SCALLA: A Platform for Scalable One-Pass Analytics using MapReduce

Boduo Li, University of Massachusetts Amherst
Edward Mazur, University of Massachusetts Amherst
Yanlei Diao, University of Massachusetts Amherst
Andrew McGregor, University of Massachusetts Amherst
Prashant Shenoy, University of Massachusetts Amherst

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.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Parallel processing, one-pass analytics, incremental computation

ACM Reference Format:
Li, B., Mazur, E., Diao, Y., McGregor, A., and Shenoy, P. 2012. SCALLA: A Platform for Scalable One-Pass Analytics using MapReduce. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 38 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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

Author's address: B. Li, E. Mazur, Y. Diao, A. McGregor and P. Shenoy, Department of Computer Science, University of Massachusetts Amherst.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 [Condie et al. 2010]). 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 those 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 partitioned 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 persist, and the reduce progress falls significantly behind the map progress, hence violating the requirements of efficient incremental processing.

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 the blocking behavior of sort-merge. We devise two hash techniques to suit different reduce functions, depending on whether the reduce function permits incremental processing or not. Besides eliminating the sorting cost from the map tasks, these hash techniques can provide 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 both deterministic and randomized techniques to dynamically recognize popular 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 popular 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, we obtain the following main results: (1) 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. (2) For challenging workloads that require a large key-state space, our dynamic hashing 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. (3) Further trade-offs exist between our hash-based techniques under different workload types, data localities, and memory sizes, with dynamic hashing working the best under constrained memory and most workloads.

2. BACKGROUND

To provide a technical context for the discussion in this paper, we begin with background on MapReduce systems and summarize the key results of our recent benchmarking study.

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 scheduling, 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 [Olston et al. 2008]) 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 [White 2009] to implement the group-by functionality for parallel processing (Google's MapReduce system is reported to use a similar implementation [Dean and Ghemawat 2004], 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.

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 ask a centralized service for completed mappers, and once notified, request 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.

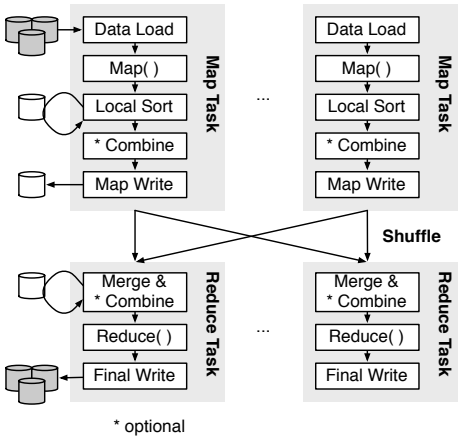


Fig. 1. Architecture of the Hadoop implementation of MapReduce.

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 [Condie et al. 2010]. 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.

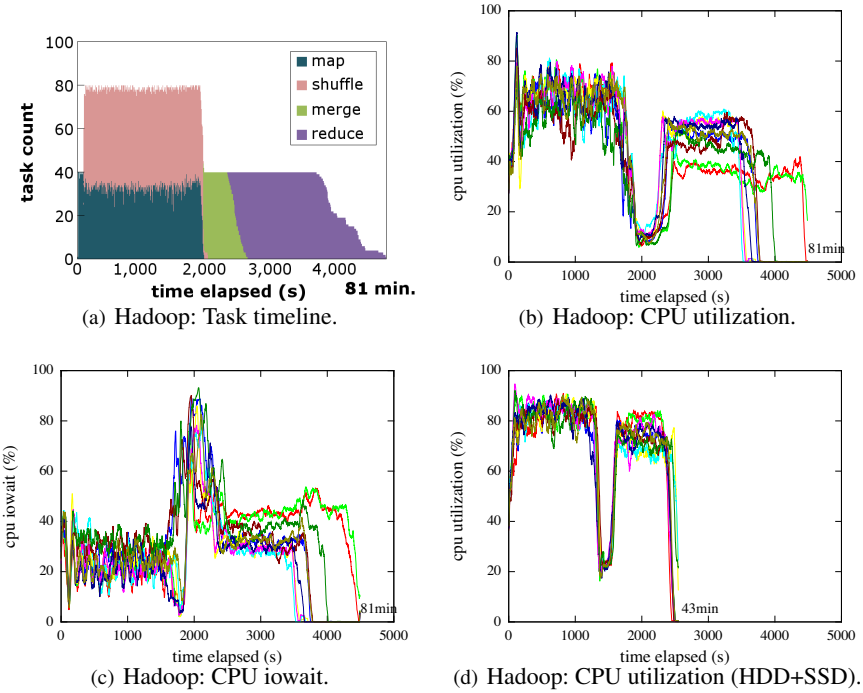
2.3. Summary of Benchmarking Results

The requirements for scalable streaming analytics—*incremental processing* and *fast in-memory processing* whenever possible—require the MapReduce program of a query to be non-blocking and have low CPU and I/O overheads. In our recent benchmarking study [Mazur et al. 2011], we examined whether current MapReduce systems meet these requirements. We considered applications such as click stream analysis and web document analysis in our benchmark. In the interest of space, we mainly report results on click stream analysis in this section.

Given a click stream, an important task is sessionization that reorders page clicks into individual user sessions. In its MapReduce program, the map function extracts the user id from each click and groups the clicks by user id. The reduce function arranges the clicks of each user by timestamp, streams out the clicks of the current session, and closes the session if the user has had no activity in the past 5 minutes. A large amount of intermediate data occurs in this task due to the reorganization of all the clicks by user id. Other click analysis tasks include counting the number of visits to each

Table I. Workloads in click analysis and Hadoop running time.

Metric	*Sessionization*	*Page frequency*	*Clicks per user*
Input	256GB	508GB	256GB
Map output	269GB	1.8GB	2.6 GB
Reduce spill	370GB	0.2GB	1.4 GB
Reduce output	256GB	0.02GB	0.6GB
Running time	4860 sec	2400 sec	1440 sec



(a) Hadoop: Task timeline.
(b) Hadoop: CPU utilization.
(c) Hadoop: CPU iowait.
(d) Hadoop: CPU utilization (HDD+SSD).

Fig. 2. Experimental results using the sessionization workload.

url and counting the number of clicks that each user has made. For these problems, using a combine function can significantly reduce intermediate data sizes. Our study used the click log from the World Cup 1998 website[1] and replicated it to larger sizes as needed.

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 settings 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.

Table I shows the running time of the workloads as well as the sizes of input, output, and intermediate data in click stream analysis. Fig. 2(a) shows the task timeline for the sessionization workload, i.e., the number of tasks for the four main operations: *map* (including sorting), *shuffle*, *merge* (the multi-pass part), and *reduce* (including the final merge to produce a single sorted run). In this task, time is roughly evenly split between the map and reduce phases. A key observation is that the CPU utilization, as shown in Fig. 2(b), is low in an extended period (from time 1,800 to 2,400) after all mappers have finished. The CPU iowait in Fig. 2(c) shows that this is largely due to disk I/O

[1]World Cup 1998: http://ita.ee.lbl.gov/html/contrib/WorldCup.html

Table II. 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
T	Time measurement for startup and I/O cost
h	Height of the tree structure for multi-pass merge

requests in multi-pass merge, as further verified by the number of bytes read from disk in that period. Multi-pass merge is not only I/O intensive but also blocking—the reduce function cannot be applied until all the data has been merged into a sorted run. To reduce disk contention, we used additional hardware so that the disk handled only the input and output with HDFS, and all the intermediate data was passed to a fast SSD. As Fig. 2(d) shows, such change can reduce overall running time but does not eliminate the I/O bottleneck or blocking incurred in the multi-pass merge.

Another main observation regards the CPU cost. We observe from Fig. 2(b) that CPUs are busy in the map phase. However, the map function in the sessionization workload is CPU light: it simply extracts the user id from each click record and emits a key-value pair where the value contains the rest of the record. The rest of the cost in the map phase is attributed to sorting of the map output.

In simpler workloads, such as counting the number of clicks per user, there is an effective combine function to reduce the size of intermediate data. As intermediate data is reduced, the merge phase shrinks as there is less data to merge, and then the reduce phase also shrinks as most data is processed in memory only. However, the overhead of sorting becomes more dominant in the overall cost.

In summary, our benchmarking study made several key observations of the sort-merge implementation of MapReduce group-by:

- The sorting step of sort-merge incurs high CPU cost, hence not suitable 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 a poor fit for incremental processing or fast in-memory processing.
- Using extra storage devices (including fast solid state drives) and alternative storage architectures does not eliminate blocking or the I/O bottleneck.

3. 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.

3.1. An Analytical Model for Hadoop

The Hadoop system has a large number of parameters. While our previous experiments used the default setting, 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 II. 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 ⟨key, value⟩ 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.

3.1.1. Modeling I/O Cost in Bytes. We analyze the I/O cost of the *existing* sort-merge implementation of Hadoop. We first consider the I/O cost in terms of the number of bytes read and written. Our main result is summarized in the following proposition.

PROPOSITION 3.1. *Given the workload description (D, K_m, K_r) and the hardware description (N, B_m, B_r), as defined in Table II, the I/O 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:

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

PROOF. Our analysis includes five I/O-types listed in Table II. 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 of 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. 3 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 all initial runs are merged, a final merge combines all the remaining files (there are at most $2F - 1$ of them).

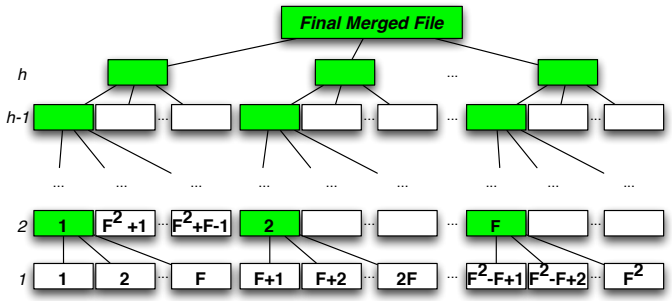


Fig. 3. Analysis of the tree of files created in multi-pass merge.

For the analysis, let α_i denote the size of a merged file on level i ($2 \leq i \leq h$) 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. □

3.1.2. Modeling the Number of I/O requests. In our analysis we also model the number of I/O requests in a Hadoop job, which allows us to estimate the disk seek time when these I/O requests are performed as random I/O operations. Again we summarize our result in the following proposition.

PROPOSITION 3.2. *Given the workload description (D, K_m, K_r) and the hardware description (N, B_m, B_r), as defined in Table II, 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.

PROOF. We again consider the five types of I/O listed in Table II. 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 not only on the data size but also on 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}. Then 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, which is:

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

Revisit the tree of files in multi-pass merge in Fig. 3. Each merge step, numbered j in the formula below, 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:

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

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. We provide detailed empirical evidence in Section 3.2.

3.1.3. Modeling the Startup Cost. We further consider the cost of starting map and reduce tasks as it has been reported to be a nontrivial cost [Pavlo et al. 2009]. Since the number of map tasks is usually much larger than that of reduce tasks, we mainly 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 $c_{start} \cdot \frac{D}{CN}$.

3.1.4. 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:

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

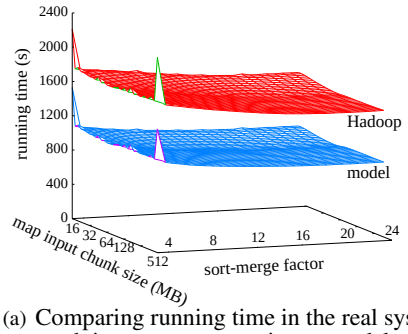
The above formula is our complete analytical model that captures the effects of all of the involved parameters.

3.2. Optimizing Hadoop based on our Analytical Model

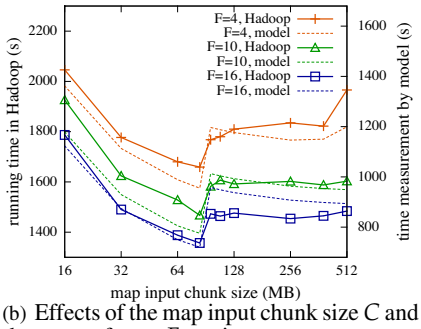
Our analytical model enables us to predict system behaviors as Hadoop parameters vary. Then, given a workload and system configuration, we can choose values of these parameters that minimize the time cost in our model, thereby optimizing Hadoop performance.

3.2.1. 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;[2] (2) Hardware: N=10, B_m=140MB, B_r=260MB; (3) Hadoop: R=4

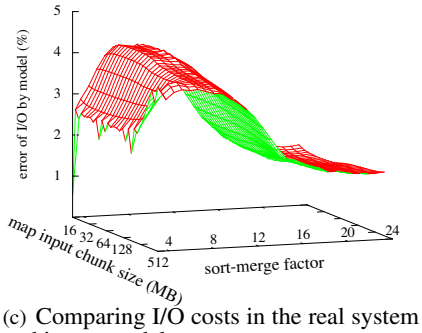
[2]We used a smaller dataset in this set of experiments compared to the benchmark because changing Hadoop configurations often required reloading data into HDFS, which was very time-consuming.



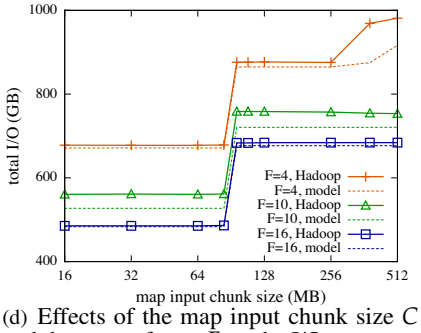
(a) Comparing running time in the real system and time measurement in our model



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



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



(d) Effects of the map input chunk size C and the merge factor F on the I/O cost

Fig. 4. Validating our model against actual measurements in a Hadoop cluster.

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. 4(a).[3] 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 setting. Fig. 4(a) 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. 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. As shown in Fig. 4(c), the differences between the predicted numbers and the actual numbers are mostly within 5%. Here the errors are mainly due to the fact that our analysis assumes the multi-pass merge tree to be full but this is not always true in practice.

Our next goal is to show how to optimize the parameters based on our model. To reveal more details from the 3-D plots, we show the results of a smaller range of (C, F) in Fig. 4(b) and Fig. 4(d)

[3]For both the real running time and modeled time cost, the respective 100 data points were interpolated into a finer-grained mesh.

where the solid lines are for the actual measurements from the Hadoop cluster and the dashed lines are for predication using our model.

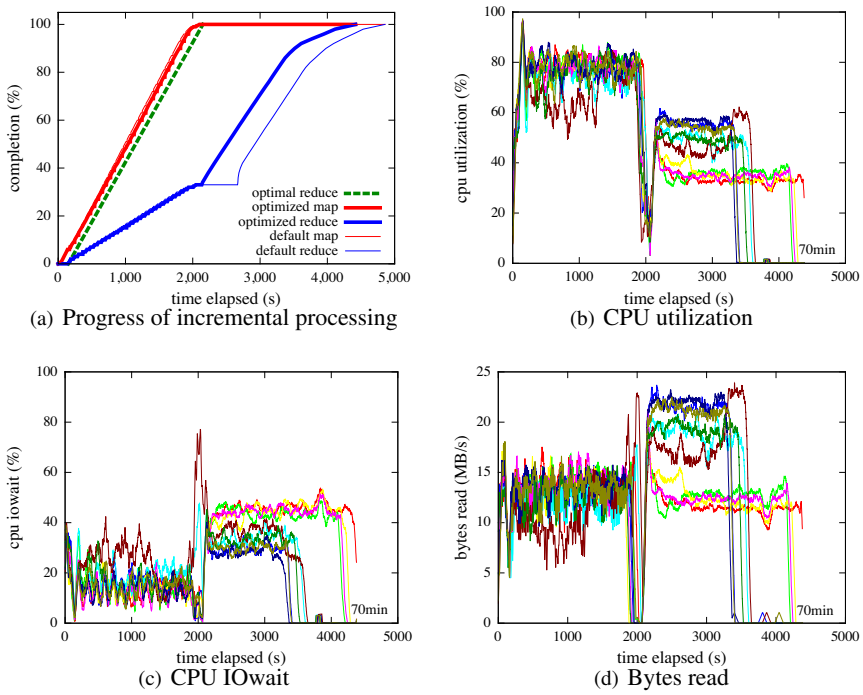
(1) Optimizing the Chunk Size. When the chunk size C is very small, the MapReduce job uses many map tasks and the map startup cost dominates in the total time cost. As C increases, the map startup cost reduces. However, once the map output exceeds its buffer size, multi-pass merge is incurred with increased I/O cost, as shown in Fig. 4(d). As a result, 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, e.g., size 256 when F=4 shown in Fig. 4(d)), the number of passes of on-disk merge goes up, thus incurring more I/Os. The overall best performance in running time 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. 4(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 as shown in Fig. 4(d). 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 [Dean and Ghemawat 2004] 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.

In summary, the above results demonstrate two key benefits of our model: (1) Our model predicts the trends in I/O cost and time cost close to the observations in real cluster computing. (2) 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.



(a) Progress of incremental processing (b) CPU utilization

(c) CPU IOwait (d) Bytes read

Fig. 5. Performance of optimized Hadoop based on our model.

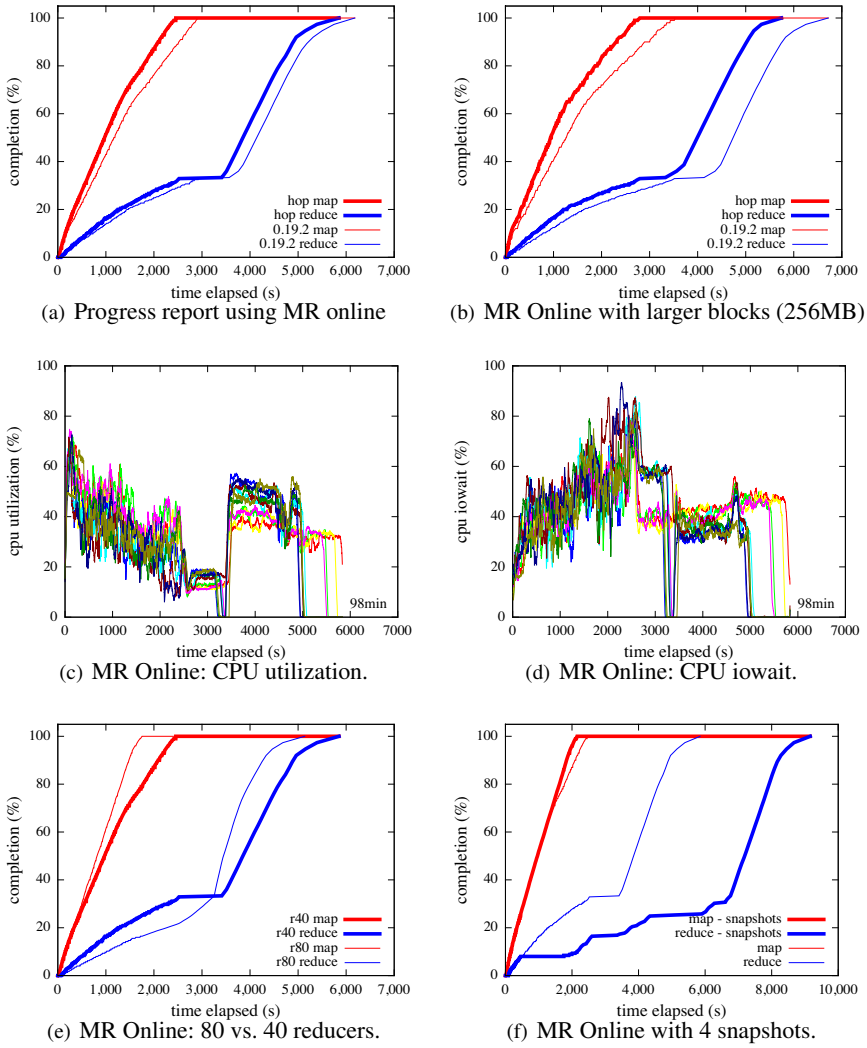
3.2.2. Analysis of Optimized Hadoop. We finally reran the 240GB sessionization workload described in our benchmark (see Section 2). We optimized Hadoop using 64MB data chunks, one-pass merge, and 4 reducers per node as suggested by the above results. The total running time was reduced from 4,860 seconds 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 3.3 (*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 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. 5(a) 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 33% in these 2,000 seconds, mainly because the shuffle progress could 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.



(a) Progress report using MR online
(b) MR Online with larger blocks (256MB)
(c) MR Online: CPU utilization.
(d) MR Online: CPU iowait.
(e) MR Online: 80 vs. 40 reducers.
(f) MR Online with 4 snapshots.

Fig. 6. Performance of MapReduce Online with pipelining of data.

Fig. 5(b), 5(c), and 5(d) further show the CPU utilization, CPU iowait, and the number of bytes read 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 and remains high in the rest of the job. This is due to the blocking of CPU by the I/O operations in the remaining single-pass merge.

3.3. Pipelining in Hadoop

Another attempt to optimize Hadoop for one-pass analytics would be to pipeline data from mappers to reducers so that reducers can start the work earlier. This idea has been implemented in MapReduce

Online [Condie et al. 2010], as described in Section 2.2. In our benchmark, we made the following observations about pipelining data from mappers to reducers in the Hadoop framework:

(1) Benefits of pipelining. We first summarize the benefits of pipelining that were observed in our benchmark. First, pipelining data from mappers to reducers can result in small performance benefits. For instance, for sessionization, Fig. 6(a) shows 5% improvement in total running time over the version of stock Hadoop, 0.19.2, on which MapReduce Online's code is based. However, the overall performance gain of MapReduce Online over Hadoop is small (e.g., 5%), less that the gain of our model-based optimization (e.g., 14%). Second, pipelining also makes the performance less sensitive to the HDFS chunk size. As Fig. 6(b) shows, when the chunk size is increased from 64MB to 256MB, the performance of MapReduce Online, denoted by the bold lines, stays about the same as that of 64MB, whereas the performance of stock Hadoop, denoted by the thin lines, degrades. This is because a larger block size places additional strain on the map output buffer and therefore increases the chance of having to spill the map output to disk in order to perform sorting. Pipelining, however, is able to mitigate this effect by eagerly sending data at a finer granularity to the reducers with the intention to use merging later to bring all the data in sorted order.

(2) Limitations of pipelining. However, we observe that adding pipelining to an overall blocking implementation based on sort-merge is not an effective mechanism for incremental processing. Most importantly, the reduce progress lags far behind the map progress, as shown in Fig. 6(a) and 6(b). To explain this behavior, we observe from Fig. 6(c) that the CPU utilization still has low values in the middle of the job and is on the average 50% or below over the entire job. While CPU can be idle due to I/O wait or network wait (given the different communication model used), the CPU iowait in Fig. 6(d) again shows a spike in the middle of the job and overall high values during the job. Hence, the problems with blocking and intensive I/O due to multi-pass merge still exist.

(3) Effect of adding reducers. We further investigate whether using more reducers can help close the gap between the map progress and reduce progress. It is important to note that pipelining does not reduce the total amount of work in sort-merge but rather rebalances the work between the mappers and reducers. More specifically, eager transmission of data from mappers to reducers reduces the sorting work in the mappers but increases the merge work in the reducers. To handle more merge work, increasing the number of reducers helps improve the overall running time, as shown in Fig. 6(e) where the number of reducers is increased from 4 per node to 8 per node. However, the reduction of the running time comes at the cost of significantly increased resource consumption and the gap between the map progress and the reduce progress is not reduced much. Moreover, further increasing the number of reducers, e.g., to 12 per node, starts to perform worse due to the drawbacks of using multiple waves of reducers mentioned in Section 3.2.1.

(4) Effect of using snapshots. Finally, MapReduce Online has an extension to periodically output snapshots (e.g., when reducers have received 25%, 50%, 75%, ..., of the data). However, this is done by repeating the merge operation for each snapshot, not by incremental in-memory processing. As a result, the simple snapshot-based mechanism can incur high I/O overheads and significantly increased running time, as shown in Fig. 6(f).

Summary. 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) Due to the above reason, the reduce progress falls far behind the map progress. (3) The map tasks still have the high CPU cost of sorting.
- Pipelining from mappers to reducers does not resolve the blocking or I/O overhead in Hadoop, hence not an effective mechanism for providing fast incremental processing.

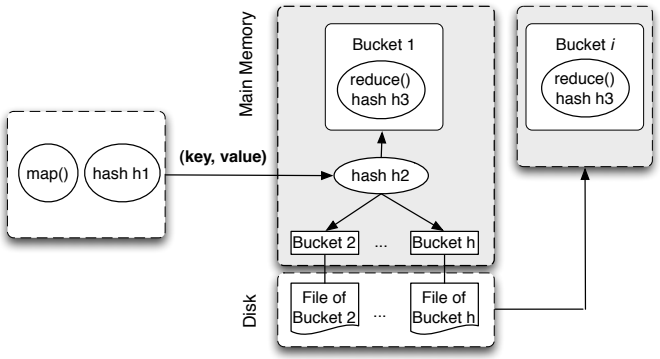


Fig. 7. MR-hash: hashing in mappers and two phase hash processing in reducers.

4. 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 aims to transform MapReduce computation into incremental one-pass processing. To build the new platform, our first mechanism is to devise a hash-based alternative to the widely used sort-merge implementation for partitioned parallel processing, with the goal to minimize computational and I/O bottlenecks as well as blocking. The hash implementation can be particularly useful when analytical tasks do not require the output of the reduce function to be sorted across different keys.[4] More specifically, we design two hash techniques for two different types of reduce functions, respectively, which we describe in Section 4.1 and Section 4.2, respectively. These techniques enable fast in-memory processing when there is sufficient memory for a given workload. In addition, our second mechanism brings the benefits of such fast in-memory processing further to workloads that require a large key-state space far exceeding available memory. Our techniques dynamically identify popular keys and update their states using a full in-memory processing path. These dynamic techniques are described in Section 4.3. In Section 4.4, we discuss the optimization of key Hadoop system parameters in our hash-based framework.

4.1. A Basic Hash Technique (MR-hash)

Recall from Section 2 that to support parallel processing, the MapReduce computation model 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, ..., across the mappers and reducers.

As depicted in Fig. 7, the hash function h_1 partitions the map output into subsets corresponding to the scheduled reducers. Hash functions h_2, h_3, ..., are used to implement group-by at each reducer. We adopt the hybrid-hash algorithm [Shapiro 1986] from parallel databases as follows: 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. This way, we can perform group-by on D_1 using the hash function h_3 and apply the reduce function to each group 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.

[4]Our implementation offers a knob for a MapReduce job to be configured with either the hash implementation or the sort-merge implementation. When our platform is used to build a query processor on top of MapReduce, if both sort-merge and hashing algorithms are available for implementing an operator like join, our system will enable the query processor to quickly implement the hash algorithm of choice by utilizing the internal hashing functionality of our MapReduce platform.

Otherwise, we recursively partition D_i using hash function h_4, and so on. In our implementation, we use standard universal hashing to construct a series of independent hash functions.

Following the analysis of the hybrid hash join [Shapiro 1986], 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 $2\sqrt{|D|}$, where $|D|$ is the number of pages of data sent to the reducer, and the I/O cost is $2(|D| - |D_1|)$ pages of data 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 [DeWitt et al. 1990], 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, this hash implementation offers a step towards incremental processing: It allows answers for the first bucket, S_0, to be returned from memory after all the data arrives, and answers for other buckets to be returned one bucket at a time. In contrast, sort-merge cannot return any answer until all the data is sorted. However, such incremental processing is very coarse-grained, as a bucket can contain a large chunk of data. Moreover, in many cases the total I/O is not significantly better than the I/O of sort-merge [Ramakrishnan and Gehrke 2003].

4.2. An Incremental Hash Technique (INC-hash)

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 [Muthukrishnan 2006]. For such reduce functions, we propose a more efficient hash algorithm, called incremental hash (INC-hash).

Algorithm. The algorithm is illustrated in Fig. 8 (the reader can ignore the darkened boxes for now, as they are used only in the third technique). In Phase 1, as a reducer receives map output, called tuples for simplicity, we build a hash table H (using hash function h_2) that maps from a key to the *state* of computation for all the tuples of that key that have been seen so far. When a new tuple arrives, if its key already exists in H, we update the key's state using the new tuple. 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, and flush the write buffer when it becomes full (similar to Hybrid Cache [Hellerstein and Naughton 1996] in this step). At the end of Phase 1, the reducer has seen all the tuples and returned final answers for all the keys in H. Then in Phase 2, it reads disk-resident buckets back one at a time, repeating the procedure above to process each bucket. If the key-state pairs produced from a specific bucket fit in memory, no further I/O will be incurred. Otherwise, the algorithm will again process some keys in memory and write the tuples of other keys to disk-resident buckets, i.e., applying recursive hashing.

INC-hash offers two major advantages over MR-hash: (1) *Reduced data volume and I/O*: For those keys held in memory, their tuples are continuously collapsed into states in memory, hence avoiding I/O's for those tuples altogether. I/O's can be completely avoided in INC-hash if the memory is large enough to hold all *key-state* pairs, in contrast to all the data in MR-hash. (2) *Earlier results*: For those keys held in memory, query answers can be returned before all the data is seen. In particular, earlier results are possible for filter queries (e.g., when the count of a URL exceeds a threshold), join queries (whose results can be pipelined out), and window queries (whose results can be output whenever a window closes).

Partial Aggregation. An opportunity for further optimization is that some reduce functions that permit incremental processing are also amenable to partial aggregation, which splits incremental processing to a series of steps on the processing pipeline with successively reduced data volume. Classical examples are the aggregates sum, $count$, and avg. To support partial aggregation in the MapReduce context, we define three primitive functions:

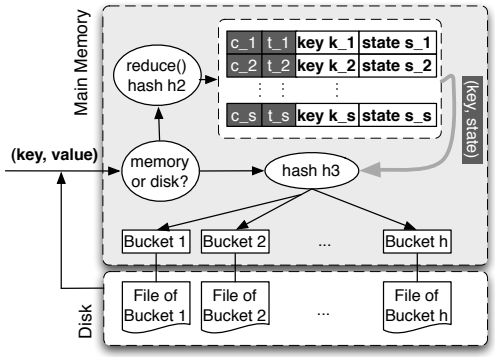


Fig. 8. (Dynamic) Incremental Hash: monitoring keys and updating states.

- The initialize function, $init()$, reduces a sequence of data items of the same key to a state;
- The combine function, $cb()$, reduces a sequence of states of the same key to a new state;
- 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 updating a key-state pair in the hash table with a new data item (which is also a key-state pair) or in packing multiple data items in the write buffer of a bucket. Finally, the original reduce function is implemented by $cb()$ followed by $fn()$.

Partial aggregation provides several obvious benefits: The initialize function reduces the amount of data output from the mappers, thereby reducing the communication cost in shuffling and the CPU and I/O costs of subsequent processing at the reducers. In addition, the reducers can apply $cb()$ in all suitable places to collapse data aggressively into compact states, hence reducing the I/O cost.

In implementation, the INC-hash algorithm is applied to "group data by key" in both $init()$ and $cb()$. The operation within each group, in both $init()$ and $cb()$, is very similar to the user-specified reduce function, as in the original proposal of combiner functions [Dean and Ghemawat 2004].

Memory and I/O Analysis. We next analyze the INC-hash algorithm for its memory requirements and I/O cost. Let D be the size of data input to the algorithm (e.g., data sent to a reducer), U be the size of the key-state space produced from the data, and B be the memory size, all in terms of the number of pages covered. Let h be the number of buckets created in the INC-hash algorithm. In Phase 1, we need 1 page for the input data and h pages for write buffers of the disk-resident buckets. So, the size of the hash table is $B-h-1 \geq 0$. Assume that the key-state pairs not covered by the hash table, whose size is $U-(B-h-1)$, are evenly distributed across h buckets. To make sure that the key-state pairs produced from each bucket fit in memory in Phase 2, the following equality has to hold: $U-(B-h-1) \leq h \cdot (B-1)$. Rewriting both constraints, we have:

$$\frac{U-1}{B-2} - 1 \leq h \leq B-1. \tag{6}$$

The above analysis of memory requirements has several implications:

- When the memory size B reaches U+1, all the data is processed in memory, i.e., $h=0$.
- When B is in the range $[\sqrt{U}+1, U]$, given a particular value of B, the number of buckets h can be chosen between the lower bound $(U-1)/(B-2)-1$ and the upper bound $B-1$, as marked by the shaded area in Fig. 9.
- Also for the above range of B, under the assumption of uniform distribution of keys in the data, no recursive partitioning is needed in INC-hash: those tuples that belong to the in-memory hash

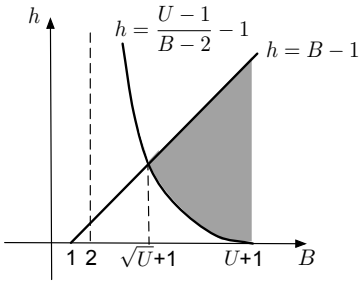


Fig. 9. Determining the number of buckets h given memory size B and key-state space size U.

table are simply collapsed into the states, and other tuples are written out and read back exactly once. In this case, the fraction of the data that is not collapsed into the in-memory hash table is $(U-(B-h-1))/U$. So the I/O cost of INC-hash is:

$$2 \cdot (1 - \frac{B-h-1}{U}) \cdot D. \tag{7}$$

- To minimize I/O according to the above formula, we want to set the number of buckets h to be its lower bound $(U-1)/(B-2)-1$.

We note that for common workloads, the memory size is expected to exceed $\sqrt{U}$. All modern computers with several GB's of memory can meet this requirement for any practical value of U.

Sensitivity to Parameters. The above analysis reveals that the optimal I/O performance of INC-hash requires setting the right number of buckets, h, which depends on the size of the key-state space, U. This issue is less a concern in traditional databases because the DBMS can collect detailed statistics from stored data and size estimation for relational operators has been well studied. For our problem of scalable, incremental analytics, the lack of knowledge of the key space often arises when data is streamed over the wire or from upstream complex operators (coded in user-defined functions) in the processing pipeline. Consider the performance loss when we do not know the key space size. According to Eq. 6, without knowing U we do not know the lower bound of h and hence the only safe value to choose would be the upper bound $B-1$. Given this worst choice of h, we pay the following extra I/O according to Eq. 7:

$$\left(\frac{(B-1)^2}{U} - 1\right) \cdot \frac{D}{B-2}.$$

For example, when $U = 20$GB and $B = 10$GB, the extra I/O paid amounts to the data size D, which can be quite large.

To mitigate such performance loss, we would like to acquire an accurate estimate of the key-state space size U. Since many workloads use fixed-sized states, estimating U is equivalent to estimating the number of distinct keys in the data set. In today's analytics workloads, the key space can be very large, e.g., tens of billions of URLs on the Web. So estimating the number of keys can itself consume a lot of memory. Hence, we propose to perform approximate estimation of the key space size using fast, memory-efficient "mini-analysis": state-of-the-art sketch techniques [Kane et al. 2010] can provide $1+\epsilon$ approximation for the number of distinct keys in space about $O(\epsilon^{-2})$ with low CPU overheads. Hence, given memory of modest size, these techniques can return fairly accurate approximations. Such "mini-analysis" can be applied in two ways: If the data is to be first loaded into the processing backend and later analyzed repeatedly, mini-analysis can be piggybacked in data loading with little extra overhead. For streaming workloads, such mini-analysis can be run periodically at data sources and the resulting statistics can be transferred to the MapReduce processing backend to better

configure our hash algorithms. A detailed mechanism for communication between data sources and the processing backend is beyond the scope of this paper and will be addressed in our future work.

4.3. A Dynamic Incremental Hash Technique (DINC-hash)

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 a widely used data stream algorithm called the FREQUENT algorithm [Misra and Gries 1982; Berinde et al. 2009] 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, such as Count-Sketch [Charikar et al. 2004], Count-Min [Cormode and Muthukrishnan 2005] and CR-Precis [Ganguly and Majumder 2007], 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 propose to perform dynamic incremental hash using a caching mechanism governed by the FREQUENT algorithm, i.e., to determine which tuples should be processed in memory and which should be written to disk. 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-data item 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 `up` be an update function that collapses an data item v into a state u to make a new state, $\texttt{up}(u,v)$.

Fig. 8 illustrates the DINC-hash algorithm. 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, the number of keys that can be processed in-memory is $s = (B-h)n_p$.[5]

The sketch of our algorithm is shown in Algorithm 1. The algorithm maintains s counters $c[1],\ldots,c[s]$, s associated keys $k[1],\ldots,k[s]$ referred to as "the keys currently being monitored", and the state $s[i]$ of a partial computation for each key $k[i]$. Initially, all the counters are set to 0, and all the keys are marked as empty (Line 1). When a new tuple (k,v) arrives, if this key is currently being monitored, the corresponding counter is incremented and the state is updated using the update function (Line 4). 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 k starts to be monitored by $c[j]$ (Line 7-8). 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 (Line 10-11). 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.

[5]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.

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.

To compare the different hash techniques, first note that the improvement of INC-hash over the baseline MR-hash is only significant when the total number of keys K is small. Otherwise, the keys processed incrementally in main memory will only account for a small fraction of the tuples, hence limited performance benefits. DINC-hash mitigates this problem 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.

Algorithm 1 Sketch of the **DINC-hash** algorithm

```
1: c[i] ← 0, k[i] ← ⊥ for all i ∈ {1, 2, ··· , s}
2: for each tuple (k, v) from input do
3:    if k is being monitored then
4:       Suppose k is monitored by c[j], do c[j] ← c[j] + 1, and s[j] ← update(s[j], v)
5:    else
6:       if ∃j such that c[j] = 0 then
7:          Evict key-state pair (k[j], s[j]) to disk
8:          c[j] ← 1, k[j] ← k, and s[j] ← v
9:       else
10:         Write tuple (k, v) to disk
11:         c[i] ← c[i] − 1 for all i ∈ {1, 2, ··· , s}
12:      end if
13:   end if
14: end for
```

We note that besides the FREQUENT algorithm, our DINC-hash technique can also be built on a closely related variant called the Space-Saving algorithm [Metwally et al. 2005]. Like the FREQUENT algorithm, Space-Saving monitors frequent items in memory and for each monitored item, maintains a counter as an estimate of the item's frequency. When the two algorithms are used as the caching mechanism in DINC-hash, they perform the same in the following two cases:

Case I: If a monitored key is received, both algorithms increment the associated counter by 1.

Case II: If an unmonitored key is received and there is at least one counter of value 0 under the FREQUENT algorithm, both algorithms evict a monitored key-state pair with the smallest counter (which must be 0 in the FREQUENT algorithm), monitor the new key-state pair with that counter, and increment the counter by 1.

The two algorithms only differ in the following case:

Case III: The arriving key is not monitored and all the counters of the monitored keys are greater than 0 in the FREQUENT algorithm. In this case, Space-Saving evicts a key-state pair with the smallest counter to make room for the new key-state pair as in Case II, whereas the FREQUENT algorithm does not take in the new key-state pair but simply decrements all counters by 1.

In most real-world workloads, the distribution of the frequencies of keys is naturally skewed. Consequently, in the FREQUENT algorithm when there is no key with counter 0, it is likely that a large number of infrequent keys have counter 1 as they appear only once in a long period of time. When Case III occurs, the counters of these infrequent keys are reduced to 0, which prevents Case III from happening again in the near future. As a result, Case III occurs infrequently under

real-world workloads, and the two algorithms perform the same most of the time. We will show that the difference in performance of the two algorithms is less than 1% in the evaluation section. Due to this reason, we focus our discussion in the rest of the section on the widely used FREQUENT algorithm.

4.3.1. Analysis of the DINC Algorithm. We next provide a detailed analysis of the DINC algorithm.

I/O Analysis. Suppose that there are f_i tuples with key k_i. Then we have the total number of tuples $M = \sum_i f_i$. Without loss of generality assume $f_1 \geq f_2 \geq \ldots \geq f_K$. Since the memory can hold s keys, the best we can hope for is to process $\sum_{1 \leq i \leq s} f_i$ arriving tuples in memory, i.e., to collapse them into in-memory states as they are 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 setting to show that using the above strategy, at least M' tuples can be collapsed into states, where

$$M' = \sum_{1 \leq i \leq s} \max\left(0, f_i - \frac{M}{s+1}\right)$$

If the data is highly skewed, our theoretical analysis can be improved by appealing to a result of [Berinde et al. 2009]. Specifically if the data is distributed with Zipfian parameter α our strategy guarantees that at least

$$M' = \sum_{1 \leq i \leq s} \max\left(0, f_i - \frac{M}{\max(s+1, (s/2)^\alpha)}\right)$$

tuples have been collapsed into states. Since every tuple that is not collapsed into an existing state in memory triggers a write-out, the number of tuples written to disk is $M - s - M' + s$, where the first s in the formula corresponds to the fact that the first s keys do not trigger a write-out, and the second s comes from the write out of the hot key-state pairs in main memory. Hence, the upper bound of the number of tuples that trigger I/O is $M - M'$.

This result compares favorably with the offline optimal if there are some very popular keys, but does not give any guarantee for non-skewed data if there are no keys whose relative frequency is more than $1/(s+1)$. For example, suppose that the $r = \epsilon s$ most popular keys have total frequency $\sum_{i=1}^{r} f_i \geq (1-\epsilon)M$, i.e., a $(1-\epsilon)$ fraction of the tuples have one of r keys. Then we can conclude that

$$M' = \sum_{1 \leq i \leq s} \max\left(0, f_i - \frac{M}{s+1}\right) \geq \sum_{1 \leq i \leq r} f_i - \frac{M}{s+1} \geq (1-2\epsilon)M$$

i.e., all but a 2ϵ fraction of the tuples are collapsed into states.

Note that even if we assume the data is skewed, in INC-hash there is no guarantee on the number of tuples processed in-memory 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.

Sensitivity to Parameters. We further investigate whether the parameters used in the DINC-hash algorithm can affect its performance. By analyzing the memory requirements as for the INC-hash algorithm, we can obtain the following constraints:

$$\frac{U}{B-1} \leq h \leq B-1, \text{ for } B \in [\sqrt{U}+1, U]$$

where U is the total size of key-state pairs, B is the memory size, and h is the number of buckets created when evicting key-state pairs from memory to disk and writing tuples of unmonitored keys to disk. Intuitively, we again want to minimize h so that we minimize the memory consumed by the write buffers of the buckets. This way, we maximize the number of keys held in memory, s, and hence the lower bound of the number of tuples collapsed in memory, M'. The sketch-based mini-analysis

proposed in the previous section can again be used here to estimate U and help set h to be its lower bound $U/(B-1)$.

However, it is worth noting that setting the number of buckets h to be its optimal value may not guarantee the optimal performance of DINC-hash. This is because DINC-hash is a dynamic caching-based scheme whose performance depends not only on the parameters used in the hash algorithm but also on the temporal locality of the keys in the input data. We detail the impact of the second factor when discussing the potential flooding behavior below.

Approximate Answers and Coverage Estimation. One of the features of DINC-hash is that a large fraction of the update operations for a very frequent key will already have been performed once all the tuples have arrived. To estimate the number of update operations performed for a given key we use the t values: these count the number of tuples that have been collapsed 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 we know that $t[i] \le f_i \le t[i] + M/(s+1)$ from the analysis of FREQUENT and therefore we have the following under-estimate

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

which is very accurate when $t[j]$ or f_j is sufficiently larger that $M/(s+1)$. Hence, for a user-determined threshold ϕ, if $\gamma_i \ge \phi$ we can opt to return the state of the partial computation rather than to complete the computation.

Potential Pathological Behaviors. Using the FREQUENT algorithm as described above leads to a deterministic policy for updating the keys being monitored. Unfortunately, by analogy to online page-replacement policies [Sleator and Tarjan 1985], it is known that any deterministic policy is susceptible to flooding and that the competitive ratio (i.e., the worst-case ratio between the number of I/O operations required by the algorithm and the optimal number) is at least a factor s, the number of keys held in memory. We note that this analysis also applies to other page-replacement policies such as LRU because they are also deterministic. For a simple example, consider the behavior of the FREQUENT algorithm on a sequence that consists of repeating the following sequence of M keys:

$$\langle k_1, k_2, \ldots, k_s, k_{s+1}, k_{s+2}, \ldots, k_{2s+1} \rangle . \tag{8}$$

The FREQUENT algorithm will require $(M-s)$ I/O operations corresponding to all but the first s terms requiring a tuple to be written to disk. In contrast, the optimal solution would be to monitor only $k_1, \ldots, k_s$ and this would require $M \cdot \frac{s+1}{2s+1}$ I/O operations.

4.3.2. Heuristic Improvement and the Marker Algorithm. In our implementation of FREQUENT we added a simple random heuristic that helps avoid the pathological cases described above. When evicting a currently monitored key (and its associated state), our heuristic is to randomly select from those keys whose counters are zero, rather than simply picking the first such key each time. For example, when processing

$$\langle k_1, k_2, \ldots, k_s, k_{s+1}, k_{s+2}, k_1, k_1, k_1, \ldots \rangle, \tag{9}$$

the heuristic means that k_1 is unlikely to be replaced by k_{s+2} (as would happen without the heuristic) due to randomization and hence we are able to combine all the k_1's in the sequence in memory. Note that the heuristic does not jeopardize the above performance guarantees. This behavior is inspired in part by the Marker algorithm [Fiat et al. 1991; McGeoch and Sleator 1991], a paging algorithm that

is known to have good I/O performance (under a worst-case analysis) as a paging algorithm, but no analysis has been conducted to determine their abilities to recognize frequent keys.

The algorithm is sketched in Algorithm 2. The basic algorithm is relatively simple and can be described in terms of the difference to the FREQUENT algorithm with our heuristic: we do not increment the counters $c[i]$ beyond 1. More specifically, the algorithm maintains s bits (binary counters) $c[1],\ldots,c[s]$, s associated keys $k[1],\ldots,k[s]$ currently being monitored, and the state $s[i]$ for each key $k[i]$. Initially, all the bits are set to 0, and all the keys are marked as empty (Line 1). When a new tuple (k,v) arrives, if this key is currently being monitored, the corresponding bit is set to 1 and the state is updated using the update function (Line 4). If k is not being monitored and there exists at least one bit $c=0$, the algorithm randomly picks a j such that $c[j]=0$, evicts the key-state pair $(k[j],s[j])$, and let $c[j]$ monitor k (Line 7-9). If k is not monitored and all $c=1$, then the tuple needs to be written to disk and all $c[i]$ are set to 0 (Line 11-12).

Algorithm 2 Sketch of the **Marker** algorithm

```
1:  c[i] ← 0, k[i] ← ⊥ for all i ∈ {1,2,··· ,s}
2:  for each tuple (k,v) from input do
3:    if k is being monitored then
4:      Suppose k is monitored by c[j], do c[j] ← 1, and s[j] ← update(s[j],v)
5:    else
6:      if {i : c[i] = 0} ≠ ∅ then
7:        j ← randomly pick from {i : c[i] = 0}
8:        Evict key-state pair (k[j],s[j]) to disk
9:        c[j] ← 1, k[j] ← k, and s[j] ← v
10:     else
11:       Write tuple (k,v) to disk
12:       c[i] ← 0 for all i ∈ {1,2,··· ,s}
13:     end if
14:   end if
15: end for
```

Marker versus Frequent Policies for DINC. The randomization of the Marker algorithm plays a key role in minimizing pathological behavior. Specifically it is known that competitive ratio of the enchanted Marker algorithm [McGeoch and Sleator 1991] has the competitive ratio $O(\log s)$ and that this is optimal. Note that this should not be understood to mean that the Marker algorithm is immune to flooding. For example the Marker algorithm will perform equally badly on the example in Eq. 8. However, the fact that the counters saturate at 1 in the Marker algorithm (in contrast to the unbounded counters in the FREQUENT algorithm) has a significant effect on the ability of the algorithm to identify frequent keys. In particular, if an otherwise popular key becomes temporarily infrequent, the Marker algorithm will quickly stop monitoring this key. Hence, there is no guarantee that the frequent elements are identified and therefore a policy based on the Marker algorithm would not support coverage estimation and early approximate answers as detailed above. On the other hand, the FREQUENT algorithm is more sensitive to keys have been historically popular. This is because a popular key k_i that is being monitored will have a high counter value $c[i]$ and therefore it will require the processing of at least $c[i]$ more tuples before k_i is in danger of being evicted. The extent to which it is advisable to use the historical frequency of an item to guide the monitoring of future keys is dependent on the data set. We will explore the issue further empirically in Section 6.3 but the summary is that because the Marker algorithm adapts more quickly to changes in the key distribution, it can end up generating more or less I/O depending on whether or not (respectively) the changes in the distribution are temporary.

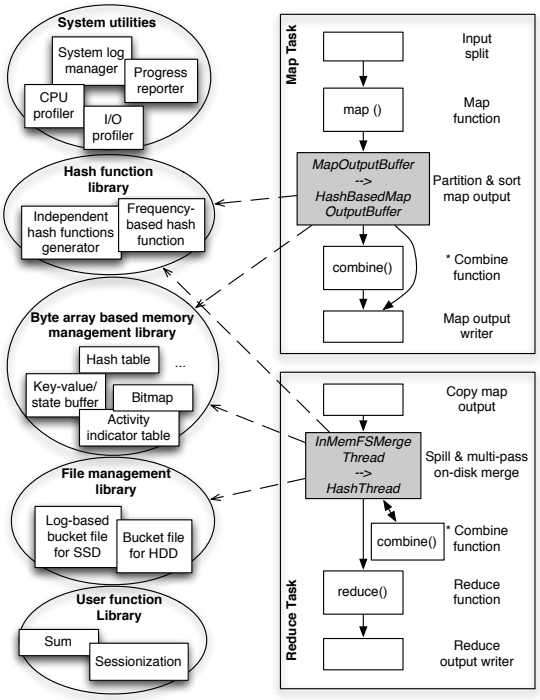


Fig. 10. Architecture of our new one-pass analytics platform.

4.4. Optimizing Other System Parameters

We finally discuss the optimization of key parameters of MapReduce systems such as the Hadoop system. We have described the optimization of Hadoop parameters under the sort-merge implementation in Section 3. Those key parameters, such as the chunk size C and the number of reducers R, are important to our hash-based implementation as well. In particular, when the chunk size is small, we incur high startup cost due to a large number of map tasks. An excessively large chunk size is not favorable, either. One reason is that when a combine function is used in the mappers, the hash algorithm used in the reducers is also applied in the mappers. When the chunk size is large, the output size of a map task may exceed the map buffer size and hence the hash algorithm applied to the map output incurs internal I/O spills, adding significant overheads. The second reason is that a large chunk size delays the delivery of data to the reducers, which is undesirable for incremental processing. Considering both reasons, we use the insights from Section 3 and set the chunk size C to the largest value that keeps the output of a map task fit in the map buffer. Regarding the number of reduce tasks R, we set it such that there is only one wave of reduce tasks, i.e. R equals the number of reduce task slots. Using multiple waves of reduce tasks incurs more I/Os because only the first wave of reduce tasks can fetch map output directly from local memory, while the reduce tasks in other waves are likely to read data from disks.

5. PROTOTYPE IMPLEMENTATION

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 10 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 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.

Pipelining. In our current system, the granularity of data shuffling is determined by the chunk size (with a default value of 64MB), which is fairly small compared with typical sizes of input data (e.g., a terabyte) and hence suitable for incremental processing. In the future, if data needs to be shuffled at a higher frequency, our hash-based framework for incremental processing can be extended with the pipelining approach used in MapReduce Online [Condie et al. 2010]. The right granularity of shuffling will be determined based on the application latency requirement, the extra network overhead of fine-grained data transmission, and the existence of the combine function or not. With some of these issues addressed in MapReduce Online, we will treat them thoroughly in the hash framework in our future work.

Integration within the Hadoop Family. Our system can be integrated with other software in the Hadoop family with no or minimal effort. Any storage system in the Hadoop family, such as HBase, can serve as input to your data analytics system. This is because our internal modification to Hadoop does not require any change of input. Any query processing or data analytics system that is built over Hadoop, such as Hive, can directly benefit from our hash-based system because all our changes are encapsulated in the MapReduce processing layer. The only slight modification required in the query processing layer is transforming the reduce function for Hadoop to $init()$, $cb()$ and $fn()$ functions tailored for incremental computation. The effort of the transformation is typically minimal for analytics tasks.

6. PERFORMANCE EVALUATION

We present an experimental evaluation of our analytics platform and compare it to optimized Hadoop 0.20.1, called 1-pass SM, as described in Section 3. We evaluate all three hash techniques, MR-hash, INC-hash and DINC-hash, described in Section 4 in terms of running time, the size of reduce spill data, and the progress made in map and reduce.

In our evaluation, we use three real-world datasets: 236GB of the WorldCup click stream, 52GB of the Twitter dataset, and 156GB of the GOV2 dataset[6]. 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 both the Twitter and the GOV2 datasets, **trigram counting**, where we report word trigrams that appear more than 1000 times. Our evaluation environment is a 10-node cluster as described in Section 2.3. 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.

6.1. Small Key-state Space: MR-hash versus INC-hash

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. 11(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 III). In contrast, INC-hash finishes earlier due to smaller reduce spills (51GB).

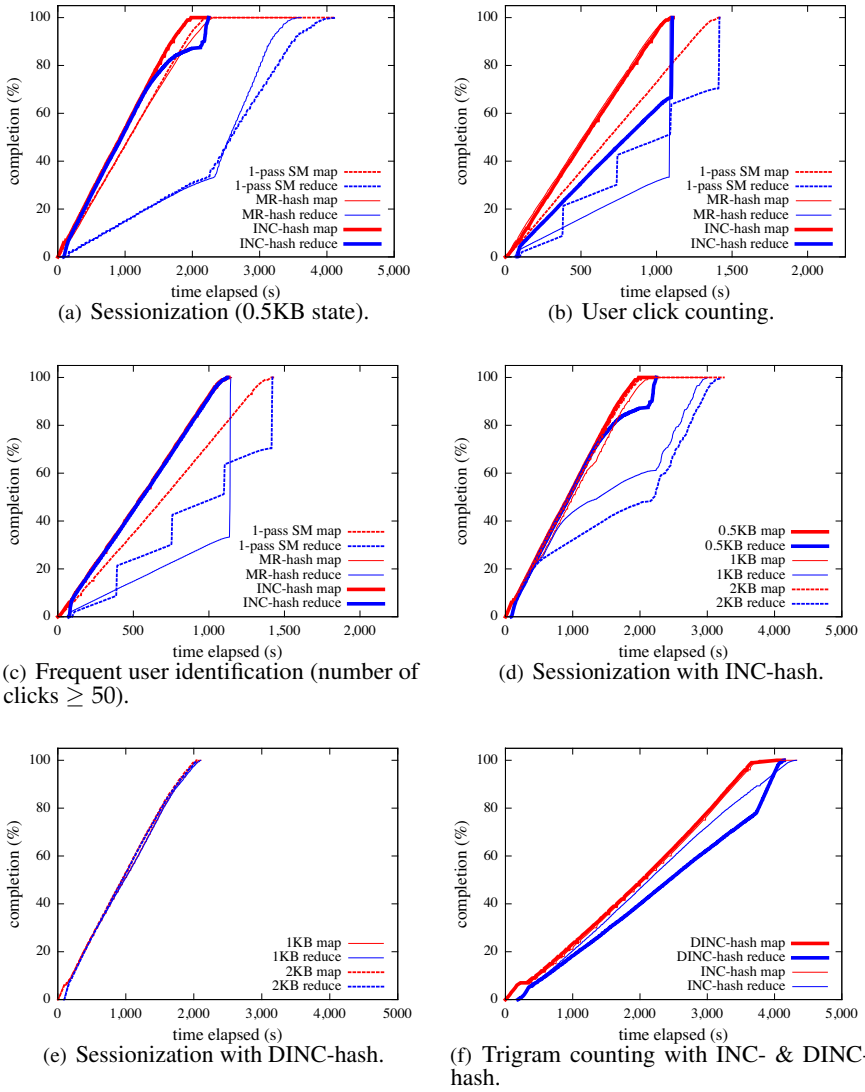
Thus by supporting incremental processing, INC-hash can provide earlier output, and generates less spill data, which further reduces the running time after the map tasks finish.

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. 11(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 III). Hence the workload is not as disk- and network-I/O- intensive.

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



(a) Sessionization (0.5KB state).

(b) User click counting.

(c) Frequent user identification (number of clicks $\geq$ 50).

(d) Sessionization with INC-hash.

(e) Sessionization with DINC-hash.

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

Fig. 11. Progress report using hash implementations.

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. 11(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 reduce 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.

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

Sessionization	1-Pass SM	MR-hash	INC-hash
Running time (s)	4135	3618	2258
Map CPU time per node (s)	1012	659	571
Reduce CPU time per node (s)	518	566	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 IV. 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

6.2. Large Key-state Space: INC-hash versus DINC-hash

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:

Sessionization with varying state sizes. Fig. 11(d) shows the map and reduce progress of INC-hash under three state sizes: 0.5KB, 1KB, and 2KB. A larger state size means that the reduce memory can hold fewer 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 IV. So after map tasks finish, the time for processing data from disk is longer.

To enable DINC-hash for sessionization, a streaming workload, we use the state of a monitored key to hold the clicks of a user in her recent sessions. 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 IV, 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. 11(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 I). 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. 11(f) shows the map and reduce progress plot for INC-hash and DINC-hash with the Gov2 dataset. 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

Table V. Comparing INC-hash, DINC-Frequent and DINC-Marker in terms of reduce spill (GB).

Workload	INC-hash	DINC-Frequent	DINC-Marker
Trigram counting (Twitter)	318	288	316
Trigram counting (Gov2)	221	176	188
Sessionization (2KB)	203	0.1	0

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, 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.

6.3. Dynamic Hashing with Alternative Caching Algorithms

We have compared INC-hash with dynamic hashing based on the deterministic FREQUENT algorithm. Now we further extend the evaluation to dynamic hashing technique based on the randomized Marker algorithm, as described in Section 4.3. We refer the two dynamic hashing techniques as DINC-Frequent, and DINC-Marker, respectively. We compare the sizes of reduce intermediate files across these schemes. We consider three workloads: trigram counting on the Twitter dataset, trigram counting on the Gov2 dataset, and sessionization on the WorldCup dataset with 2KB state.

Different Workloads. To enable an in-depth analysis of these hashing techniques, we characterize our workloads as in one of the following two types:

- *Non-streaming*: When a key-state pair is evicted from memory, it is written to a reduce intermediate file for further processing. Trigram counting belongs to this type of workload. If a trigram is evicted, its state, which is a counter, is written to disk and later summed with other counters for the same trigram.
- *Streaming*: For streaming workloads, the decision to monitor a new key in memory is based both on the existence of a state containing complete data for reduce processing, e.g., when a window closes, and on how the paging algorithm used in each hashing technique finds such a state in memory to evict: (1) For DINC-Frequent, the states with zero counters are scanned to find a qualified state with complete data for reduce processing. (2) For DINC-Marker, the states with zero markers are scanned in a random order to find a qualified state. (3) INC-hash does not allow any state to be replaced. If such a key-state pair exists for eviction, it is output directly but not written to a file on disk. Sessionization belongs to this type of workload. For each user, its state is a window containing the user's recent clicks. The window closes when the user's click session is considered complete. If a user's state needs to be replaced, it is directly output at the time of eviction.

As shown in Table V, for the non-streaming workloads, i.e. trigram counting using Twitter and Gov2 datasets, DINC-Frequent has the smallest intermediate file size because it maintains more history information and is thus more effective to identify hot key than the others. For sessionization, the streaming workload, the two dynamic hashing techniques perform remarkably better than INC-hash. INC-hash incurs significant intermediate I/O because it does not allow a state to be replaced, and thus all the input tuples with keys not in memory are written to files. Both dynamic hashing techniques have negligible intermediate I/O. Their I/O numbers are slightly different due to the way we use them to find a qualified state to evict. It does not show intrinsic difference of the two techniques.

Effects of Locality and Memory Size. We further evaluate our hashing techniques under different data locality properties and different memory sizes using trigram counting on the Twitter dataset. To explore different localities, we construct two datasets: (1) In the original dataset, tweets in various languages are mixed together. (2) The manipulated dataset is generated from the original dataset. We group the tweets that belong to 8 language families into 8 subsets. The set of trigrams from one subset hardly overlaps that from another subset. The subsets are arranged sequentially in the input to each algorithm, and thus show strong locality. This type of locality is similar to the evolving trends of hot topics in real world social networks. In our evaluation we also explore the effect of the memory size by using two settings: *Constrained memory* with 100MB per reducer, which can hold 2% of all key-states across all reducers; *Larger memory* with 600MB per reducer, which can hold 11%-15% of all key-states, depending on the meta data size in different techniques.

Besides DINC-Frequent and DINC-Marker, we also take this opportunity to report the performance of dynamic hashing using another deterministic algorithm, the Space-Saving algorithm (referred to as DINC-SS), as we vary the data locality and memory size. For fair comparison, each technique allocates the given memory for both key-state pairs and meta data. Meta data includes the data structures that organize key-states as a hash-table in order to efficiently search any state by its key, and additional data structures for realizing a replacement policy efficiently. All the techniques share the same meta data for the hash table. The sizes of additional meta data for cache management vary. INC-hash has no additional meta data since it does not support state replacement. DINC-Frequent maintains a set of counters with distinct values, and the two-way mapping between a key-state and its corresponding counter. The implementation of DINC-SS does not include the data structures in the Space-Saving algorithm that do not affect the replacement policy. As a result, DINC-SS has the same data structures for meta data as DINC-Frequent. DINC-Marker maintains the two-way mapping between a key-state and two static markers: 0 and 1. So, the comparison of meta data sizes of the three techniques is: INC-hash $<$ DINC-Marker $<$ DINC-Frequent $=$ DINC-SS.

The comparison of intermediate file sizes is shown in Table VI. In all the combinations of data locality types and memory sizes, DINC-Frequent and DINC-SS differ less than 1% in terms of intermediate I/Os. The reason is the intrinsic similarity of the two algorithms as explained in Section 4.3. With constrained memory size, DINC-Frequent/SS outperforms the others on both the original dataset and the manipulated dataset. This is because DINC-Frequent/SS recognizes hot keys more efficiently based on the history memorized by the counters, and thus makes better use of limited memory. It is also noticeable that the advantage of DINC-Frequent/SS over DINC-Marker is less with the manipulated dataset. This shows the trade-off between DINC-Frequent/SS and DINC-Marker. On one hand, DINC-Frequent/SS is still better at identifying hot keys than DINC-Marker. On the other hand, as stated in Section 4.3, when a subsequent subset of data is read from the manipulated dataset, the states from the previous subset are protected by counters in memory. Since the keys from the new subset are different from the old subset, some tuples from the new subset are written to disk due to this reason until the counters of old states become zero. To the contrary, DINC-Marker protects states with markers, which are boolean counters. So, DINC-Marker is more I/O efficient in the process of clearing states from an old subset for a new subset. We will show this trade-off again with the larger memory setting.

In the larger memory setting using the original dataset, INC-hash performs the best. INC-hash covers the hot keys very well for two reasons. First, there is no strong locality in the dataset. That is, the distinct keys are quite evenly distributed in the dataset. Second, the memory is sufficiently large such that most hot keys are likely to show up in the first s keys. In this case, since INC-hash can hold more keys than the others due to the smallest meta data size as explained before. Hence, it has the least intermediate I/O. In the larger memory setting using the manipulated dataset, INC-hash becomes much worse and DINC-Marker performs the best. INC-hash incurs more I/O because the first s keys do not cover hot keys well due to data locality. DINC-Marker outperforms DINC-Frequent/SS, demonstrating their trade-off again. But with larger memory, the advantage of DINC-Frequent/SS on identifying hot keys is not as significant as before. Thus, DINC-Marker outperforms DINC-Frequent/SS.

Table VI. Comparing INC-hash, DINC-Frequent, DINC-SS and DINC-Marker with the trigram counting workload over the Twitter dataset in terms of reduce spill (GB).

Language	Memory per reducer(MB)	INC-hash	DINC-Frequent	(DINC-SS)	DINC-Marker
Mixed	100	318	**288**	(**288**)	316
Separate	100	502	**266**	(**265**)	272
Mixed	600	**217**	241	(241)	244
Separate	600	455	251	(251)	**212**

In this set of experiments, we show that for streaming workloads, all dynamic hashing techniques have significantly less intermediate I/O than INC-hash. For non-streaming workloads with constrained memory, DINC-Frequent (or similarly, DINC-SS) outperforms the others because it recognizes hot keys more efficiently and thus makes better use of limited memory. For non-streaming workloads with larger memory, when keys are evenly distributed in the data INC-hash outperforms the others due to the early appearance of hot keys and the small meta data used, whereas in the presence of strong locality in the key distribution, dynamic hashing works better than INC-hash and in particular DINC-Marker performs the best for the type of locality of evolving trends.

6.4. Validation using Amazon EC2

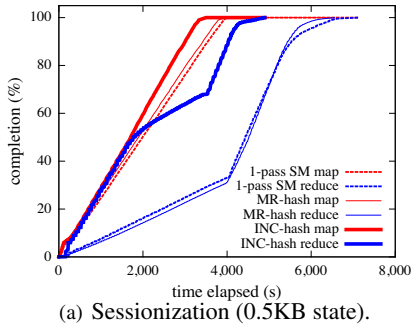
We further validate our evaluation results using much larger datasets in the Amazon Elastic Compute Cloud (EC2). Our EC2 cluster consists of 20 standard on-demand extra large instances (4 virtual cores and 15GB memory per instance) launched with 64-bit Amazon Linux AMI 2012.03 (based on Linux 3.2). One of the instances serves as a head node running the name node and the job tracker. The other 19 instances serve as slave nodes, where each node carries a data node and a task tracker. Each slave node is attached with three Elastic Block Store (EBS) volumes: an 8GB volume for the operating system, a 200GB volume for HDFS, and another 200GB volume for intermediate files. Each slave node is configured with 3 map task slots and 4 reduce task slots.[7] Due to the constraints of the physical memory size and the EC2 setting, the maximum buffer size that we can set for a map task or a reduce task is slightly more than 700MB of memory.[8] Therefore, we set the buffer size for a map or reduce task to be 700MB by default, unless we intentionally reduce the buffer size to evaluate our techniques in the case of constrained memory.

Click Stream Analysis. We first validate the results of click stream analysis on EC2. We use 1TB of the WorldCup click stream, the size of which is increased by a factor of 4 in order to keep the data-to-memory ratio close to that in our previous experiments.

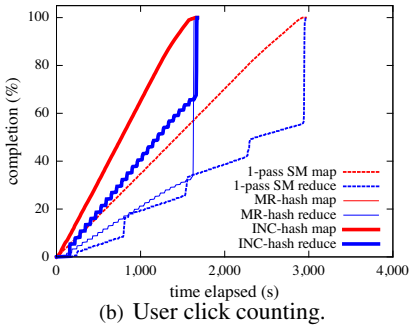
The first set of experiments compare INC-hash with 1-pass SM and MR-hash in the case that memory is sufficiently large for holding most (but not all) key-state pairs. The progress plots of three workloads, namely sessionization, user click counting, and frequent user identification, are shown in Fig. 12(a), 12(b) and 12(c), respectively. These results agree with the observations in Section 6.1: (1) When the memory is sufficiently large, INC-hash is able to perform incremental processing fully in memory, which allows the reduce progress to get closer to the map progress when early output can be generated for the query; this is shown in Fig. 12(a) for sessionization before the map progress reaches 50% and in Fig. 12(c) for frequent user identification over the entire job. (2) The INC-hash technique runs faster than 1-pass SM and MR-hash due to the elimination of sorting and less I/O, which can be seen in all of the three experiments.

[7]Due to the limited performance of a virtual core in EC2 (in contrast to a real core in our previous cluster), the data shuffling progress cannot keep up with the map progress under the setting of 4 map task slots and 4 reduce task slots per node, which was used in our previous experiments. To solve the problem, we set 3 map task slots and 4 reduce task slots per node in EC2.

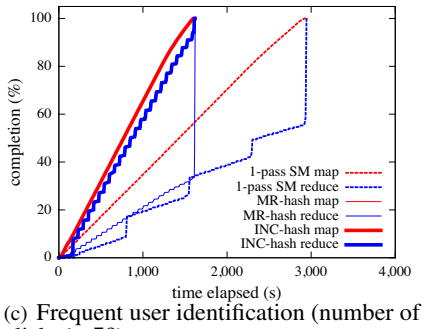
[8]The details of our memory allocation scheme are the following: On a slave node, we are able to allocate 1.4GB JVM heap space for each of the 7 map and reduce task slots, while reserve the remaining memory for the data node, task tracker and the other services in the operating system. And we are able to allocate about half of the JVM heap space, 700MB per task, as the buffer.



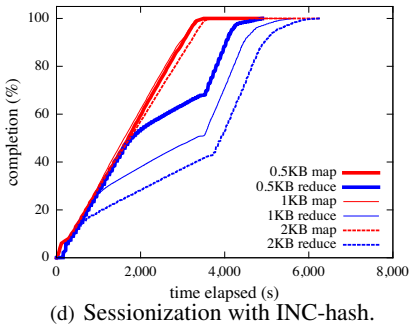
(a) Sessionization (0.5KB state).



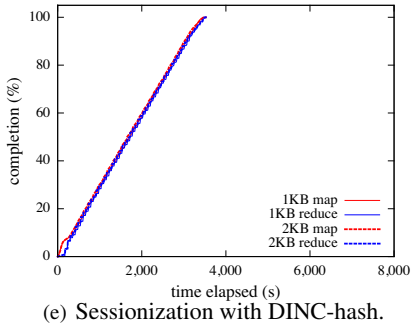
(b) User click counting.



(c) Frequent user identification (number of clicks $\geq$ 50).



(d) Sessionization with INC-hash.



(e) Sessionization with DINC-hash.

Fig. 12. Progress report using hash implementations on EC2.

We next compare the performance of INC-hash (Fig. 12(d)) and that of DINC-hash (Fig. 12(e)) using the sessionization workload that involves a large key-state space far exceeding available memory. These results validate our observations in Section 6.2: When the memory cannot hold all key-state pairs, DINC-hash dramatically reduces I/O to realize incremental processing, and enables the reduce progress to keep up with the map progress.

The results on EC2 also show some different characteristics. We list the performance measurements of the above experiments in Table VII and Table VIII. (1) *Higher CPU cost:* All the experiments have higher per-node CPU cost than our previous experiments, as shown in Table VII. This is because each node on EC2 processes more data whereas the virtual cores on each node have less processing power than the real cores used in our previous experiments. (2) *More I/O for the hash-based techniques:* Comparing the reduce spill numbers in Table VIII and the corresponding numbers in Table IV, we can see that the I/O cost of reduce spill on EC2 increases by a factor ranging between 4.8 and 10.7, more than the factor 4 by which the data increases. This is because the total buffer size on EC2 is less

Table VII. Comparing optimized Hadoop (using sort-merge), MR-hash, and INC-hash on EC2.

Sessionization	1-Pass SM	MR-hash	INC-hash
Running time (s)	6605	6694	4895
Map CPU time per node (s)	1872	1121	1112
Reduce CPU time per node (s)	1353	3080	1648
Map output / Shuffle (GB)	1087	1087	1087
Reduce spill (GB)	1048	1126	547
User click counting	1-Pass SM	MR-hash	INC-hash
Running time (s)	2965	1641	1674
Map CPU time per node (s)	1949	960	960
Reduce CPU time per node (s)	107	91	84
Map output / Shuffle (GB)	8.5	8.5	8.5
Reduce spill (GB)	4.8	0	0
Frequent user identification	1-Pass SM	MR-hash	INC-hash
Running time (s)	2951	1626	1616
Map CPU time per node (s)	1953	961	958
Reduce CPU time per node (s)	93	82	84
Map output / Shuffle (GB)	8.5	8.5	8.5
Reduce spill (GB)	4.8	0	0

Table VIII. Comparing sessionization to INC-hash with 0.5KB state, INC-hash with 2KB state, and DINC-hash with 2KB state on EC2.

	INC (0.5KB)	INC (2KB)	DINC (2KB)
Running time (s)	4895	6252	3523
Reduce spill (GB)	547	978	0.9

Table IX. Comparing INC-hash, DINC-Frequent and DINC-Marker with the trigram counting workload over the Twitter dataset on EC2 in terms of reduce spill (GB).

Language	Memory per reducer(MB)	INC-hash	DINC-Frequent	DINC-Marker
Mixed	100	825	**754**	820
Separate	100	1189	**597**	612
Mixed	700	**557**	581	572
Separate	700	1074	542	**428**

than 4 times the buffer size in our previous experiments, as we explained earlier, which aggregates the memory pressure and causes more I/Os. (3) *Longer running time:* As shown in Table VII and Table VIII, the running times of all the experiments are longer than our pervious experiments. Both the increase in CPU cost and the increase in I/O contribute to the longer running times. We finally note that we also encountered unexpected CPU measurements largely due to the interference from other jobs, such as the reduce CPU time of MR-hash for sessionization in Table VII, which illustrates the difficulty of doing CPU studies in shared environments like EC2.

Trigram Counting. We next validate the results of trigram counting using the Twitter dataset. The goal is to evaluate different hash-based techniques under different data locality properties and reduce buffer sizes. Since the maximum total size of reduce buffers on EC2 is approximately 2.25 times that in our previous experiments, in order to validate results using the same data-to-buffer ratio, we also increase the input data by a factor of 2.25, to 117GB. Similar to our previous experiments, we use two datasets: the *Original dataset*, where tweets in various languages are mixed together; and the *Manipulated dataset*, where the tweets are grouped into multiple language families and fed into each algorithm sequentially. We also vary the reduce buffer size using two settings: *Constrained memory* with 100MB per reducer, and *Larger memory* with 700MB per reducer. The comparison of intermediate file sizes is shown in Table IX. In the constrained memory setting, DINC-Frequent performs the best on both datasets. In the larger memory setting using the original dataset, INC-hash outperforms the other two. In the larger memory setting using the manipulated dataset, DINC-Marker performs the best. The results agree with our previous experiments, as summarized in Section 6.3.

Summary. The results in this section are summarized below:

- Our incremental hash technique provides much better performance than optimized Hadoop using sort-merge and the baseline MR-hash: INC-hash significantly improves the progress of the map tasks, due to the elimination of sorting, and given sufficient memory, enables fast in-memory processing of the reduce function.
- For a large key-state space, dynamic hashing based on frequency analysis can significantly reduce intermediate I/O and enable the reduce progress to keep up with the map progress for incremental processing.
- For streaming workloads, our dynamic hashing techniques can provide up to 3 orders of magnitude reduction of intermediate I/O compared to other techniques.
- For non-streaming workloads, there are trade-offs between hashing techniques depending on the data locality and memory size. Dynamic hashing using the frequency analysis tends to work well under constrained memory. When there is sufficiently large memory, other hashing techniques may perform better for various data locality properties.
- Most of the above results are also validated using much larger datasets and a larger cluster in Amazon EC2, with only minor differences due to the use of virtual cores and the hard constraint on the buffer size in the cloud setting.

7. RELATED WORK

Query Processing using MapReduce. [Chaiken et al. 2008; Jiang et al. 2010; Olston et al. 2008; Pavlo et al. 2009; Thusoo et al. 2009; Yu et al. 2009] 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 [Condie et al. 2010] which we discussed in detail in Sections 2 and 3. Dryad [Yu et al. 2009] 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 [Yang et al. 2007] implements hash join using a technique similar to our baseline MR-hash, but lacks further implementation details. SCOPE [Chaiken et al. 2008] is a SQL-like declarative language designed for parallel processing with support of three key user-defined functions: process (similar to `map`), reduce (similar to `reduce`) and combine (similar to a join operator). SCOPE provides the same functionality as Merge Reduce Merge and does not propose new hash techniques beyond the state-of-the-art. Several other projects are in parallel to our work: The work in [Babu 2010] focuses on optimizing Hadoop parameters and ParaTimer [Morton et al. 2010] 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.

Computation Models for MapReduce. Karloff et al. [Karloff et al. 2010] suggest a theoretical abstraction of MapReduce. The abstraction employs a model that is both general and sufficiently simple such that it encourages algorithm research to design and analyze more sophisticated MapReduce algorithms. However, this model is based on simple assumptions that are not suitable for system research like our study. First, no distinction is made between disk and main memory. Second, the model is only applicable to the case that each machine has memory $O(n^{1-\epsilon})$ where n is the size of entire input and ϵ is a small constant. Third, the model assumes that all map tasks have to complete before the beginning of the reduce phase, which makes the model invalid for incremental computation.

Advanced Algorithms for Finding Frequent Items. The FREQUENT algorithm [Misra and Gries 1982; Berinde et al. 2009], also known as the Misra-and-Gries algorithm, is a deterministic algorithm to identify ϵ-approximate frequent items in a entire data stream with $1/\epsilon$ counters. Each counter corresponds to an item monitored in memory, and gives an underestimate about the frequency of the item. The algorithm makes no assumption on the distribution of the item frequencies, and can

be implemented such that each update caused by a data item takes $O(1)$ time. The Space-Saving algorithm [Metwally et al. 2005] employs a mechanism similar to that in the FREQUENT algorithm. Each counter in Space-Saving gives an overestimate about the frequency of the corresponding monitored item. In addition, an upper bound for the error of each counter from the true frequency is maintained. Thus, Space-Saving is also able to give an underestimate of the frequency for each monitored item. However, when Space-Saving is used as a paging algorithm, the paging behavior relies only on the counters, but not affected by the error bounds.We explained the similar performance for paging between the above algorithms in Section 4.3 and exhibited it empirically in Section 6.3. The algorithm proposed in [Lee and Ting 2006] aims to find frequent elements in sliding windows, which combines the FREQUENT algorithm with another existing algorithm. Nevertheless, the update time scales with the number of monitored keys, and hence renders poor performance without window operators. This work is related to our future work of extending our one-pass analytics platform to support stream processing.

Parallel Databases. Parallel databases [DeWitt et al. 1986; DeWitt and Gray 1992] 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 [DeWitt et al. 1986], 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 [Abadi et al. 2005] and the routing of tuples between nodes [Tian and DeWitt 2003], 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 [Zou et al. 2010] and S4 [Neumeyer et al. 2010]. 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.

Our work presented in this paper significantly extends our previous work [Li et al. 2011]: (*i*) We provide a more detailed analysis of MapReduce online [Condie et al. 2010] with pipelining of data. Through an extensive set of experiments we provide new understanding of the effects of the HDFS chunk size, the number of reducers, and the snapshot mechanism when adding pipelining to existing MapReduce implementations. (*ii*) To better validate our analytical model of existing MapReduce systems, we strengthen our evaluation with new results on the I/O cost in addition to the time cost. (*iii*) We significantly extend our discussion of incremental hashing, INC-hash, by separating the concepts of incremental processing and partial aggregation, adding a new analysis of memory requirements and I/O cost, and proposing to treat its sensitivity to unknown parameters using sketch-based analysis. (*iv*) We also include a major extension of dynamic incremental hashing, DINC-hash, with new guarantees for skewed data, discussion of its sensitivity to unknown parameters and potential pathological behaviors (e.g., flooding), and a new randomized caching algorithm to overcome the flooding problem. (*v*) We perform over a dozen new experiments to compare various hashing algorithms and caching policies under different workloads, data localities, and memory sizes. The detailed results are reported in Section 6.

8. 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 build use cases and evaluate our system in more application domains, such as web document analysis, web log analysis [PigMix 2008], complex graph analysis and genomic analysis [Roy et al. 2012]. We will also extend our hash-based data analytics platform in several ways. First, we will explore further improvement of our dynamic incremental hashing techniques, including seeking performance gains by generalizing the Marker algorithm, where the counter of each monitored item can go up to a parameterized value instead of 1, and seeking hybrid approaches that adapt between the FREQUENT algorithm and the marker algorithm for best performance. Second, we will explore a dynamic mechanism for controlling the number of hash buckets in our hash algorithms, which provides good performance without requiring precise knowledge of the key-state space of input data. Third, we will extend our work to online query processing that provides early approximate answers with statistical guarantees. Fourth, we will further extend to stream processing by adding the support of pipelining and window operations. Our incremental data processing platform developed in this work provides fundamental systems support for the above two analytics systems.

ACKNOWLEDGMENTS

The work has been supported in part by the NSF grants IIS-0746939, CCF-0953754, CNS-0855128, CNS-0916972, CNS-0923313 and CNS-1117221, as well as grants from Google Research, IBM Research, NEC Labs, UMass Science & Technology Fund and Massachusetts Green High Performance Computing Center.

REFERENCES

ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CHERNIACK, M., HYON HWANG, J., LINDNER, W., MASKEY, A. S., RASIN, E., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. 2005. The design of the borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Database Research (CIDR)*. 277–289.

BABU, S. 2010. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. ACM, New York, NY, USA, 137–142.

BERINDE, R., CORMODE, G., INDYK, P., AND STRAUSS, M. J. 2009. Space-optimal heavy hitters with strong error bounds. In *PODS*. 157–166.

CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. 2008. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1,* 2, 1265–1276.

CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. 2004. Finding frequent items in data streams. *Theor. Comput. Sci. 312,* 1, 3–15.

CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. 2010. Mapreduce online. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, Berkeley, CA, USA, 21–21.

CORMODE, G. AND MUTHUKRISHNAN, S. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms 55,* 1, 58–75.

DEAN, J. AND GHEMAWAT, S. 2004. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 10–10.

DEWITT, D. AND GRAY, J. 1992. Parallel database systems: the future of high performance database systems. *Commun. ACM 35,* 6, 85–98.

DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. 1986. Gamma - a high performance dataflow database machine. In *VLDB*. 228–237.

DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D. A., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. 1990. The gamma database machine project. *IEEE Trans. Knowl. Data Eng. 2,* 1, 44–62.

FIAT, A., KARP, R. M., LUBY, M., MCGEOCH, L. A., SLEATOR, D. D., AND YOUNG, N. E. 1991. Competitive paging algorithms. *J. Algorithms 12,* 4, 685–699.

GANGULY, S. AND MAJUMDER, A. 2007. Cr-precis: A deterministic summary structure for update data streams. In *ESCAPE*. 48–59.

HELLERSTEIN, J. M. AND NAUGHTON, J. F. 1996. Query execution techniques for caching expensive methods. In *SIGMOD Conference*. 423–434.

JIANG, D., OOI, B. C., SHI, L., AND WU, S. 2010. The performance of mapreduce: an in-depth study. In *VLDB*.

KANE, D. M., NELSON, J., AND WOODRUFF, D. P. 2010. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '10. ACM, New York, NY, USA, 41–52.

KARLOFF, H., SURI, S., AND VASSILVITSKII, S. 2010. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 938–948.

LEE, L. K. AND TING, H. F. 2006. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 290–297.

LI, B., MAZUR, E., DIAO, Y., MCGREGOR, A., AND SHENOY, P. J. 2011. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD Conference*, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 985–996.

MAZUR, E., LI, B., DIAO, Y., AND SHENOY, P. J. 2011. Towards scalable one-pass analytics using mapreduce. In *IPDPS Workshops*. IEEE, 1102–1111.

MCGEOCH, L. A. AND SLEATOR, D. D. 1991. A strongly competitive randomized paging algorithm. *Algorithmica 6,* 6, 816–825.

METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. 2005. Efficient computation of frequent and top-k elements in data streams. In *Database Theory - ICDT 2005*, T. Eiter and L. Libkin, Eds. Lecture Notes in Computer Science Series, vol. 3363. Springer Berlin / Heidelberg, 398–412.

MISRA, J. AND GRIES, D. 1982. Finding repeated elements. *Sci. Comput. Program. 2,* 2, 143–152.

MORTON, K., BALAZINSKA, M., AND GROSSMAN, D. 2010. Paratimer: a progress indicator for mapreduce dags. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. ACM, New York, NY, USA, 507–518.

MUTHUKRISHNAN, S. 2006. *Data Streams: Algorithms and Applications*. Now Publishers.

NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. 2010. S4: Distributed stream computing platform. In *ICDM Workshops*. 170–177.

OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*. 1099–1110.

PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. 2009. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*. 165–178.

PIGMIX. 2008. *Pig Mix benchmark*. `https://cwiki.apache.org/confluence/display/PIG/PigMix`.

RAMAKRISHNAN, R. AND GEHRKE, J. 2003. *Database management systems (3. ed.)*. McGraw-Hill.

ROY, A., DIAO, Y., MAUCELI, E., SHEN, Y., AND WU, B.-L. 2012. Massive genomic data processing and deep analysis. *Proc. VLDB Endow. 5,* 12, 1906–1909.

SHAPIRO, L. D. 1986. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11,* 3, 239–264.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM 28,* 2, 202–208.

THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. 2009. Hive - a warehousing solution over a map-reduce framework. *PVLDB 2,* 2, 1626–1629.

TIAN, F. AND DEWITT, D. J. 2003. Tuple routing strategies for distributed eddies. In *VLDB '03: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, 333–344.

WHITE, T. 2009. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc.

YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 1029–1040.

YU, Y., GUNDA, P. K., AND ISARD, M. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, New York, NY, USA, 247–260.

ZOU, Q., WANG, H., SOULÉ, R., HIRZEL, M., ANDRADE, H., GEDIK, B., AND WU, K.-L. 2010. From a stream of relational queries to distributed stream processing. *PVLDB 3,* 2, 1394–1405.