# Towards Scalable One-Pass Analytics Using MapReduce

Edward Mazur, Boduo Li, Yanlei Diao, Prashant Shenoy
*Department of Computer Science*
*University of Massachusetts, Amherst*
*Email: {mazur, boduo, yanlei, shenoy}@cs.umass.edu*

***Abstract*—An integral part of many data-intensive applications is the need to collect and analyze enormous datasets efficiently. Concurrent with such application needs is the increasing adoption of MapReduce as a programming model for processing large datasets using a cluster of machines. Current MapReduce systems, however, require the data set to be loaded into the cluster before running analytical queries, and thereby incur high delays to start query processing. Furthermore, existing systems are geared towards batch processing.**

**In this paper, we seek to answer a fundamental question: what architectural changes are necessary to bring the benefits of the MapReduce computation model to incremental, one-pass analytics, i.e., to support stream processing and online aggregation? To answer this question, we first conduct a detailed empirical performance study of current MapReduce implementations including Hadoop and MapReduce Online using a variety of workloads. By doing so, we identify several drawbacks of existing systems for one-pass analytics. Based on the insights from our study, we list key design requirements for incremental one-pass analytics and argue for architectural changes of MapReduce systems to overcome their current limitations. We conclude by sketching an initial design of our new MapReduce-based platform for incremental one-pass analytics and showing promising preliminary results.**

***Keywords*-MapReduce; performance analysis; data streams; parallel data processing**

## I. Introduction

An integral part of many data-intensive applications is the need to collect and analyze enormous datasets efficiently. MapReduce has emerged as a popular programming model for parallel processing of large datasets using a cluster of machines. Current MapReduce systems, however, require the data set to be loaded into the cluster before running analytical queries, and thereby incur high delays to start query processing and produce results. Furthermore, existing systems are geared towards batch processing. Despite these limitations, it is attractive to consider whether the benefits of the MapReduce model can be brought to systems that perform scalable one-pass analytics over large data sets.

Consequently, in this paper, we seek to answer a fundamental question: is the MapReduce computation model suitable for incremental, one-pass analytics? This includes both stream processing and online aggregation with early approximate answers. To perform incremental one-pass analytics, the system must be able to evaluate the query on a large data set as data arrives into the system[1]. There are two primary challenges in doing so:

- ► *Incremental processing*. Data stream processing requires that an answer to a query be produced as soon as all the data needed to produce this answer has been read. Since current MapReduce systems wait until the *entire* data set is loaded to begin query processing and further employ batch processing of query programs, they are ill-suited for incremental processing.
- ► *Performance*. Scalable one-pass analytics for parallel stream processing requires *fast in-memory processing* of a MapReduce query program for all (or most) of the data. If some subset of data has to be staged to disks, then the I/O cost of such disk operations must be minimized. In contrast, existing MapReduce query programs can trigger both expensive CPU operations and significant amounts of intermediate I/O to stage data in map and reduce processing.

To better understand how to address these challenges, in this paper, we conduct a detailed performance study of existing MapReduce systems using a range of workloads and storage architectures. In doing so, our paper makes the following technical contributions.

1. We begin by benchmarking the current MapReduce computation model using the popular open-source Hadoop implementation and the recently proposed MapReduce Online system [6] with pipelining of intermediate data. We consider a variety of workloads, including click stream analysis and web document analysis, and a range of architectural choices. Based on a rigorous analysis of our experimental results, we find that the current MapReduce computation model uses a sort-merge technique to support parallel processing, which poses a significant barrier to incremental one-pass analytics: (*i*) The sorting step of the sort-merge implementation incurs expensive CPU operations. (*ii*) The merge step in sort-merge is blocking and can incur significant I/O costs given large amounts of intermediate data. (*iii*) Using extra storage devices and alternative storage architectures do not eliminate blocking or the I/O bottleneck.

We also find that MapReduce Online takes only a limited step towards stream processing—its pipelining does not

[1]We do not consider an infinite sequence due to the overhead of fault tolerance as explained in [6].

significantly reduce blocking or CPU and I/O overheads, from a stream processing standpoint, and only redistributes workloads between the map and reduce tasks in a query program.

2. Based on the insights from our study, we highlight several key design requirements that must be addressed to perform incremental one-pass analytics using MapReduce. We argue for architectural changes of MapReduce systems in order to overcome their current limitations, including (*i*) replacing the sort-merge implementation of the MapReduce computation model with a purely hash-based method, (*ii*) extending the hash implementation with incremental processing and partial aggregation for a wide range of analytical tasks, and (*iii*) when memory is limited, supporting fast in-memory processing of important groups of data with little I/O cost.

We conclude the paper by sketching an initial design of our new MapReduce-based platform for incremental one-pass analytics and showing promising preliminary results.

## II. Background on MapReduce

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

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. As an example, consider how we would parse a click stream to find the most visited pages. More specifically, we want to count how many times each page has been visited. Suppose the schema for a visits table is (timestamp, user, url). Consider the following SQL query:

```
SELECT COUNT(*) FROM visits GROUP BY url;
```

Page clicks are grouped by url and then aggregated using `COUNT` for each url. Now consider the equivalent MapReduce job for computing page frequencies below. The map function emits a new <url, count> tuple for each visit in the click stream, where the count is 1. The reduce function then groups these tuples by url and sums the number of visits to each url.

```
function map(rid, visit) {
  emit(visit.url, 1); }

function reduce(url, iterator<int> count)
{
  int total = 0;
  foreach count c
    total += c;
  emit(url, total); }
```

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

The MapReduce program of an analytical query includes both the map and reduce functions compiled from the query (e.g., using a MapReduce-based query compiler [23], [14]) and the MapReduce system's code for parallelism.

### A. Overview of the Hadoop Implementation

We consider Hadoop, the most popular open-source implementation of MapReduce, in our study. Hadoop uses block-level scheduling and a sort-merge technique [33] to implement the group-by functionality for parallel processing (Google's MapReduce system is reported to use a similar implementation [7], but further details are lacking due to the use of proprietary code).

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

Given a query, its MapReduce job is assigned $m$ map tasks (mappers) and $r$ reduce tasks (reducers) concurrently on each node. As Fig. 1 shows, each mapper reads a block 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 *block-level 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 a file using synchronous I/O. A mapper completes after its output has been persisted for fault tolerance.

Map output is then shuffled to the reducers (in the *shuffling* phase). To do so, reducers periodically poll a centralized service asking about completed mappers and once notified, requests data directly from the completed mappers (*pull*-based communication). Under normal circumstances, this data transfer happens soon after a mapper completes and so this data is often available in the mapper's memory.

Over time, a reducer collects pieces of sorted output from many completed mappers. Unlike before, this data cannot be assumed to fit in memory for larger 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

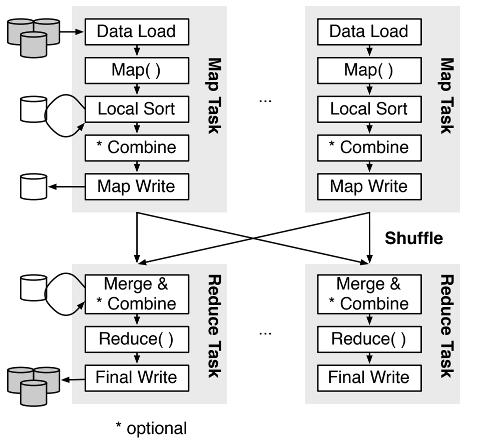


Figure 1. Architecture of the Hadoop implementation of MapReduce.

such files exceeds a threshold $F$ (in a so-called *multi-pass merge* phase). When a reducer has collected all of the map output, it will perform a multi-pass merge if the on-disk files exceed $F$; otherwise, it will perform a final merge to produce all key-value pairs in sorted order of the key. Over the sorted file, the reducer identifies each list of values sharing the same key and then applies the reduce function to the list. The output of the reduce function is written back to HDFS.

Finally, when the reduce function is commutative and associative, a combine function (typically sharing the code with the reduce function) is applied right after the map function, as shown in Fig. 1, to perform partial aggregation and reduce the size of map output. It can be further applied in a reducer when its data buffer fills up.

## III. Benchmarking and Analysis

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

### A. Experimental Setup

We consider two applications in benchmarking: click stream analysis which represents workloads for stream processing, and web document analysis which represents workloads for one-pass analysis over stored data. The workloads tested are summarized in Table I. (In ongoing work, we are extending our benchmark to Twitter feed analysis and complex queries such as top-$k$ and graph queries.) [2]

In click stream analysis, an important task is sessionization, which reorders click logs into individual user sessions. Its MapReduce program employs the map function to extract the url and user id from each click log, then groups click logs by user id, and implements the sessionization algorithm in the reduce function. A key feature of this task is a large amount of intermediate data due to the reorganization of all click logs by user id. Another task in click stream analysis is page frequency counting. As a simple variant on the canonical word counting problem, it counts the number of visits to each url. A similar task counts the number of clicks that each user has made. For such counting problems, a combine function can be applied to significantly reduce the amount of intermediate data. For this application, we use the click logs from the World Cup 1998 website [3] and replicate it to larger sizes as needed.

The second application is web document analysis. A key task is inverted index construction, in which a large collection of web documents (or newly crawled web documents) is parsed and an inverted index on the occurrences of each word in those documents is created. In its MapReduce program, the map function extracts (word, (doc id, position)) pairs and the reduce function builds a list of document ids and positions for each word. The intermediate data is typically smaller than the document collection itself, but still of a substantial size. Other useful tasks in this application involve word frequency analysis, which are similar to page frequency analysis mentioned above, hence omitted in Table I. For this application, we use the 427GB GOV2 document collection created from an early 2004 crawl of government websites.[4]

The Hadoop configuration mainly used the default settings with a few changes. We ran the NameNode and JobTracker daemons on the head node and ran DataNode and TaskTracker daemons on each of the 10 compute nodes. The HDFS block size was 64MB. HDFS replication was turned down to 1 from the default 3. The map output buffer was tuned for each workload to ensure there were no spills to disk. JVM reuse was enabled. The JVM heap size was set to 1GB.

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

### B. Result Analysis

Table I shows the running time of the workloads as well as the sizes of input, output, and intermediate data in our benchmark. Due to space constraints, our analysis below focuses on the sessionization workload that involves the largest amount of intermediate data. We comment on the results of other workloads in the discussion whenever

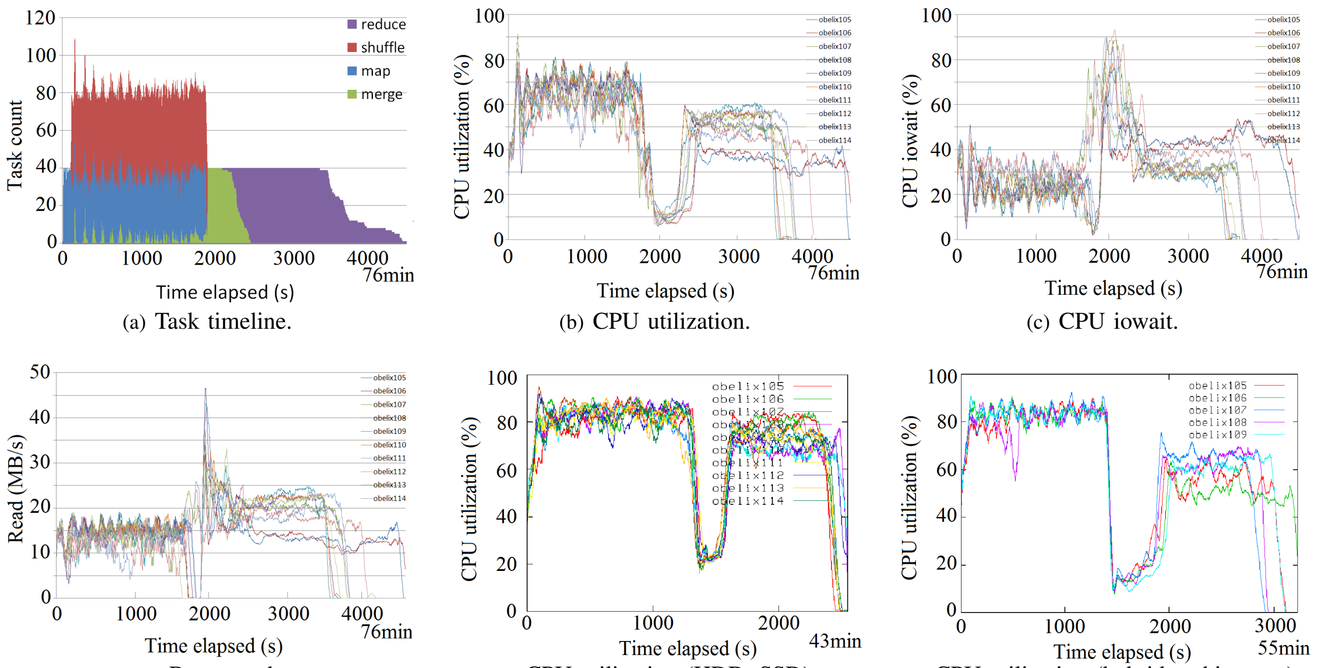
[2]An existing benchmark [24] mostly contains simple aggregate queries over stored data. Our benchmark includes more complex tasks required in real-world applications, many of which are performed on data streams.

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

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

Table I
WORKLOADS AND THEIR RUNNING TIME IN THE BENCHMARK.

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



(a) Task timeline.
(b) CPU utilization.
(c) CPU iowait.
(d) Bytes read.
(e) CPU utilization (HDD+SSD).
(f) CPU utilization (hybrid architecture).

Figure 2. Experimental results using the sessionization workload.

appropriate. Fig. 2 (a) shows the task timeline for the sessionization workload, i.e., the number of tasks for the four main operations in its MapReduce job: *map* (including sorting), *shuffle*, *merge* (the multi-pass part), and *reduce* (including the final scan to produce a single sorted run). As can be seen, time is roughly evenly split between the map and reduce phases, with a substantial merge phase in between. Also note that some periodic background merges take place even before all map tasks complete. When the intermediate data is reduced as in other workloads, first the merge phase shrinks and then the reduce phase also shrinks.

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

**2. Cost of Map Output**. A potential I/O bottleneck can be the writes of map output to disk using synchronous I/O, required for fault tolerance in MapReduce. In our benchmark, we observed that although each map task did block while performing this write, it did not take up a large portion of a map task's lifetime. In the sessionization workload with a large amount of map output data, these writes took 1.3 seconds on average, while the average map task running time took 21.6 seconds. This 6% time did not make a significant

Table II
AVERAGE CPU CYCLES PER NODE, MEASURED BY CPU SECONDS, IN THE MAP PHASE (256GB WORLDCUP DATASET).

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

contribution to a map task's running time relative to other parts. Furthermore, the recent MapReduce Online system [6] proposes to pipeline map output to the reducers and persists the data using asynchronous I/O. Hence, it can be used as a solution if the map output may be observed as an I/O bottleneck elsewhere.

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

First, we observe from Fig. 2 (b) that CPUs are busy in the map phase. It is important to note that the map function in the sessionization workload is relatively CPU light: it parses each click log into user id, timestamp, url, etc., and emits a key-value pair where the key is the user id and the value contains other attributes. The rest of cost in the map phase is attributed to sorting of the map output buffer. To quantify the costs of the map function and sorting, we performed detailed profiling of CPU cyles consumed by each, as shown in Table II. In the sessionization workload, roughly 61% of CPU cycles were consumed by the map function while 39% was by sorting. In the per-user click counting workload, the map function simply emits pairs in the form of (user id, “1”), and up to 48% of CPU cycles were consumed by sorting these pairs. We further note that if we expedite click log parsing in the map function using the recent proposal of mutable parsing [17], the overhead of sorting will be even more prominent in the map phase.

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

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

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

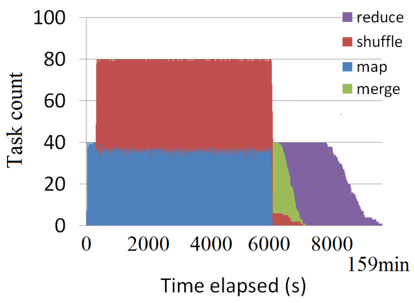
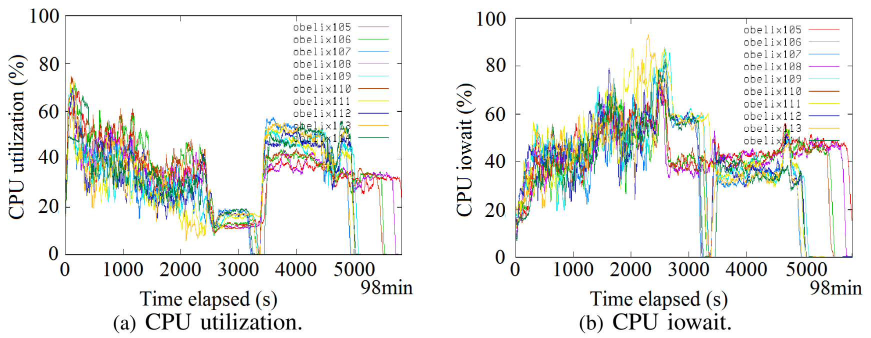


Figure 3. Task timeline using the inverted index construction workload.

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

Moreover, the multi-pass merge operation is also I/O intensive. Our profiling tool shows that in sessionization, the reducers read and write 370GB data in the multi-pass merge operation while the input data has only 256 GB, as shown in Table I. The inverted index workload incurs a somewhat reduced but still substantial I/O cost of 150GB data in this operation. As shown in Fig. 3, the blocking merge phase is present in this workload as well. Progress is stopped until local intermediate data is merged on each node. In simpler workloads, such as counting the number of clicks per user, there is an effective combine function to reduce the intermediate data size. However, it is interesting to observe from Table I that even if there is ample memory to perform in-memory processing, the multi-pass merge still causes I/O, e.g., 1.4GB spill from the reducers. This is because when the memory fills up, each reducer applies the combine function to the data in memory but still writes the data to disk waiting for all future data to produce a single sorted run.

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



(a) CPU utilization. (b) CPU iowait.

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

### C. Architectural Improvements

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

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

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

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

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

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

### D. MapReduce Online

We finally consider a recent system called MapReduce Online that implements a Hadoop Online Prototype (HOP) with pipelining of data [6]. This prototype has two distinct features: First, as each map task produces output, it can push data eagerly to the reducers. The granularity of such data transmission is controlled by a parameter. Second, an adaptive control mechanism is in place to balance work between mappers and reducers. For instance, if the reducers become overloaded, the mappers will write the output to local disks and wait until reducers are able to keep up again. A potential benefit of HOP is that with pipelining, reducers receive map output earlier and can begin multi-pass merge earlier, thereby reducing the time required for the merge work after all mappers finish.

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

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

There are several subtle differences from the previous results of benchmarking Hadoop. The total running time is actually longer using MapReduce Online. A possible explanation for this difference is that MapReduce Online is based off an older version of Hadoop, 0.19.2, whereas we benchmarked using 0.20.0. Any performance optimizations made during this time will only be present in the newer version. Another possible reason is that MapReduce Online transmits map output eagerly in finer granularity and hence increases network cost, which in turn causes lower CPU utilization. Another thing to note is that the CPU utilization in the map phase when running HOP is lower than when running on stock Hadoop. We verified that the total number of CPU cycles consumed in the map phase are similar across both implementations by observing that HOP spends a greater amount of time in the map phase, with a somewhat reduced level of CPU utilization. Finally, this prototype moves some of the sorting work to reducers, which may also affect the CPU utilization in different phases of the job. In our ongoing work, we will continue benchmarking MapReduce Online, including the use of other workloads, to better explain its behavior.

### *E. Summary of Results*

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

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

## IV. Design Requirements

In this section, we articulate the design requirements for incremental, one-pass analytics using MapReduce. The insights from our experimental evaluation indicate several architectural drawbacks that must be addressed before MapReduce can be effectively used for incremental one-pass analytics. They also reveal techniques that can be incorporated to improve performance. Below, we discuss how MapReduce systems should be rearchitected to overcome their current drawbacks and augmented with additional techniques for performance. To aid our discussion, we highlight the architectural choices made by Hadoop, MapReduce Online, and an ideal system for incremental, one-pass analytics in Table III.

1. To perform group-by for parallel processing, Hadoop and MapReduce Online adopt the sort-merge implementation, which causes unnecessary CPU and I/O overheads, and blocks the reduce function from starting especially when multi-pass merge is applied—an observation that is corroborated by our experimental findings. Consequently, we advocate a hash-only implementation to replace the sort-merge implementation of group-by, thereby eliminating the CPU overhead of sorting. It further offers a flexible framework for incorporating new techniques to address other drawbacks of existing systems including blocking and I/O overheads.

2. Regarding communication, MapReduce Online allows push-based shuffling in addition to the pull-only style in Hadoop. By pushing data items from the map output, it allows the reducers to receive data earlier, from which latency-sensitive applications may potentially benefit. Our proposed system also supports push-based shuffling to extract this benefit.

3. Incremental processing requires that an answer to a query be produced as soon as all the data needed to produce this answer has been read. Consider a query that returns all the groups where the count of items exceeds a threshold. For incremental processing, a group needs to be output as soon as the count of its items has reached the threshold. Hadoop does not satisfy this requirement since it can at most perform partial aggregation in individual data blocks but do not perform global aggregation until sort-merge completes. MapReduce Online can perform sort-merge and global aggregation periodically when a snapshot is generated.

| | **Hadoop** | **MR Online** | **Incremental One-pass Analytics** |
|---|---|---|---|
| *Group By* | Sort-Merge | Sort-Merge | Hash only |
| *Shuffling* | Pull | Push / Pull | Push / Pull |
| *Incremental Processing* | No | No (periodic snapshot-based output only) | Fully incremental |
| *In-memory Processing* | No | No | Yes, if data $<$ memory; otherwise, in-memory processing for important keys |

Table III
COMPARISON BETWEEN HADOOP, MAPREDUCE ONLINE, AND AN IDEAL SYSTEM FOR INCREMENTAL, ONE-PASS ANALYTICS.

Using a hash-based implementation, it is possible to support incremental processing using new techniques. As data items arrive at a reducer, the reduce function can be applied to all groups simultaneously when there is sufficient memory. To reduce the cost of such incremental processing, partial aggregation using the combine function can be applied in the mappers to reduce the size of data sent to reducers. How to support the combine function for complex analytical tasks such as top-$k$ and graph queries is an open question.

4. For performance, stream processing requires fast in-memory processing without disk I/O. Hadoop and MapReduce Online are unable to meet this requirement due to the expensive CPU operations and significant I/O operations in the sort-merge implementation, as observed in our experimental study. Our proposed hash-based system can support fast in-memory processing whenever the computation states of the reduce function for all groups fit in memory. In cases when memory is not large enough to do so, a plausible solution is to recognize important groups (e.g., specified by the application or based on their frequencies in the data set) and allocate memory intelligently to support fast in-memory processing for these groups.

To meet all the requirements for incremental one-pass analytics, we propose to design a new parallel data streaming platform that (*i*) uses a purely hash-based framework to support MapReduce's group-by functionality for parallelism, (*ii*) extends the hash framework with incremental computation, where the computation can be either exact or approximate, to minimize intermediate data sizes and return pipelined answers, and (*iii*) supports in-memory stream processing for frequent keys in the data set when memory is less than the intermediate data size. This new platform aims to offer near real-time stream processing that obviates the need for data loading and returns pipelined answers as data arrives, while also supporting efficient query processing.

## V. System Implementation and Initial Results

In this section, we sketch an initial design of our system and show preliminary results. Fig. 5 shows the architecture of our system based on Hadoop. We re-implement the sort-merge related components in map and reduce modules (the two gray boxes) using hash-based techniques.

In the map module, we provide two options to replace the use of sorting for partitioning and grouping map output by keys. (1) The map output is scanned once for partitioning, and no effort is spent for grouping. This method only works if there is no combine function. (2) Our system also implements Hybrid Hash [25] if there is a combine function. In most cases, the map output fits in memory so Hybrid Hash is simply in-memory hashing. Both options avoid sorting so as to save CPU cycles.

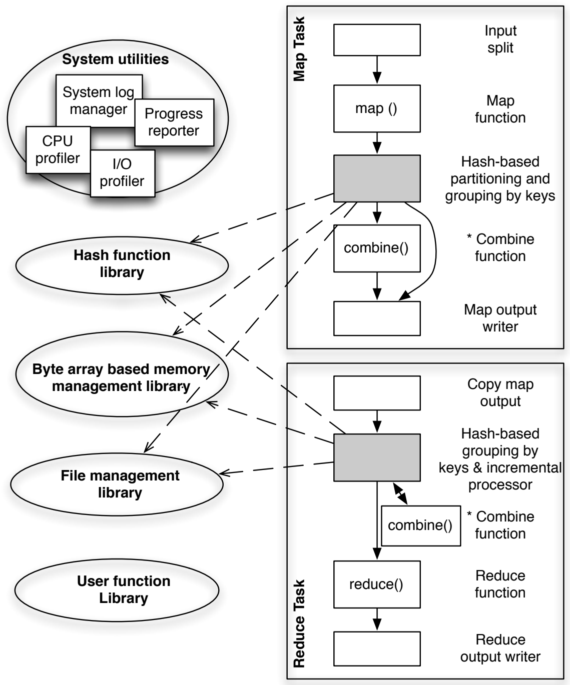


Figure 5. Architecture of our system implementation.

In the reduce module, we implement three hash-based techniques. (1) Our system uses Hybrid Hash [25] to group key-value pairs by key. This method works with or without a combine function, but is still blocking and results in a I/O cost comparable to the sort-merge based implementation in Hadoop. (2) To support incremental computation and reduce I/Os when a combine function is available, we further implement an incremental hash technique, which maintains a state for each key, and updates it incrementally. (3) For the case that the memory cannot hold the states of all the keys, we further optimize the incremental hash by borrowing an existing online frequent algorithm to identify hot keys, and keep hot keys in memory. As the size of a state is usually sublinear in the number of values aggregated, maintaining hot keys instead of random keys in memory results in less I/Os. Moreover, hot keys are typically of greater importance to the users. This technique can return (approximate) results for these keys as early as when all the input data has arrived at the reducers.

Our hash techniques are implemented based on several libraries developed by us for this system. The hash function library provides a set of pair-wise independent hash functions to meet the requirement of hashing techniques. In order to avoid the performance overhead of creating a large number of Java objects, our system implements its the key data structures in byte arrays in the memory management library. The file management library organizes the on-disk data of the hash techniques. The user function library provides some commonly used map and reduce functions. Our system also includes some system utilities such as system log manager, progress reporter, and CPU and I/O profiler.

We conducted an initial experimental comparison between carefully tuned stock Hadoop and our hash-based system. The hash-based system can save up to 48% of CPU cycles, and up to 53% of running time. Furthermore, the I/O cost due to internal data spills in the reduce phase can be reduced by three orders of magnitude when the frequent algorithm is used together with hashing.

## VI. Related Work

**Query Processing using MapReduce** [12], [14], [15], [19], [21], [23], [24], [30], [34] has been a research topic of significant interest lately. To the best of our knowledge, none of these systems support stream processing or one-pass processing of stored data with early approximate answers. The closest work to ours is MapReduce Online [6] which we have discussed in an earlier section.

There have also been several proposals to improve MapReduce performance. These efforts target specific areas of the MapReduce infrastructure, but are not targeted at one-pass analytics. An improved speculative execution strategy [35], for example, will only have a significant impact on the running time of short jobs because only the final wave of tasks is affected. Many of the other proposed techniques can be leveraged in our one-pass analytical system as optimizations.

**Parallel Databases**: Parallel databases in the 80's and 90's [9], [8] required special hardware and lacked sufficient solutions to fault tolerance, hence hard to scale. Emerging large-scale parallel databases [2], [10] have started to employ MapReduce for its benefits of parallelism and fault tolerance. Our research is aligned with these efforts but focuses on one-pass analytics. Hyracks [4] is a new parallel software platform that offers a DAG-based programming model, which is more general than the MapReduce programming model. However, Hyracks does not offer more for efficient incremental computation than Hadoop.

**Data Stream Systems**: Data stream processing has been intensively studied in the database community. Most data stream systems (e.g., [3], [5]) have not considered highly scalable stream processing for massive data sets. Recent work on distributed stream processing has considered a distributed federation of participating nodes in different administrative domains [1], the routing of tuples between nodes [31], and pipelined parallelism for joins [32], but 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**: In the systems community, several parallel stream systems including System S [11], [26], [18], [36], StreamIt [27], [28], [29] and S4 [22] have been developed. These systems adopt a workflow based programming model. These systems usually trade fault-tolerance for performance. Therefore, unlike MapReduce, the output of these systems can be nondeterministic; that is, different execution plans may result in different output. Moreover, these systems leave many systems issues such as memory management and I/O operations to user code, whereas MapReduce systems abstract away these issues in a simple user programming model and handle all the memory and I/O related issues internally within the system.

## VII. Conclusions and Future Work

In this paper, we studied whether it is possible to bring the benefits of the MapReduce computational model to incremental one-pass analytics. We conducted a detailed empirical performance study of Hadoop and MapReduce Online using a variety of workloads and identified several architectural drawbacks of current systems, such as data loading, the CPU bottleneck of sort-merge, and the I/O bottleneck and blocking in multi-pass merge. Based on these insights, we argued for several architectural changes that must be made to MapReduce, such as hash-based group-by, incremental processing, partial aggregation, and full in-memory processing of important keys. Our ongoing work comprises the design of a new parallel streaming system that implements these design changes to efficiently support incremental one-pass analytics using the MapReduce model.

## References

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

[2] A. Abouzeid, K. Bajda-Pawlikowski, et al. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[3] A. Arasu, et al. The CQL continuous query language: semantic foundations and query execution.. VLDB J. , 15(2): 121-142, 2006.

[4] V. Borkar, M. Carey, et al. Hyracks: a flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[5] S. Chandrasekaran, O. Cooper, etal. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[6] T. Condie, N. Conway, et al. Mapreduce online. In *NSDI*, 21–21, 2010.

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

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

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

[10] E. Friedman, P. Pawlowski, et al. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[11] B. Gedik, H. Andrade, et al. SPADE: The System S declarative stream processing engine. In *SIGMOD*, 2008.

[12] P. K. Gunda, L. Ravindranath, et al. Nectar: automatic management of data and computation in data centers. In *OSDI*, 2010.

[13] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 287–298, 1999.

[14] Hive project. http://hadoop.apache.org/hive/.

[15] M. Isard, M. Budiu, et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 59–72, 2007.

[16] C. Jermaine, A. Dobra, et al. A disk-based join with probabilistic guarantees. In *SIGMOD*, 563–574, 2005.

[17] D. Jiang, B.C. Ooi, et al. The Performance of MapReduce: An In-depth Study. In *VLDB*, 472–483, 2010.

[18] V. Kumar, H. Andrade, et al. DEDUCE: at the intersection of MapReduce and stream processing. In *EDBT*, 657–662, 2010.

[19] Y. Kwon, M. Balazinska, et al. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, 2010.

[20] E. Mazur, B. Li, Y. Diao, and P. Shenoy. Towards scalable one-pass analytics using mapreduce. Technical report, UMass Amherst, 2010. http://cs.umass.edu/~mazur/tech-onepassmr.pdf.

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

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

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

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

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

[26] R. Soulé, M. Hirzel, et al. A universal calculus for stream processing languages. In *ESOP*, 2010.

[27] W. Thies, M. Karczmarek, et al. StreamIt: a language for streaming applications. In *CC*, 179–196, 2002.

[28] W. Thies, M. Karczmarek, et al. Teleport messaging for distributed stream programs. In *PPoPP*, 2005.

[29] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, 2010.

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

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

[32] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In *EDBT*, 299–310, 2009.

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

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

[35] M. Zaharia, A. Konwinski, et al. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

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