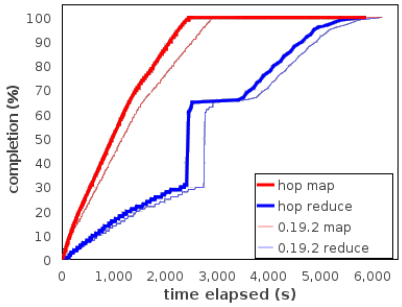
There are several subtle differences from the previous results of benchmarking Hadoop. The total running time is actually longer using MapReduce Online. A possible explanation for this difference is that MapReduce Online is based off an older version of Hadoop, 0.19.2, whereas we benchmarked using 0.20.0. Any performance optimizations made during this time will only be present in the newer version. Another possible reason is that MapReduce Online transmits map output eagerly in finer granularity and hence increases



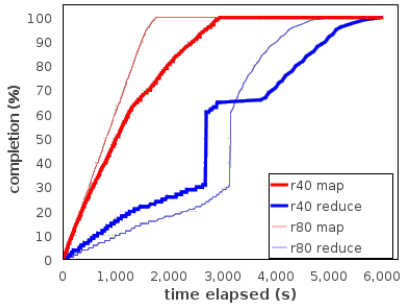
(a) CPU utilization.



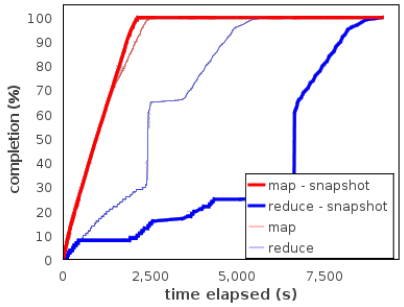
(b) CPU iowait.



(c) MR Online vs. Hadoop 0.19.2 (64MB)



(d) MR Online: 80 vs. 40 reducers.



(e) MR Online with 4 snapshots.

Figure 4: Results for MapReduce Online using the sessionization workload.

network cost, which in turn causes lower CPU utilization. Another thing to note is that the CPU utilization in the map phase when running HOP is lower than when running on stock Hadoop. We verified that the total number of CPU cycles consumed in the map phase are similar across both implementations by observing that HOP spends a greater amount of time in the map phase, with a somewhat reduced level of CPU utilization. Finally, this prototype moves some of the sorting work to reducers, which may also affect the CPU utilization in different phases of the job. In our ongoing work, we will continue benchmarking MapReduce Online, including the use of other workloads, to better explain its behavior.

3.5 Summary of Results

In this section, we benchmarked Hadoop and MapReduce Online which both use the sort-merge implementation of the group by operation in MapReduce. Our goal was to answer the question that we raised at the beginning of the study: Do current MapReduce systems satisfy the requirements for scalable one-pass analytics, namely, *incremental processing* and *fast in-memory processing* whenever possible? Our benchmarking results can be summarized as follows.

- The sorting step of the sort-merge implementation incurs high CPU cost, hence unsuitable for fast in-memory processing.
- Multi-pass merge in sort-merge is blocking and can incur high I/O cost given substantial intermediate data, hence not suitable for incremental processing or fast in-memory processing.
- Using extra storage devices and alternative storage architectures do not eliminate blocking or the I/O bottleneck.
- The Hadoop Online Prototype with pipelining does not eliminate blocking, the CPU bottleneck, or the I/O bottleneck.

4. OPTIMIZING HADOOP

Building on our previous benchmarking results, we perform an in-depth analysis of Hadoop in this section. Our goal is to understand whether the performance issues identified by our benchmarking study are inherent to Hadoop or whether they can be overcome by appropriate tuning of key system parameters.

4.1 An Analytical Model for Hadoop

The Hadoop system has a large number of parameters. While our previous experiments used the default settings, we examine these parameters more carefully in this study. After a nearly year-long effort to experiment with Hadoop, we identified several parameters that impact performance from the standpoint of incremental one-pass analytics, which are listed in Part (1) of Table 3. Our analysis below focuses on the effects of these parameters on I/O and startup costs. We do not aim to model the actual running time because it depends on numerous factors such as the actual server configuration, how map and reduces tasks are interleaved, how CPU and I/O operations are interleaved, and even how simultaneous I/O requests are served. Once we optimize these parameters based on our model, we will evaluate performance empirically using the actual running time and the progress with respect to incremental processing.

Our analysis makes several assumptions for simplicity: The MapReduce job under consideration does not use a combine function. Each reducer processes an equal number of $\langle$key, value$\rangle$ pairs. Finally, when a reducer pulls a mapper for data, the mapper has just finished so its output can be read directly from its local memory. The last assumption frees us from the onerous task of modeling the caching behavior at each node in a highly complex system.

1. Modeling I/O Cost in Bytes. We first analyze the I/O cost of the *existing* sort-merge implementation of Hadoop. We summarize our main result in the following proposition.

Proposition 4.1 *Given the workload description (D, K_m, K_r) and the hardware description (N, B_m, B_r), as defined in Table 3, the I/O*

Table 3: Symbols used in Hadoop analysis.

Symbol	Description
(1) System Settings	
R	Number of reduce tasks per node
C	Map input chunk size
F	Merge factor that controls how often on-disk files are merged
(2) Workload Description	
D	Input data size
K_m	Ratio of output size to input size for the map function
K_r	Ratio of output size to input size for the reduce function
(3) Hardware Resources	
N	Number of nodes in the cluster
B_m	Output buffer size per map task
B_r	Shuffle buffer size per reduce task
(4) Symbols Used in the Analysis	
U	Bytes read and written per node, $U = U_1 + \ldots + U_5$ where U_i is the number of bytes of the following types 1: map input; 2: map internal spills; 3: map output; 4: reduce internal spills; 5: reduce output
S	Number of sequential I/O requests per node, $S = S_1 + \ldots + U_S$ where S_i is the number of sequential I/O requests per node for I/O type i
T	Time measurement for startup and I/O cost
h	Height of the tree structure for multi-pass merge

cost in terms of bytes read and written in a Hadoop job is:

$$U = \frac{D}{N} \cdot (1 + K_m + K_m K_r) + \frac{2D}{CN} \cdot \lambda_F(\frac{CK_m}{B_m}, B_m) \cdot \mathbb{1}_{[C \cdot K_m > B_m]} + 2R \cdot \lambda_F(\frac{DK_m}{NRB_r}, B_r), \quad (1)$$

where $\mathbb{1}_{[\cdot]}$ is an indicator function, and $\lambda_F(\cdot)$ is defined to be:

$$\lambda_F(n,b) = \left(\frac{1}{2F(F-1)}n^2 + \frac{3}{2}n - \frac{F^2}{2(F-1)}\right) \cdot b. \quad (2)$$

Analysis. Our analysis includes five I/O-types listed in Table 3. Each map task reads a data chunk of size C as input, and writes $C \cdot K_m$ bytes as output. Given the workload D, we have D/C map tasks in total and $D/(C \cdot N)$ map tasks per node. So, the input cost, U_1, and output cost, U_3, of all map tasks on a node are:

$$U_1 = \frac{D}{N} \quad \text{and} \quad U_3 = \frac{D \cdot K_m}{N}.$$

The size of the reduce output on each node is $U_5 = \frac{D \cdot K_m \cdot K_r}{N}$.

Map and reduce internal spills result from the multi-pass merge operation, which can take place in a map task if the map output exceeds the memory size and hence needs to use external sorting, or in a reduce task if the reduce input data does not fit in memory.

We make a general analysis of multi-pass merge first. Suppose that our task is to merge n sorted runs, each of size b. As these initial sorted runs are generated, they are written to spill files on disk as $f_1, f_2, \ldots$ Whenever the number files on disk reaches $2F-1$, a background thread merges the *smallest* F files into a new file on disk. We label the new merged files as $m_1, m_2, \ldots$ Fig. 5 illustrates this process, where an unshaded box denotes an initial spill file and a shaded box denotes a merged file. For example, after the first $2F-1$ initial runs generated, $f_1, \ldots, f_F$ are merged together and the resulting files on disk are $m_1, f_{F+1}, \ldots, f_{2F-1}$ in order of decreasing size. Similarly, after the first F^2+F-1 initial runs are generated, the files on disk are $m_1, \ldots, m_F, f_{F^2+1}, \ldots, f_{F^2+F-1}$. Among them, $m_1, f_{F^2+1}, \ldots, f_{F^2+F-1}$ will be merged together and the resulting files on disk will be $m_{F+1}, m_2, \ldots, m_F$ in order of decreasing size. After the initial runs, a final merge combines all the remaining files (there are at most $2F-1$ of them).

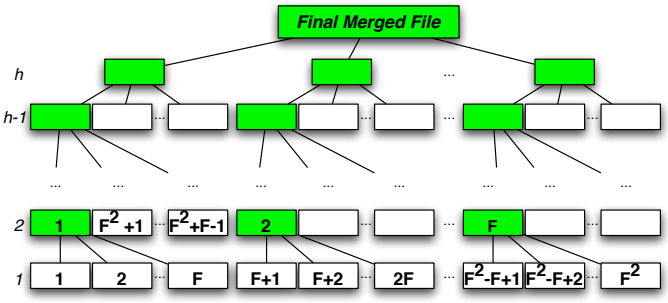


Figure 5: Analysis of the tree of files created in multi-pass merge.

For the analysis, let α_i denote the size of a merged file on level i ($2 \le i \le h$) and let $\alpha_1 = b$. Then $\alpha_i = \alpha_{i-1} + (F-1)b$. Solving this recursively gives $\alpha_i = (i-1)Fb - (i-2)b$. Hence, the total size of all the files in the first h levels is:

$$F(\alpha_h + \sum_{i=1}^{h-1}(\alpha_i + (F-1)b)) = bF\left(hF + \frac{(F-1)(h-2)(h+1)}{2}\right).$$

If we count all the spill files (unshaded boxes) in the tree, we have $n = (F + (F-1)(h-2))F$. Then we substitute h with n and F using the above formula and get

$$\lambda_F(n,b) = \left(\frac{1}{2F(F-1)}n^2 + \frac{3}{2}n - \frac{F^2}{2(F-1)}\right) \cdot b$$

Then, the total I/O cost is $2\lambda_F(n,b)$ as each file is written once and read once. The remaining issue is to derive the exact numbers for n and b in the multi-pass merge in a map or reduce task.

In a map task, if its output fits in the map buffer, then the merge operation is not needed. Otherwise, we use the available memory to produce sorted runs of size B_m each and later merge them back. So, $b = B_m$ and $n = \frac{C \cdot K_m}{B_m}$. As each node handles $D/(C \cdot N)$ map tasks, we have the I/O cost for map internal spills on this node as:

$$U_2 = \begin{cases} \frac{2D}{C \cdot N} \cdot \lambda_F(\frac{C \cdot K_m}{B_m}, B_m) & \text{if } C \cdot K_m > B_m; \\ 0 & \text{otherwise.} \end{cases}$$

In a reduce task, as we do not have a combine function, the input for reduce usually cannot fit in memory. The size of input to each reduce task is $\frac{D \cdot K_m}{N \cdot R}$. So, $b = B_r$ and $n = \frac{D \cdot K_m}{N \cdot R \cdot B_r}$. As each node handles R reduce tasks, we have the reduce internal spill cost:

$$U_4 = 2R \cdot \lambda_F(\frac{D \cdot K_m}{N \cdot R \cdot B_r}, B_r)$$

Summing up $U_1, \ldots, U_5$, we then have Eq. 1 in the proposition.

2. Modeling the Number of I/O requests. The following proposition models the number of I/O requests in a Hadoop job.

Proposition 4.2 *Given the workload description (D, K_m, K_r) and the hardware description (N, B_m, B_r), as defined in Table 3, the number of I/O requests in a Hadoop job is:*

$$S = \frac{D}{CN}\left(\alpha + 1 + \mathbb{1}_{[CK_m > B_m]} \cdot \left(\lambda_F(\alpha,1)(\sqrt{F}+1)^2 + \alpha - 1\right)\right) + R\left(\beta K_r(\sqrt{F}+1) - \beta\sqrt{F} + \lambda_F(\beta,1)(\sqrt{F}+1)^2\right), \quad (3)$$

where $\alpha = \frac{CK_m}{B_m}$, $\beta = \frac{DK_m}{NRB_r}$, $\lambda_F(\cdot)$ is defined in Eq. 2, and $\mathbb{1}_{[\cdot]}$ is an indicator function.

Analysis. We again consider the five types of I/O listed in Table 3. For each map task, a chunk of input data is sequentially read until the map output buffer fills up or the chunk is completely finished. So, the number of I/O requests for the map input $\frac{C \cdot K_m}{B_m}$. All map tasks on a node will trigger the number of I/O requests, S_1, as:

$$S_1 = (\frac{D}{CN}) \cdot (\frac{CK_m}{B_m}).$$

If the map output fits in memory, there is no internal spill and the map output is written to disk using one sequential I/O. Considering all map tasks on a node, we have

$$S_2 + S_3 = \frac{D}{CN} \quad \text{if} \quad CK_m \leq B_m \ .$$

If the map output exceeds the memory size, it is sorted using external sorting which involves multi-pass merge.

Since both map and reduce tasks may involve multi-pass merge, we first do a general analysis of the I/O requests incurred in this process. How many I/O requests to make depends on not only the data sizes but also the memory allocation scheme which can vary with the implementation and system resources available, Hence, we consider the optimal scheme regarding the I/O requests below.

Suppose that a merge step is to merge F files, each of size f, into a new file with memory size B. For simplicity, we assume the buffer size for each input file is the the same, denoted by B_{in}. So, the buffer size for the output file is $B - F \cdot B_{in}$. The number of read and write requests is $s = \frac{F \cdot f}{B_{in}} + \frac{F \cdot f}{B - F \cdot B_{in}}$. By taking the derivative with respect to B_{in} we can minimize s:

$$s^{opt} = \frac{F \cdot f}{B}(\sqrt{F}+1)^2 \quad \text{when} \quad B_{in}^{opt} = \frac{B}{F + \sqrt{F}} \ .$$

Revisit the tree of files in multi-pass merge in Fig. 5. Each merge step corresponds to the creation of a merged file (shaded box) in the tree. When we sum up the I/O requests of all these steps, we can apply our previous result on the total size of all the files: $\sum_j s_j^{opt} = \frac{\sum_j F \cdot f_j}{B}(\sqrt{F}+1)^2 = \frac{\lambda_F(n,b)}{B}(\sqrt{F}+1)^2$, where n is the number of initial spill files containing sorted runs and b is the size of each sorted run. But this above analysis does not include the I/O requests of writing the n initial sorted runs from memory to disk, so we add n requests and have the total number:

$$s^{merge} = n + \frac{\lambda_F(n,b)}{B}(\sqrt{F}+1)^2. \tag{4}$$

The value of n and b in map and reduce tasks have been analyzed previously. In a map task, if $CK_m > B_m$, then multi-pass merge takes place. For the above formula, $B = B_m, b = B_m$ and $n = \frac{CK_m}{B_m}$. Considering all $\frac{D}{CN}$ map tasks on a node, we have:

$$S_2 + S_3 = \frac{D}{CN}\left(\frac{CK_m}{B_m} + \lambda_F(\frac{CK_m}{B_m}, 1)(\sqrt{F}+1)^2\right) \quad \text{if} \quad CK_m > B_m.$$

For a reduce task, we have $B = B_r, b = B_r$ and $n = \frac{DK_m}{NRB_r}$. We can get the I/O requests by plugging these values in Eq. 4. However, this result includes the disk requests for writing output in the final merge step, which does not actually exist because the output of the final merge is directly fed to the reduce function. The overestimation is the number of requests for writing data of size $\frac{DK_m}{NR}$ with an output buffer of size $B_r - F \cdot B_{in}^{opt} = \frac{B_r}{\sqrt{F}+1}$. So, the overestimated number of requests is $\frac{DK_m(\sqrt{F}+1)}{NRB_r}$. Given R reduce tasks per node, we have:

$$S_4 = R\left(\lambda_F(\frac{DK_m}{NRB_r}, 1)(\sqrt{F}+1)^2 - \frac{DK_m}{NRB_r} \cdot \sqrt{F}\right).$$

Finally, the output size of a reducer task is $\frac{DK_mK_r}{NR}$, written to disk with an output buffer of size $\frac{B_r}{\sqrt{F}+1}$. So, we can estimate the I/O requests for all reduce tasks on a node, S_5, with

$$S_5 = R\left(\frac{DK_m}{NRB_r} \cdot K_r(\sqrt{F}+1)\right).$$

The sum of $S_1, \ldots, S_5$ gives the result in the proposition.

We note that for common workloads, the I/O cost is dominated by the cost of reading and writing all the bytes, not the seek time. [4]

3. Modeling the Startup Cost. Since the number of map tasks is usually much larger than reduce tasks, we consider the startup cost for map tasks. If c_m is the cost in second of creating a map task, the total map startup cost per node is $c_{start} \cdot \frac{D}{CN}$.

4. Combining All in Time Measurement. Let U be the number of bytes read and written in a Hadoop job and let S be the number of I/O requests made. Let c_{byte} denote the sequential I/O time per byte and c_{seek} denote the disk seek time for each I/O request. We define the time measurement T that combines the cost of reading and writing all the bytes, the seek cost of all I/O requests, and the map startup cost as follows:

$$T = c_{byte} \cdot U + c_{seek} \cdot S + c_{start} \cdot \frac{D}{CN}. \tag{5}$$

The above formula is our complete analytical model that captures the effects of the involved parameters.

4.2 Optimizing Hadoop based on the Model

Our analytical model enables us to predict system behaviors as Hadoop parameters vary. Then, given a workload and system configuration, we can choose values of these parameters that minimize the time cost in our model, thereby optimizing Hadoop performance.

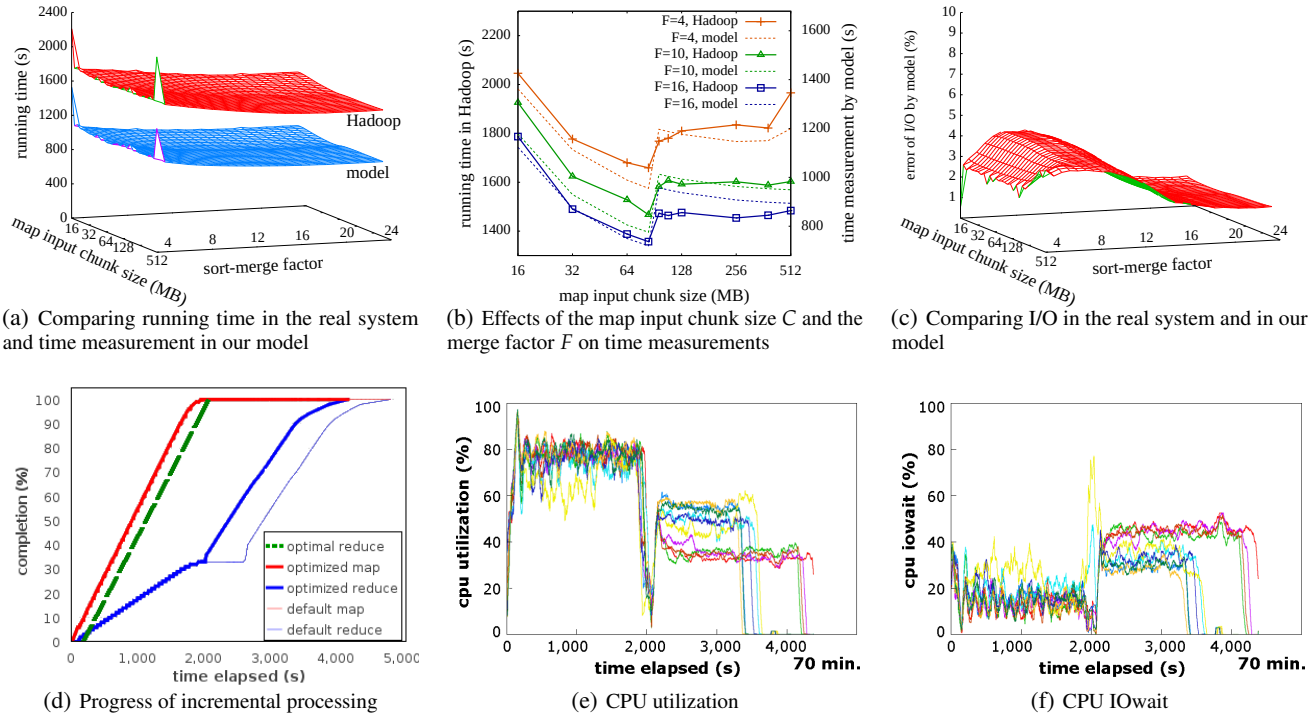
Optimizations. To show the effectiveness of our model, we compare the predicted system behavior based on our model and the actual running time measured in our Hadoop cluster. We used the sessionization task and configured the workload, our cluster, and Hadoop as follows: (1) Workload: D=97GB, K_m=K_r=1;[5] (2) Hardware: N=10, B_m=140MB, B_r=260MB; (3) Hadoop: R=4 or 8, and varied values of C and F. We also fed these parameter values to our analytical model. In addition, we set the constants in our model by assuming sequential disk access speed to be 80MB/s, disk seek time to be 4 ms, and the map task startup cost to be 100 ms.

Our first goal is to validate our model. In our experiment, we varied the map input chunk size, C, and the the merge factor, F. Under 100 different combinations of (C, F), we measured the running time in a real Hadoop system, and calculated the time cost predicted by our model. The result is shown as a 3-D plot in Fig. 6(a).[6] Note that our goal is not to compare the absolute values of these two time measurements: In fact, they are not directly comparable, as the former is simply a linear combination of the startup cost and I/O costs based on our model, whereas the latter is the actual running time affected by many system factors as stated above. Instead, we expect our model to predict the changes of the time measurement when parameters are tuned, so as to identify the optimal parameter

[4] For both propositions, in on-disk merge, if the number of initial sorted runs is less than the sort factor, to be more accurate, F should be set to the number of initial sorted runs, instead of the system setting.

[5] We used a smaller dataset in this set of experiments compared to the benchmark because changing Hadoop configurations often required reloading data into HDFS, which was very time-consuming.

[6] For either real running time or modeled time cost, the 100 data points were interpolated into a finer-grained mesh.



(a) Comparing running time in the real system and time measurement in our model

(b) Effects of the map input chunk size C and the merge factor F on time measurements

(c) Comparing I/O in the real system and in our model

(d) Progress of incremental processing

(e) CPU utilization

(f) CPU IOwait

Figure 6: Results of optimizing Hadoop parameters.

setting. Fig. 6(c) shows that indeed the performance predicted by our model and the actual running time exhibit very similar trends as the parameters C and F are varied.

Our next goal is to show how to optimize the parameters based on our model. To reveal more details from the 3-D plot, we show the results of a smaller range of (C, F) in Fig. 6(b), where the solid lines are for the actual running time and the dashed lines are for predication using our model.

(1) Optimizing the Chunk Size. When the chunk size C is very small, the MapReduce job uses many map tasks and the map startup cost dominates. As C increases, the map startup cost reduces, but once the map output exceeds its buffer size, multi-pass merge is incurred with increased I/O cost. The time cost jumps up at this point, and then remains nearly constant since the reduction of startup cost is not significant. When C exceeds a large size (whose exact value depends on the merge factor), the number of passes of on-disk merge goes up, thus incurring more I/Os. Overall, good performance is observed at the maximum value of C that allows the map output to fit in the buffer. Given a particular workload, we can easily estimate K_m, the ratio of output size to input size, for the map function and estimate the map output buffer size B_m to be about $\frac{2}{3}$ of the total map memory size (given the use of other metadata). Then we can choose the maximum C such that $C \cdot K_m \leq B_m$.

(2) Optimizing the Merge Factor. We then investigate the merge factor, F, that controls how frequently on-disk files are merged in the multi-pass merge phase. Fig. 6(b) shows three curves for three F values. The time cost decreases with larger values of F (from 4 to 16), mainly due to fewer I/O bytes incurred in the multi-pass merge. When F goes up to the number of initial sorted runs (around 16), the time cost does not decrease further because all the runs are merged in a single pass. For several other workloads tested, one-pass merge was also observed to provide the best performance.

Our model can also reveal potential benefits of small F values. When F is small, the number of files to merge in each step is small, so the reads of the input files and the writes of the output file are mostly sequential I/O. As such, a smaller F value incurs more I/O bytes, but fewer disk seeks. According to our model, the benefits of small F values can be shown only when the system is given limited memory but a very large data set, e.g., several terabytes per node, which is beyond the current storage capacity of our cluster.

(3) Effect of the Number of Reducers. The third relevant parameter is the number of reducers per node, R. The original MapReduce proposal [6] has recommended R to be the number of cores per node times a small constant (e.g., 1 or 2). As this parameter does not change the workload but only distributes it over a variable number of reduce workers, our model shows little difference as R varies. Empirically, we varied R from 4 to 8 (given 4 cores on each node) while configuring C and F using the most appropriate values as reported above. Interestingly, the run with R=4 took 4,187 seconds, whereas the run with R=8 took 4,723 seconds. The reasons are two-fold. First, by tuning the merge factor, F, we have minimized the work in multi-pass merge. Second, given 4 cores on each node, we have only 4 reduce task slots per node. Then for R=8, the reducers are started in two waves. In the first wave, 4 reducers are started. As some of these reducers finish, a reducer in the second wave can be started. As a consequence, the reducers in the first wave can read map output soon after their map tasks finish, hence directly from the local memory. In contrast, the reducers in the second wave are started long after the mappers have finished. So they have to fetch map output from disks, hence incurring high I/O costs in shuffling. Our conclusion is that optimizing the merge factor, F, can reduce the actual I/O cost in multi-pass merge, and is a more effective method than enlarging the number of reducers beyond the number of reduce task slots available at each node.

We also compared the I/O costs predicated by our model and those actually observed. Not only do we see matching trends, the predicted numbers are also close to the actual numbers. In Fig. 6(c), we show the error of the I/O size predicted by our model comparing

to the actual I/O size under different (C, F) combinations. The error is always less than 10%. All of the above results show that given a particular workload and hardware configuration, one can run our model to find the optimal values of the chunk size C and merge factor F, and choose an appropriate value of R based on the recommendation above.

Analysis of Optimized Hadoop. We finally reran the 240GB sessionization workload described in our benchmark (see §3). We optimized Hadoop using 64MB data chunks, one-pass merge, and 4 reducers per node as suggested by the above results. The total running time was reduced from 4,860 seconds to 4,187 seconds, a 14% reduction of the total running time.

Given our goal of one-pass analytics, a key requirement is to perform *incremental processing* and deliver a query answer as soon as all relevant data has arrived. In this regard, we propose metrics for the map and reduce progress, as defined below.

Definition 1 (Incremental Map and Reduce Progress) *The map progress is defined to be the percentage of map tasks that have completed. The reduce progress is defined to be:* $\frac{1}{3}\cdot$ *% of shuffle tasks completed +* $\frac{1}{3}\cdot$ *% of combine function or reduce function completed +* $\frac{1}{3}\cdot$ *% of reduce output produced.*

Note that our definition differs from the default Hadoop progress metric[7] where the reduce progress includes the work on multi-pass merge. In contrast, we discount multi-pass merge because it is irrelevant to a user query, and emphasize the actual work on the reduce function or combine function and the output of answers.

Fig. 6(d) shows the progress of optimized Hadoop in bold lines (and the progress of stock Hadoop in thin lines as a reference). The map progress increases steadily and reaches 100% around 2,000 seconds. The reduce progress increases to around 33% in these 200 seconds, mainly because the shuffle progress would keep up with the map progress. Then the reduce progress slows down, due to the overhead of merging, and lags far behind the map progress. The optimal reduce progress, as marked by a dashed line in this plot, keeps up with the map progress, thereby realizing fast incremental processing. As can be seen, there is a big gap between the optimal reduce progress and what the optimized Hadoop can currently achieve.

Fig. 6(e) and 6(f) further show the CPU utilization and CPU iowait using optimized Hadoop. We make two main observations: (1) The CPU utilization exhibits a smaller dip in the middle of a job compared to stock Hadoop in Fig. 2(b). However, the CPU cycles consumed by the mappers, shown as the area under the curves before 2,000 seconds, are about the same as those using stock Hadoop. Hence, the CPU overhead due to sorting, as mentioned in our benchmark, still exists. (2) The CPU iowait plot still shows a spike in the middle of job, due to the blocking of CPU by the I/O operations in the remaining single-pass merge.

We close the discussion in this section with the summary below:

- ► Our analytical model can be used to choose appropriate values of Hadoop parameters, thereby improving performance.
- ► Optimized Hadoop, however, still has a significant barrier to fast incremental processing: (1) The remaining one-pass merge can still incur blocking and a substantial I/O cost. (2) The reduce progress falls far behind the map progress. (3) The map tasks still have the high CPU cost of sorting.

[7] In Hadoop, the map progress is also defined to be the percentage of map tasks that have completed. By default, the reduce progress is defined as $\frac{1}{3}\cdot$ % of shuffle tasks completed + $\frac{1}{3}\cdot$ % of (multi-pass) merge tasks completed + $\frac{1}{3}\cdot$ % of reduce function completed. The progress plots in Fig. 4 are in the Hadoop default metric. From now on, we will instead use the incremental progress metric.

5. A NEW HASH-BASED PLATFORM

Based on the insights from our experimental and analytical evaluation of current MapReduce systems, we next propose a new data analysis platform that transforms MapReduce computation into incremental one-pass processing. Our first mechanism replaces the widely used sort-merge implementation for partitioning and parallel processing with a purely hash-based framework to minimize computational and I/O bottlenecks as well as blocking. Two hash techniques, designed for different types of reduce functions, are described in §5.1 and §5.2, respectively. These techniques enable fast in-memory processing when there is sufficient memory for the current workload. Our second mechanism further brings the benefits of fast in-memory processing to workloads that require a large key-state space that far exceeds available memory. Our technique efficiently identifies popular keys and updates their states using a full in-memory processing path. This mechanism is detailed in §5.3.

5.1 A Basic Hash Technique (MR-hash)

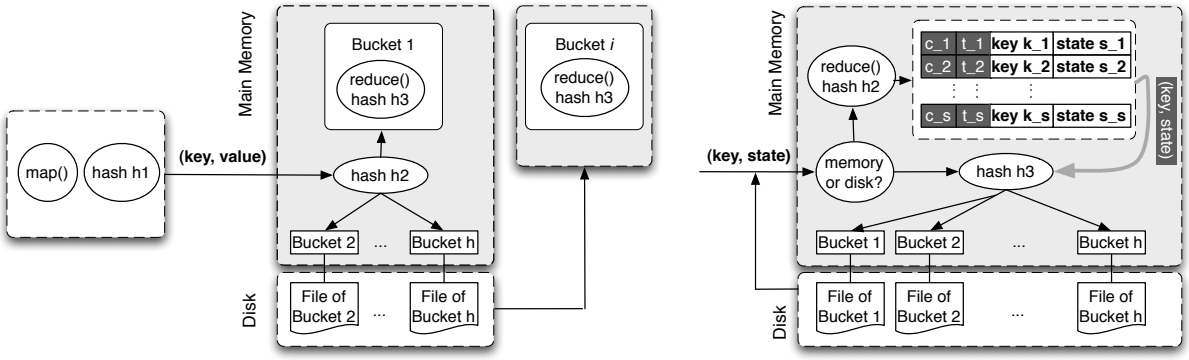
Recall from Section 2 that to support parallel processing, the MapReduce computation model essentially implements "*group data by key, then apply the reduce function to each group*". The main idea underlying our hash framework is to implement the MapReduce group-by functionality using a series of independent hash functions $h_1, h_2, h_3, \ldots$, across the mappers and reducers.

As depicted in Fig. 7(a), the hash function h_1 partitions the map output into subsets corresponding to the scheduled reducers. Hash functions $h_2, h_3, \ldots$, are used to implement (recursive) partitioning at each reducer. More specifically, h_2 partitions the input data to a reducer to n buckets, where the first bucket, say, D_1, is held completely in memory and other buckets are streamed out to disks as their write buffers fill up (which is similar to hybrid hash join [17]). This way, we can perform group-by on D_1 using the hash function h_3 and apply the reduce function to each group completely in memory. Other buckets are processed subsequently, one at a time, by reading the data from the disk. If a bucket D_i fits in memory, we use in-memory processing for the group-by and the reduce function; otherwise, we further partition it using hash function h_4, and so on. In our implementation, we use standard universal hashing to endure that the hash functions are independent of each other.

Following the analysis of the hybrid hash join [17], simple calculation shows that if h_2 can evenly distribute the data into buckets, recursive partitioning is not needed if the memory size is greater than $2\sqrt{|D_r|}$, where $|D_r|$ is the size of the data sent to the reducer, and the I/Os involve $2(|D_r| - |D_1|)$ bytes read and written. The number of buckets, h, can be derived from the standard analysis by solving a quadratic equation.

The above technique, called MR-hash, exactly matches the current MapReduce model that collects all the values of the same key into a list and feeds the entire list to the reduce function. This baseline technique in our work is similar to the hash technique used in parallel databases [8], but implemented in the MapReduce context. Compared to stock Hadoop, MR-hash offers several benefits: First, on the mapper side, it avoids the CPU cost of sorting as in the sort-merge implementation. Second, it allows early answers to be returned for the buffered bucket, D_1, as data comes to the reducer. If the application specifies a range of keys to be more important than others, we can design h_2 so that D_1 contains those important keys.

5.2 An Incremental Hash Technique (INC-hash)



(a) MR-hash: hashing in mappers and two phase hash processing in reducers. (b) Dynamic Inc-hash: monitoring keys and updating states.

Figure 7: Hash techniques in our new data analysis platform.

Our second hash technique is designed for reduce functions that permit incremental processing, including simple aggregates like sum and count, and more complex problems that have been studied in the area of sublinear-space stream algorithms [13]. In our work, we define three functions to implement incremental processing: The initialize function, $init()$, reduces a sequence of data items to a state; The combine function, $cb()$, reduces a sequence of states to a state; and the finalize function, $fn()$, produces a final answer from a state. The initialize function is applied immediately when the map function finishes processing. This changes the data in subsequent processing from the original key-value pairs to key-state pairs. The combine function can be applied to any intermediate step that collects a set of states for the same key, e.g., in the write buffers of a reducer that pack data to write to disks. Finally, the original reduce function is implemented by $cb()$ followed by $fn()$.

Such incremental processing can offer several benefits: The initialize function reduces the amount of data output from the mappers. In addition, existing data items can be collapsed to a compact *state* so that the reducer no longer needs to hold all the data in memory. Furthermore, as a result of incremental processing, query answers can be derived as soon as the relevant data is available, e.g., when the count in a group exceeds a query-specified threshold or when a window closes in window-based stream processing.

To realize the above benefits, we propose an alternative incremental hash implementation, called INC-hash. The algorithm is illustrated in Fig. 7(b) (the reader can ignore the darkened boxes for now, as they are used only in the third technique). As a reducer receives map output, which includes key-state pairs created by the initialize function, called tuples for simplicity, we build an in-memory hashtable H (using hash function h_2) that maps from a key to the state of computation. When a new tuple arrives, if its key already exists in H, we update the key's state with the new tuple using the combine function. If its key does not exist in H, we add a new key-state pair to H if there is still memory. Otherwise, we hash the tuple (using h_3) to a bucket, place the tuple in the write buffer of this bucket, and flush the write buffer when it becomes full. When the reducer has seen all the tuples and output the results for all the keys in H, it then reads disk-resident buckets back one at a time, repeating the procedure above to process each bucket.

The analysis of INC-hash turns out to be similar to that in Hybrid Cache for handling expensive predicates [9]. We summarize our main results below. The key improvement of INC-hash over MR-hash is that for those keys in H, their tuples are continuously collapsed into states in memory, avoiding I/Os for those tuples altogether. I/Os will be completely eliminated in INC-hash if the memory is large enough to hold all *distinct* key-state pairs, whose size is denoted by Δ, in contrast to all the data items in MR-hash. When memory size is less than Δ but greater than $\sqrt{\Delta}$, we can show that tuples that belong to H are simply collapsed into the states in memory, and other tuples are written out and read back exactly once—no recursive partitioning is needed in INC-hash. The number of buckets, h, can be derived directly from this analysis.

5.3 A Dynamic Incremental Hash Technique

Our last technique is an extension of the incremental hash approach where we dynamically determine which keys should be processed in memory and which keys will be written to disk for subsequent processing. The basic idea behind the new technique is to recognize hot keys that appear frequently in the data set and hold their states in memory, hence providing incremental in-memory processing for these keys. The benefits of doing so are two-fold. First, prioritizing these keys leads to greater I/O efficiency since in-memory processing of data items of hot keys can greatly decrease the volume of data that needs to be first written to disks and then read back to complete the processing. Second, it is often the case that the answers for the hot keys are more important to the user than the colder keys. Then this technique offers the user the ability to terminate the processing before data is read back from disk if the coverage of data is sufficiently large for those keys in memory.

Below we assume that we do not have enough memory to hold all states of *distinct* keys. Our mechanism for recognizing and processing hot keys builds upon ideas in an existing data stream algorithm called the FREQUENT algorithm [11, 3] that can be used to estimate the frequency of different values in a data stream. While we are not interested in the frequencies of the keys per se, we will use estimates of the frequency of each key to date to determine which keys should be processed in memory. However, note that other "sketch-based" algorithms for estimating frequencies will be unsuitable for our purposes because they do not explicitly encode a set of hot keys; Rather, additional processing is required to determine frequency estimates and then use them to determine approximate hot keys, which is too costly for us to consider.

Dynamic Incremental (DINC) Hash. We use the following notation in our discussion of the algorithm: Let K be the total number of *distinct* keys. Let M be the total number of key-state pairs in input, called tuples for simplicity. Suppose that the memory contains B pages, and each page can hold n_p key-state pairs with their associated auxiliary information. Let `cb` be a combine function that combines a state u and a state v to make a new state $\texttt{cb}(u,v)$.

While receiving tuples, each reducer divides the B pages in memory into two parts: h pages are used as write buffers, one for each of h files that will reside on disk, and $B-h$ pages for "hot" key-state

pairs. Hence, $s = (B-h)n_p$ keys can be processed in-memory at any given time.[8] Fig. 7(b) illustrates our algorithm.

Our algorithm maintains s counters $c[1],\ldots,c[s]$ and s associated keys $k[1],\ldots,k[s]$ referred to as "the keys currently being monitored" together with the state $s[i]$ of a partial computation for each key $k[i]$. Initially $c[i]=0, k[i]=\perp$ for all $i \in [s]$. When a new tuple (k,v) arrives, if this key is currently being monitored, $c[i]$ is incremented and $s[i]$ is updated using the combine function. If k is not being monitored and $c[j]=0$ for some j, then the key-state pair $(k[j],s[j])$ is evicted and $(c[j],k[j],s[j]) \leftarrow (1,k,v)$. If k is not monitored and all $c>0$, then the tuple needs to be written to disk and all $c[i]$ are decremented by one. Whenever the algorithm decides to evict a key-state pair in-memory or write out a tuple, it always first assigns the item to a hash bucket and then writes it out through the write buffer of the bucket, as in INC-hash.

Once the tuples have all arrived, most of the computation for the hot keys may have already been performed. At this point we have the option to terminate if the partial computation for hot keys is "good enough" in a sense we will make explicit shortly. If not, we proceed with performing all the remaining computation: we first write out each key-state pair currently in memory to disk to the appropriate bucket file. We then read each bucket file into memory and complete the processing for each key in the bucket file.

I/O Analysis. Suppose there are f_i tuples with key k_i and note that $M=\sum_i f_i$. Without loss of generality assume $f_1 \geq f_2 \geq \ldots \geq f_K$. Then the best we can hope for is performing $\sum_{1\leq i\leq s} f_i$ steps of in-memory computation as the tuples are being sent to the reducer. This is achieved if we know the "hot" keys, i.e., the top-s, in advance. Existing analysis for the FREQUENT algorithm can be applied to our new setting to show that the above strategy guarantees that $M' := \sum_{1\leq i\leq s} \max(0, f_i - \frac{M}{s+1})$ combine operations have been performed. Since every tuple that is not combined with an existing state in memory triggers a write-out, the number of tuples written to disk is $M-M'+s$ where the additional s comes from the write out of the hot key-state pairs in main memory. This result compares favorably with the offline optimal if there are some very popular keys, but does not give any guarantee if there are no keys whose relative frequency is more than $1/(s+1)$. If the data is skewed, the theoretical analysis can be improved [3]. Note that for INC-hash there is no guarantee on the steps of computation performed before the hash files are read back from disk. This is because the keys chosen for in-memory processing are just the first keys observed.

After the input is consumed, we write out all key-state pairs from main memory to the appropriate bucket file. Then the number of unique keys corresponding to each bucket file to be K/h. Consequently, if $K/h \leq B \cdot n_p$, then the key-state pairs in each bucket can be processed sequentially in memory. Setting h as small as possible increases s and hence decreases M'. Hence we set $h = Kn_p/B$.

To compare the different hash techniques, first note that the improvement of INC-hash over MR-hash is only significant when K is small. This is because the keys processed incrementally in main memory will only account for a small fraction of the tuples. DINC-hash mitigates this in the case when, although K may be large, some keys are considerably more frequent then other keys. By ensuring that it is these keys that are usually monitored in memory, we ensure that a large fraction of the tuples are processed before the remaining data is read back from disk.

Approximate Answers and Coverage Estimation. One of the features of DINC-hash is that a large fraction of the combine operations for a very frequent key will already have been performed once all the tuples have arrived. To estimate the number of combine operations performed for a given key we use the t values: these count the number of key-state tuples that have been combined for key k since most recent time k started being monitored. Define the *coverage* of key k_i to be

$$\text{coverage}(k_i) = \begin{cases} t[j]/f_i & \text{if } k[j]=k_i \text{ for some } j \\ 0 & \text{otherwise} \end{cases}.$$

Hence, once the tuples have arrived, the state corresponding to k_i in main-memory represents the computation performed on a coverage(k_i) fraction of all the tuples with this key. Unfortunately we do not know the coverage of a monitored key exactly, but it can be shown that we have a reasonably accurate under-estimate:

$$\gamma_i := t[j]/(t[j]+M/(s+1)) \leq t[j]/f_i = \text{coverage}(k_i)\,.$$

Hence, for a user-determined threshold ϕ, if $\gamma_i \geq \phi$ we can opt to return the state of the partial computation rather than to complete the computation.

6. PROTOTYPE IMPLEMENTATION

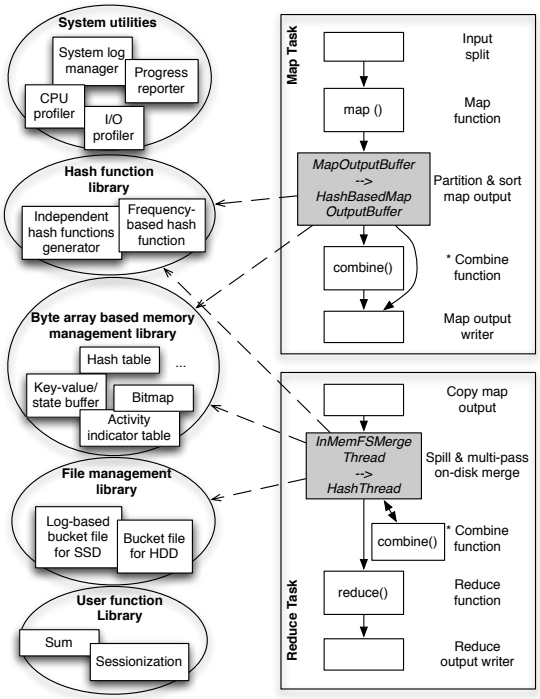


Figure 8: Architecture of our new one-pass analytics platform.

We have built a prototype of our incremental one-pass analytics platform on Hadoop. Our prototype is based on Hadoop version 0.20.1 and modifies the internals of Hadoop by replacing key components with our Hash-based and fast in-memory processing implementations. Figure 8 depicts the architecture of our prototype; the shaded components and the enlarged sub-components show the various portions of Hadoop internals that we have built. Broadly these modifications can be grouped into two main components.

Hash-based Map Output: Vanilla Hadoop consists of a Map Output Buffer component that manages the map output buffer, collects map output data, partitions the data for reducers, sorts the data by partition id and key (external sort if the data exceeds memory), and feeds the sorted data to the combine function if there is

[8] If we use $p>1$ pages for each of the h write buffers (to reduce random-writes), then $s = n_p \cdot (B-hp)$. We omit p below to simplify the discussion.

one or writes sorted runs to local disks otherwise. Since our design eliminates the sort phase, we replace this component with a new Hash-based Map Output component. Whenever a combine function is used, our Hash-based Map Output component builds an in-memory hash table for key-value pairs output by hashing on the corresponding keys. After the input has been processed, the values of the same key are fed to the combine function, one key at a time. In the scenario where no combine function is used, the map output must be grouped by partition id and there is no need to group by keys. In this case, our Hash-based Map Output component records the number of key-value pairs for each partition while processing the input data chunk, and moves records with the same key to a particular segment in the buffer, while scanning the buffer once.

HashThread Component: Vanilla Hadoop comprises an *InMemFSMerge* thread that performs in-memory and on-disk merges and writes data to disk whenever the shuffle buffer is full. Our prototype replaces this component with a HashThread implementation, and provides a user-configurable option to choose between MR-hash, INC-hash, and DINC-hash implementations within HashThread.

In order to avoid the performance overhead of creating a large number of Java objects, our prototype implements its own memory management by placing key data structures into byte arrays. Our current prototype includes several byte array-based memory managers to provide core functionality such as hash table, key-value or key-state buffer, bitmap, or counter-based activity indicator table, etc., to support our three hash-based approaches.

We also implement a bucket file manager that is optimized for hard disks and SSDs and provide a library of common combine and reduce functions as a convenience to the programmer. Our prototype also provides a set of independent hash functions, such as in recursive hybrid hash, in case such multiple hash functions are needed for analytics tasks. Also, if the frequency of hash keys is available a priori, our prototype can customize the hash function to balance the amount of data across buckets.

Finally, we implement several "utility" components such as a system log manager, a progress reporter for incremental computation, and CPU and I/O profilers to monitor system status.

7. PERFORMANCE EVALUATION

We present an experimental evaluation of our analytics platform and compare it to optimized Hadoop (1-pass SM) version 0.20.1. We evaluate all three hash techniques (MR-hash, INC-hash and DINC-hash) in terms of running time, the size of reduce spill data, and the progress made in map and reduce (by Definition 2).

In our evaluation, we use two real-world datasets: 236GB of the WorldCup click stream, and 156GB of the GOV2 dataset[9]. We use workloads over the WorldCup dataset: (1) **sessionization** where we split the click stream of each user into sessions; (2) **user click counting**, where we count the number of clicks made by each user; (3) **frequent user identification**, where we find users who click at least 50 times. We also use a fourth workload over the GOV2 dataset, **trigram counting**, where we report word trigrams that appear more than 1000 times. Our evaluation environment is a 10-node cluster as described in §3. Each compute node is set to hold a task tracker, a data node, four map slots, and four reduce slots. In each experiment, 4 reduce tasks run on each compute node.

7.1 Small Key-state Space

We first evaluate MR-hash and INC-hash under the workloads with small key-state space, where the distinct key-state pairs fit in memory or slightly exceed the memory size. We consider sessionization, user click counting, and frequent user identification.

Sessionization. To support incremental computation of sessionization in reduce, we configure INC-hash to use a fixed-size buffer that holds a user's clicks. A fixed size buffer is used since the order of the map output collected by a reducer is not guaranteed, and yet online sessionization relies on the temporal order of the input sequence. When the disorder of reduce input in the system is bounded, a sufficiently large buffer can guarantee the input order to the online sessionization algorithm. In the first experiment, we set the buffer size, i.e. the state size, to 0.5KB.

Fig. 9(a) shows the comparison of 1-pass SM, MR-hash, and INC-hash in terms of map and reduce progress. Before the map tasks finish, the reduce progress of 1-pass SM and MR-hash is blocked by 33%. MR-hash blocks since incremental computation is not supported. In 1-pass SM, the sort-merge mechanism blocks the reduce function until map tasks finish; a combine function can't be used here since all the records must be kept for output. In contrast, INC-hash's reduce progress keeps up with the map progress up to around 1,300s, because it performs incremental in-memory processing and generates pipelined output until the reduce memory is filled with states. After 1,300s, some data is spilled to disk, so the reduce progress slows down. After map tasks finish, it takes 1-pass SM and MR-hash longer to complete due to the large size of reduce spills (around 250GB as shown in Table 4). In contrast, INC-hash finishes earlier due to smaller reduce spills (51GB).

Thus by supporting incremental processing, INC-hash can provide earlier output, and generates less spill data, which further reduces the running time after the map tasks finish.

User click counting & Frequent user identification. In contrast to sessionization, user-click counting can employ a combine function and the states completely fit in memory at the reducers.

Fig. 9(b) shows the results for user click counting. 1-pass SM applies the combine function in each reducer whenever its buffer fills up, so its progress is more of a step function. Since MR-hash does not support the combine function, its overall progress only reaches 33% when the map tasks finish. In contrast, INC-hash makes steady progress through 66% due to its full incremental computation. Note that since this query does not allow any early output, no technique can progress beyond 66% until all map tasks finish.

This workload generates less shuffled data, reduce spill data, and output data when compared to sessionization (see Table 4). Hence the workload is not as disk- and network-I/O- intensive. Consequently both hash-based techniques have shorter running times, when compared to 1-pass SM, due to the reduction in CPU overhead gained by eliminating the sort phase.

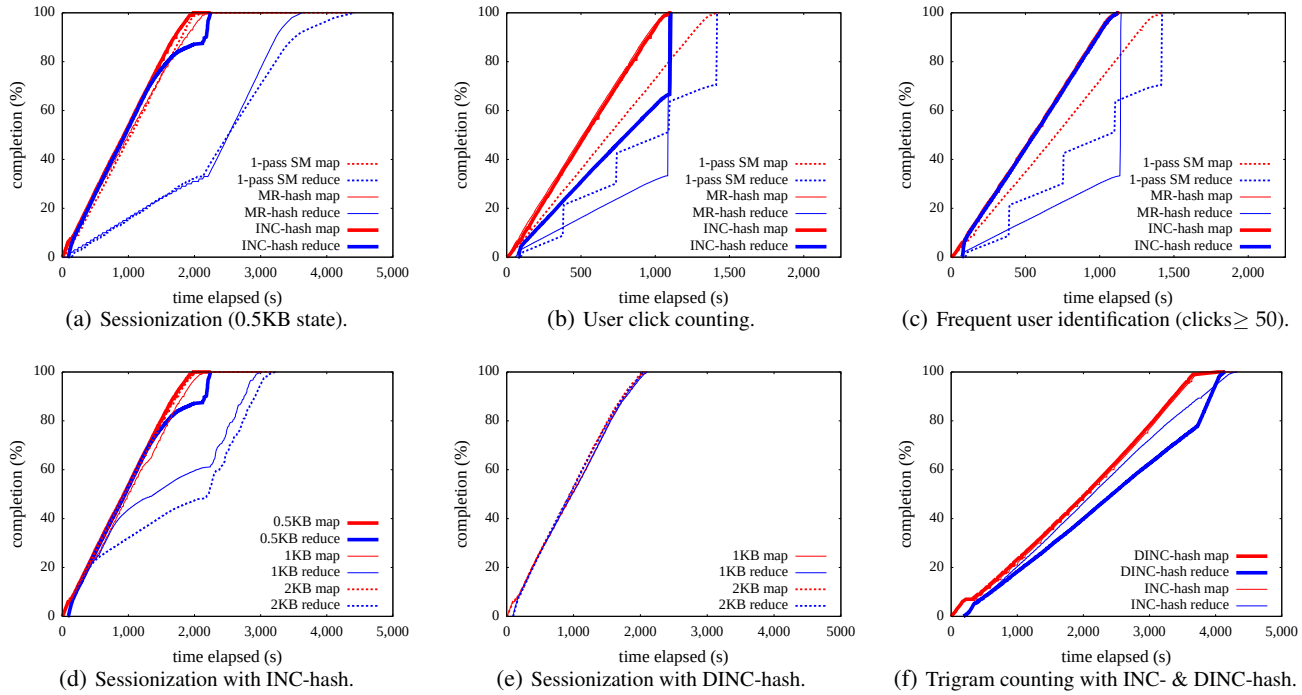
We further evaluate MR-hash and INC-hash with frequent user identification. This query is based on user click counting, but allows a user to be output whenever the counter of the user reaches 50. Fig. 9(c) shows 1-pass SM and MR-hash perform similarly as in user click counting, as the reduce function cannot be applied until map tasks finish. The reduce progress of INC-hash completely keeps up with the map progress due to the ability to output early.

In summary, given sufficient memory, INC-hash performs fully in-memory incremental processing, due to which, its reducer progress can potentially keep up with the map progress for queries that allow early output. Hash techniques can run faster if I/O and network are not bottlenecks due to the elimination of sorting.

7.2 Large Key-state Space

We next evaluate INC-hash and DINC-hash for incremental processing for workloads with a large key-state space, which can trigger substantial I/O. Our evaluation uses two workloads below:

[9] http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm



(a) Sessionization (0.5KB state). (b) User click counting. (c) Frequent user identification (clicks≥ 50).

(d) Sessionization with INC-hash. (e) Sessionization with DINC-hash. (f) Trigram counting with INC- & DINC-hash.

Figure 9: Progress report using hash implementations.

Table 4: Comparing optimized Hadoop (using sort-merge), MR-hash, and INC-hash .

Sessionization	1-Pass SM	MR-hash	INC-hash
Running time (s)	4424	3577	2258
Map CPU time per node (s)	936	566	571
Reduce CPU time per node (s)	1104	1033	565
Map output / Shuffle (GB)	245	245	245
Reduce spill (GB)	250	256	51
User click counting	1-Pass SM	MR-hash	INC-hash
Running time (s)	1430	1100	1113
Map CPU time per node (s)	853	444	443
Reduce CPU time per node (s)	39	41	35
Map output / Shuffle (GB)	2.5	2.5	2.5
Reduce spill (GB)	1.1	0	0
Frequent user identification	1-Pass SM	MR-hash	INC-hash
Running time (s)	1435	1153	1135
Map CPU time per node (s)	855	442	441
Reduce CPU time per node (s)	38	38	34
Map output / Shuffle (GB)	2.5	2.5	2.5
Reduce spill (GB)	1.1	0	0

Table 5: Comparing sessionization to INC-hash with 0.5KB state, INC-hash with 2KB state, and DINC-hash with 2KB state.

	INC (0.5KB)	INC (2KB)	DINC (2KB)
Running time (s)	2258	3271	2067
Reduce spill (GB)	51	203	0.1

Sessionization with varying state size. Fig. 9(d) shows the map and reduce progress under three state sizes: 0.5KB, 1KB, and 2KB. A larger state size implies the reduce memory can hold few states and that the reduce progress diverges earlier from the map progress. Also, larger states cause more data to be spilled to disk, as shown in Table 5. Consequently, after map tasks finish, the time for processing data from disk is longer. To enable DINC-hash for sessionization, we evict a state from memory if: (1) all the clicks in the state belong to an expired session; (2) the counter of the state is zero. Rather than spilling the evicted state to disk, the clicks in it can be directly output. As shown in Table 5, DINC-hash only spills 0.1 GB data in reduce with 2KB state size, in contrast to 203 GB for the same workload in INC-hash. As shown in Fig. 9(e), the reduce progress of DINC-hash closely follows the map progress, and spends little time processing the on-disk data after mappers finish.

We further quote numbers about stock Hadoop for this workload (see Table 1). Using DINC-hash, the reducers output continuously and finish as soon as all mappers finish reading the data in 34.5 minutes, with 0.1GB internal spill. In contrast, the original Hadoop system returns all the results towards the end of the 81 minute job, causing 370GB internal data spill to disk, 3 orders of magnitude more than DINC-hash.

Trigram Counting. Fig. 9(f) shows the map and reduce progress plot for INC-hash and DINC-hash. The reduce progress in both keeps growing below, but close to the map progress, with DINC-hash finishing a bit faster. In this workload, the reduce memory can only hold 1/30 of the states, but less than half of the input data is spilled to disk in both approaches. This implies that both hash techniques hold a large portion of hot keys in memory. DINC-hash does not outperform INC-hash like with sessionization because the trigrams are distributed more evenly than the user ids, so most hot trigrams appear before the reduce memory fills up. INC-hash naturally holds them in memory. The reduce progress in DINC-hash falls slightly behind that of INC-hash because if the state of a key is evicted, and the key later gets into memory again, the counter in its state starts from zero again, making it harder for a key to reach the

threshold of 1,000. Both hash techniques finish the job in the range of 4,100-4,400 seconds. In contrast, 1-pass SM takes 9,023 seconds. So both hash techniques outperform Hadoop.

In summary, results in this section show that our hash techniques significantly improve the progress of the map tasks, due to the elimination of sorting, and given sufficient memory, enable fast in-memory processing of the reduce function. For workloads that require a large key-state space, our frequent-key mechanism significantly reduces I/Os and enables the reduce progress to keep up with the map progress, thereby realizing incremental processing.

8. RELATED WORK

Query Processing using MapReduce [4, 10, 15, 16, 18, 22] has been a research topic of significant interest lately. To the best of our knowledge, none of these systems support incremental one-pass analytics as defined in our work. The closest work to ours is MapReduce Online [5] which we discussed in detail in Sections 2 and 3. Dryad [22] uses in-memory hashing to implement MapReduce group-by but falls back on the sort-merge implementation when the data size exceeds memory. Merge Reduce Merge [21] implements hash join using a technique similar to our baseline MR-hash, but lacks further implementation details. Several other projects are in parallel to our work: The work in [2] focuses on optimizing Hadoop parameters and ParaTimer [12] aims to provide an indicator of remaining time of MapReduce jobs. Neither of them improves MapReduce for incremental computation. Finally, many of the above systems support concurrent MapReduce jobs to increase system resource utilization. However, the resources consumed by each task will not reduce, and concurrency does not help achieve one-pass incremental processing.

Parallel Databases: Parallel databases [8, 7] require special hardware and lacked sufficient solutions to fault tolerance, hence having limited scalability. Their implementations use hashing intensively. In contrast, our work leverages the massive parallelism of MapReduce and extends it to incremental one-pass analytics. We use MR-hash, a technique similar to hybrid hash used in parallel databases [8], as a baseline. Our more advanced hash techniques emphasize incremental processing and in-memory processing for hot keys in order to support parallel stream processing.

Distributed Stream Processing has considered a distributed federation of participating nodes in different administrative domains [1] and the routing of tuples between nodes [19], without using MapReduce. Our work differs from these techniques as it considers the new MapReduce model for massive partitioned parallelism and extends it to incremental one-pass processing, which can be later used to support stream processing.

Parallel Stream Processing: The systems community has developed parallel stream systems like System S [23] and S4 [14]. These systems 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 related issues in the system.

9. CONCLUSIONS

In this paper, we examined the architectural design changes that are necessary to bring the benefits of the MapReduce model to incremental one-pass analytics. Our empirical and theoretical analyses showed that the widely-used sort-merge implementation for MapReduce partitioned parallelism poses a fundamental barrier to incremental one-pass analytics, despite optimizations. We proposed a new data analysis platform that employs a purely hash-based framework, with various techniques to enable incremental processing and fast in-memory processing for frequent keys. Evaluation of our Hadoop-based prototype showed that it can significantly improve the progress of map tasks, allows the reduce progress to keep up with the map progress with up to 3 orders of magnitude reduction of internal data spills, and enables results to be returned early. In future work, we will extend our one-pass analytics platform to support a wider range of incremental computation tasks with minimized I/O, online aggregation with early approximate answers, and stream query processing with window operations.

10. REFERENCES

[1] D. J. Abadi, Y. Ahmad, et al. The design of the Borealis stream processing engine. In *CIDR*, 277–289, 2005.

[2] S. Babu. Towards automatic optimization of MapReduce programs. In *SoCC*, 137–142, 2010.

[3] R. Berinde, G. Cormode, et al. Space-optimal heavy hitters with strong error bounds. In *PODS*, 157–166, 2009.

[4] R. Chaiken, B. Jenkins, et al. Scope: easy and efficient parallel processing of massive data sets. In *PVLDB*, 1(2):1265–1276, 2008.

[5] T. Condie, N. Conway, et al. MapReduce online. In *NSDI*, 2010.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 10–10, 2004.

[7] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[8] D. J. DeWitt, R. H. Gerber, et al. Gamma: a high performance dataflow database machine. In *VLDB*, 228–237, 1986.

[9] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *SIGMOD*, 423–434, 1996.

[10] D. Jiang, B. C. Ooi, et al. The performance of MapReduce: an in-depth study. In *VLDB*, 2010.

[11] J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[12] K. Morton, M. Balazinska, et al. Paratimer: a progress indicator for MapReduce dags. In *SIGMOD*, 507–518, 2010.

[13] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2006.

[14] L. Neumeyer, B. Robbins, et al. S4: distributed stream computing platform. In *KDCloud*, 2010.

[15] C. Olston, B. Reed, et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 1099–1110, 2008.

[16] A. Pavlo, E. Paulson, et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 165–178, 2009.

[17] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[18] A. Thusoo, J. S. Sarma, et al. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[19] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 333–344, 2003.

[20] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[21] H.-c. Yang, A. Dasdan, et al. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 1029–1040, 2007.

[22] Y. Yu, P. K. Gunda, et al. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, 247–260, 2009.

[23] Q. Zou, H. Wang, et al. From a stream of relational queries to distributed stream processing. In *VLDB*, 2010.