Reducing MapReduce Abstraction Costs for Text-Centric Applications

Chun-Hung Hsiao  Michael Cafarella  Satish Narayanasamy
University of Michigan, Ann Arbor
{chhsiao,michjc,nsatish}@umich.edu

Abstract—

The MapReduce framework has become widely popular for programming large clusters, even though MapReduce jobs may use underlying resources relatively inefficiently. There has been substantial research in improving MapReduce performance for applications that were inspired by relational database queries, but almost none for text-centric applications, including inverted index construction, processing large log files, and so on.

We identify two simple optimizations to improve MapReduce performance on text-centric tasks: frequency-buffering and spill-matcher. The former approach improves buffer efficiency for intermediate map outputs by identifying frequent keys, effectively shrinking the amount of work that the shuffle phase must perform. Spill-matcher is a runtime controller that improves parallelization of MapReduce framework background tasks. Together, our two optimizations improve the performance of text-centric applications by up to 39.1%. We demonstrate gains on both a small local cluster and Amazon’s EC2 cloud service. Unlike other MapReduce optimizations, these techniques require no user code changes, and only small changes to the MapReduce system.

I. INTRODUCTION

In recent years, the MapReduce programming framework has become extremely popular. There has been a corresponding explosion in the amount of research dedicated to MapReduce, including many studies devoted to improving the system’s performance [1], [7], [9], [11], [12], [15], [17], [19], [20]. Many of these projects use MapReduce to accomplish relational database-style operations, such as selecting a subset of entries from a log file, or joining data from two separate files. However, the class of text-centric MapReduce applications has been largely unexamined.

Text-centric applications operate on large input document corpora. The tasks include performing word counts, computing text statistics, building inverted indexes, and different kinds of natural language processing (NLP). Many of these tasks served as motivation for the original MapReduce designers [6], and they are a critical workload for web data processing.

Unfortunately, text-centric MapReduce programs benefit little or not at all from optimizations designed for relational workloads. Many relational operators (e.g., selection and projection) can ignore effectively huge portions of the input data; their optimizations work by avoiding processing of input that cannot possibly impact the output, and generating small amounts of post-map() data [7], [11], [12]. In contrast, text-centric tasks (e.g., inverted index construction) often aim to transform the entire input set. They generally must process every byte of input and can produce very large intermediate data from the map() function. Text-centric programs must find performance benefits from some other source.

We propose to improve text-centric application performance by reducing the overhead costs of the MapReduce framework — all of the serialization, deserialization, sorting, buffering, and other work that has nothing to do with the user’s code. Unlike some past approaches to MapReduce optimization, we do not require the user to change her code in any way: text-centric applications are likely to use custom parsing libraries or NLP code that would be difficult to port to a new language. Further, we can implement our two optimizations with only modest changes to the MapReduce implementation.

The first, frequency-buffering, exploits the observation that a small subset of keys are seen far more frequently than others in text-centric applications. For example, the frequency of a word in the text corpus of a natural language is inversely proportional to the rank of the word according to Zipf’s law [23]. We can exploit this observation when sorting the map() function’s output. In particular, by dedicating an in-memory buffer for the most frequently occurring keys, we can reduce I/O and sort costs from the MapReduce shuffle phase.

The second optimization, spill-matcher, aims to enable better parallelism within the map phase. It uses a custom parallelism manager to orchestrate multiple parallel threads by dynamically adjusting how often the user’s map() function transmits data for downstream sort processing during the map phase. Sending data too frequently will result in a high work overhead, but sending data too infrequently will result in poor parallelism. Because the CPU intensiveness of map() functions varies by application and by machine, choosing this policy dynamically can yield better results than the standard approach of choosing a static threshold.

Our optimizations are general and should be effective with any MapReduce implementation. In addition to demonstrating their impact on two different Hadoop clusters, we also perform a theoretical analysis to show that our results are not tied to idiosyncrasies of the Hadoop code base. Moreover, we show that our techniques yield large benefits when run on text-centric applications, and either improve or do not substantially change execution times of relational-style jobs.

The primary contributions of this paper include:

• An in-depth diagnosis of MapReduce abstraction costs for text-centric applications, driven by detailed instrumentation of Hadoop. (Section II)

• The frequency-buffering technique, which reduces the quantity of intermediate map() output written to
disk by cheaply finding duplicates among the most-frequent keys, thereby reducing the amount of time taken to sort and combine data. (Section III)

- The spill-matcher technique, which fine-tunes parallelism in the map phase by dynamically controlling the data is moved between users’ map code and MapReduce’s sort/combine procedures. (Section IV)

- Experiments demonstrating the effectiveness of these techniques on two Hadoop clusters: a local 7-node system and a 20-node system running on Amazon’s EC2 service. Combined, the two techniques can eliminate up to 39.1% of a program’s runtime, with absolutely no modification to the user’s program. (Section V)

Finally, we discuss related work in Section VI and conclude with a discussion of future work in Section VII.

II. MAPREDUCE: WHERE DOES THE TIME GO?

A core strength of MapReduce is that the system can parallelize a user’s program at runtime without changing the program semantics. The system can hide much of the complexity associated with large cluster environments, allowing the user to focus on program correctness. However, hiding this complexity also makes it difficult to understand what MapReduce is actually doing internally. Understanding standard MapReduce execution is critical in deciding which optimizations to pursue, but most published research to date describes MapReduce internals at a fairly coarse level.

Thus, before we propose our two optimizations, it is useful to examine exactly what MapReduce currently does, and what changes are likely to yield substantial performance improvements. In this section we present detailed low-level instrumentation results of MapReduce and its Hadoop implementation, then use these findings to motivate our work. But first we review basics of the framework.

A. MapReduce Model and Implementation

The standard MapReduce programming model is described in detail elsewhere [6], so we summarize it briefly. The MapReduce user writes a program that consists of a map() function, a reduce() function and an optional combine() function, plus some helper code to describe object serialization and deserialization. A MapReduce runtime system applies the resulting MapReduce program to a bytestream-oriented file in a distributed filesystem. Figure 1 shows the processing pipeline for MapReduce. Most MapReduce papers refer to three main phases of execution (Table I):

1) In the map phase, the user’s map() function is applied to every ⟨key, value⟩ pair in the input. Each application yields some ⟨key′, value′⟩ pairs. The resulting records on a single node can be sorted and combined by their keys through combine().

2) In the shuffle phase, these ⟨key′, value′⟩ pairs are sorted by key′ for the next phase. No user code is involved; any time spent in shuffle is pure overhead imposed by the MapReduce abstraction. This phase often involves transmitting data over the network.

3) The reduce phase invokes the user’s reduce() function once for every unique key′, passing as an argument the entire associated value′ set. The output of this phase is written to the distributed filesystem.

The overall MapReduce execution is essentially equivalent to a user-written transformation of the inputs (map), followed by a SQL-style GROUP BY on the resulting data (shuffle), followed by a user-written aggregation function (reduce).

Note that although the MapReduce model as described by Dean and Ghemawat [6] entails an actual sort, some user reduce() functions require only a grouping by the intermediate key. For example, it is possible to count the total number of times a URL is observed in a log file using a hash-based grouping mechanism instead of a sort. Indeed, Lin, et al. do not do full sorting at all [15]. However, some MapReduce programs, including many text-centric ones, rely on sort properties. For example, an inverted index must have its keys in sorted output order. Thus, we assume that sorting is a required part of the MapReduce model.

B. Benchmark Applications

The following describes six real benchmark applications we used to examine MapReduce’s behavior. The first three are text-centric. The later three are non-text benchmarks that people also use MapReduce to solve. They are not targeted by our work but useful for comparison’s sake.

WordCount and InvertedIndex: these are typical MapReduce programs presented in [6]. WordCount computes the number of occurrences of each distinct word appears in a text corpus; InvertedIndex constructs, for each word in a corpus, a list of all the locations where the word appears.

WordPOSTag: it performs a part-of-speech (POS) tagging, which is a computation-intensive process used in natural language processing to classify each word in a text corpus as one of several linguistic types such as noun, verb or adjective.

---

1We used the Apache OpenNLP (http://incubator.apache.org/opennlp/) to implement this application.
For each word, map() emits an array of counters, each counts the times this word is of a certain type, and reduce() sums the counters up to get the final POS statistics of all words.

AccessLogSum and AccessLogJoin: they are typical relational tasks similar to Pavlo et al.’s benchmark queries [19]. They processes two input files: the first one (identified below as UserVisit) is a web server access log where each input line represents an HTTP request that contains the destination URL associated ad revenue, and other attributes; the second one (Rankings) is a list of URLs along with their ranking scores. In short, AccessLogSum implements a MapReduce version of the following SQL query:

\[
\text{SELECT destURL, sum(adRevenue)} \\
\text{FROM UserVisits GROUP BY destURL;}
\]

And AccessLogJoin implements the following one:

\[
\text{SELECT sourceIP, adRevenue, pageRank} \\
\text{FROM UserVisits AS UV, Rankings AS R} \\
\text{WHERE UV.destURL = R.pageURL;}
\]

PageRank: it performs a single iteration of the PageRank algorithm [18], over the directed web graph. An input record consists of a \((URL, (pagerank, outlinks))\) pair. The map() function emits two pieces of data: \((URL, (0, outlinks))\) (to reconstruct the graph), plus \((T, (pagerank, outlinks, φ))\) for each outgoing link \(T ∈ outlinks\). The combiner and reducer simply sum ranks for each observed URL.

C. Profiling Results and Design Goals

We now present detailed instrumentation results using our benchmark applications, and how they motivate our proposed optimizations. In this section we break the three MapReduce phases into finer-grained operations summarized in Table I.

1) Sorting and Shuffling: Figure 2 shows where Hadoop MapReduce currently spends its time on the applications. It shows a “serialized” view of the work performed: this data was computed by measuring all the CPU cycles used by any thread on any machine during the job, then grouping by phase, then summing and normalizing. Thus, it does not show the degree of internal MapReduce parallelism: it only shows the entire volume of work that the job must complete.

We first note that execution of actual user-defined code (map() and combine() in the map phase, and the reduce phase) takes a surprisingly small portion of the time for all applications except WordPOSTag. The total only goes over 50% for WordPOSTag and AccessLogJoin. WordPOSTag has a map() function that is extremely computationally intensive; its running time is much larger than the other applications here. For most applications, the majority of the time is spent on work that just supports the MapReduce model itself. Thus, any serious improvement to running time must address the framework overhead, not just user-supplied code.

Another remarkable quality of Figure 2 is that many post-map() operations — emit, sort, I/O, the shuffle phase — have nothing to do with user code, but do scale with the size of the intermediate output emitted by the user’s map() function. Any work we can do to reduce the size of this set should yield benefits across phases and operations. This observation motivates our first optimization, frequency-buffering, which attempts to eliminate as much work as possible from the post-map() operations. We cover this technique in Section III.

2) Map-Side Parallelism: Of course, the total amount of work performed by a job is not the only quality that determines how quickly a job can complete; the degree of possible parallelism is critical. Enabling parallelism between independent map tasks is a central MapReduce design goal and is well known. However, intra-map parallelism is also important: Figure 2 shows that the full map phase accounts for more than 50% of the work in AccessLogJoin and PageRank, and for more than 80% of the work in the other four applications. Any speedup to map will help the entire job’s performance.

Work in a map task can be divided into two distinct parts and executed on multiple threads as follows. One or more threads (referred to as the map threads) repeatedly apply the user’s map() function on the input records and output the results. These output data are transmitted through a shared in-memory buffer to another set of threads (referred to as the support threads), which sort and combine them with the user’s combine() function, and write the results into the local disk. Later, these partially sorted outputs are merged to generate ordered records as the final outcome of the map phase.

The above mechanism is a common parallelization of the map phase and Hadoop currently tries to obtain some intra-map parallelism with a 1-map 1-support implementation and a simple thread-orchestration policy. Table II shows the effectiveness of this technique across our six applications. The map and support threads show poor parallelism in Hadoop, with WordCount showing a map thread idle 38.01% of the time, while the support thread is idle 34.33% of the time. As we discuss in Section IV, controlling data transmission policy between the map and support threads is critical in obtaining

---

**TABLE II. PERCENTAGE OF TIME THAT THE MAP-PHASE MAP AND SUPPORT THREADS ARE IDLE IN SIX APPLICATIONS.**

<table>
<thead>
<tr>
<th>Application</th>
<th>Map, Idle</th>
<th>Support, Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>38.01</td>
<td>34.33</td>
</tr>
<tr>
<td>InvertedIndex</td>
<td>34.86</td>
<td>33.98</td>
</tr>
<tr>
<td>WordPOSTag</td>
<td>0.00</td>
<td>95.14</td>
</tr>
<tr>
<td>AccessLogSum</td>
<td>19.09</td>
<td>58.33</td>
</tr>
<tr>
<td>AccessLogJoin</td>
<td>19.39</td>
<td>54.38</td>
</tr>
<tr>
<td>PageRank</td>
<td>39.78</td>
<td>29.32</td>
</tr>
</tbody>
</table>
good parallelism. Our second optimization uses the spill-matcher to dynamically control the data transmission policy to ensure that at least the slower threads never waste time idling to obtain as much parallelism as possible, while maximizing sort efficiency. spill-matcher was motivated by the extreme variability in CPU-intensity for text-centric applications, but is effective for all of the applications we tested.

### III. Frequency Buffering

In Section II-C1, we observed that the post-map data processing costs could be quite significant (Figure 2). Much of this cost is due to the overhead of sorting records from map(). Our frequency-buffering optimization is motivated by the observation that many text-centric applications, such as inverted index computation and natural language processing, have a skewed frequency distribution for their map()’s output keys. For example, according to Zipf’s law, the frequency of a word in a natural language text corpus is inversely proportional to the rank of the word, as shown in Figure 3. This skewed distribution is perhaps best known in text, but is not limited to it. For example, past research [4] has also shown that the frequency with which web pages are requested from a user community also conforms to a Zipfian distribution.

We first discuss how we can improve local sort-merge efficiency in the presence of a skewed key distribution with known frequent keys. However, the exact frequent key set depends on both the input data and the user’s code. We then discuss our method, which includes a component that dynamically estimates the most-frequent keys for each MapReduce job.

#### A. Frequent Key Dataflow

In the standard MapReduce dataflow, all (key,value) pairs emitted by the user’s map() function are aggregated using an inefficient sort-based GROUPBY operation. As shown in Figure 4, small chunks of map() function’s output are collected in a space constrained in-memory spill buffer, sorted and combine()’d, and then written to disk. These records are read back again from the disk at the end of the map() task, merge-sorted and combine()’d, and then written to disk again for the shuffle phase.

We observe that, because of the skewed key distribution in map() output, the final post-combine() data can be much smaller than map()’s original raw output. Put another way, the system spends a lot of time writing, sorting, and reading data associated with a very small set of keys. If we could store the entire intermediate dataset in an in-memory hash table, we could combine identical keys without the associated I/O and sort costs, but such RAM sizes are generally not practical.

Our frequency-buffering optimization aims to move as much post-map() work as possible into a limited amount of memory, thereby saving I/O and sort costs. It is illustrated in Figure 4. Some tuples emitted by map() have keys that are infrequent and are processed using the standard store-sort-combine dataflow. Other tuples have frequent keys and are stored in a hash table that occupies the limited amount of available RAM. When a new tuple is emitted with a frequent key, our optimization inserts it into the hash table. When the values associated with a single key have hit a space limit, the user’s combine() function is applied to the tuples to aggregate them, which generally yields a single much-smaller tuple. In the case where there is not enough space to store the aggregated record, it is written to disk using the original dataflow. When the entire input is processed, we apply combine() to each key in the hash table, then write each result to disk, again as with the original dataflow. Thus, frequency-buffering effectively utilizes the limited RAM space for combining frequent keys, eliminating as much intermediate data as possible before they are written to disk.

Of course, this design requires that we can distinguish frequent from infrequent output keys before actually generating the output data. For a fixed input data set and a given map() function, we can perform an offline analysis to find frequent keys, but doing so is usually not possible for novel datasets or user code. Thus, we split the frequency-buffering optimization into two stages. In the initial profiling stage, the system observes map() outputs to estimate the frequent set, but all outputs follow the standard MapReduce data path. In the subsequent optimization stage, we use our estimates for the frequent set and send map() outputs either straight to disk or to an in-memory hash table, as described above.

#### B. Finding Frequent Keys

An important component in our frequency-buffering optimization is the profiling method that finds the top-k frequent keys in the map() function’s output stream. We devise our profiling method based on the assumption that the distribution of the keys remains the same through the input to a MapReduce job. This assumption is a heuristic that approximately holds in text-centric applications that operate on a large input data.

Initially, the map task executes without knowledge of any frequent keys. As it processes the input data stream, it keeps track of key frequencies using the algorithm proposed by Metwally, et al. [16]. This algorithm uses a table where each
entry contains a frequency value and a linked list of keys that have been observed for that number of times. When a new key is encountered, and the table is not full, the new key is simply added. If the table is full, one victim key with the lowest frequency is evicted from the table; the new key is inserted with a frequency slightly higher than the lowest frequency in the table to avoid thrashing. Profiling continues until the task has processed a sufficiently large sample to approximate the top-$k$ frequent keys with reasonable accuracy. The sampling fraction $s$ is specified as a percentage of the total input records to the map() function. Once profiling is complete, a hash table is instantiated with $k$ entries, one for each frequent key. We can then proceed with the algorithm as described above.

Of course, the longer this initial key-finding phase lasts, the smaller our chance to actually exploit frequent keys for improved performance; thus, we want the key-finding phase to be as short as possible. We note that if the key distribution does not significantly change across different map tasks within a single job, then it is redundant to profile for the top-$k$ keys in each task. Instead, our system finds the top-$k$ frequent-key set just once for all the tasks that run on a single node; after the first task, the top-$k$ are shared with all subsequent ones.

C. Auto-tuning Profiler

The effectiveness of frequency-buffering relies on several factors: the distribution of keys from map(), the size of $k$, and the sampling fraction $s$. The distribution of keys is fixed by the task and its input. A larger $k$ would allow our optimization to track a larger set of intermediate records, but it is largely fixed by the amount of memory available and the size of intermediate data records. The only parameter our algorithm can adjust is the sampling fraction $s$. However, $s$ is quite important: a too-small $s$ will mean an inaccurate top-$k$, while a too-large $s$ means we lose potential optimization opportunities.

The best value for $s$ depends on the key distribution. For example, if we know the distribution has a tiny number of extremely frequent values, then we can set $s$ to be quite small and still obtain an accurate top-$k$ key set. In contrast, a flatter distribution with many more less-frequent values may require us to obtain more samples to learn it accurately. However, in general the distribution is not known by anyone — even the MapReduce programmer — prior to execution. Therefore, we try to obtain a Zipfian approximation to the key distribution instead — we choose Zipfian here since it is well known that many human activities are Zipfian [4], [23], and also Belevitch [3] had shown that a first-order truncation of many statistical distributions are Zipfian, so it can approximate the key distribution of many text-centric applications.) We introduce a pre-profiling step before starting the profiling method describe in the last section to examine a small fraction of the intermediate records (about 1%) and estimate the Zipfian parameter $\alpha$ to approximate the key distribution. We then use $\alpha$ to determine a value for $s$ as described below.

Let $f_1, f_2, f_3, ...$ be the frequencies of the keys in descending order. With a Zipfian distribution, we have $f_i = C i^{-\alpha}$ for some constant $C$, and taking logarithms on both sides gives us $\log f_i = (-\log i)\alpha + C$, which is a linear equation of unknown coefficients $\alpha$ and $C$. Hence we can estimate $\alpha$ by a linear regression on the log-ranks and log-frequencies of the keys seen in the profiling step. To determine the sampling fraction $s$, let us assume that the keys conform to a Zipfian distribution with parameter $\alpha$. Then we have the probability function

$$P(K \text{ is of rank } i) = p_i = \frac{i^{-\alpha}}{H_{m,\alpha}},$$

where $H_{1,\alpha} = \sum_{j=1}^{i} j^{-\alpha}$. Let $n$ be the total number of the intermediate records output from map(). To find the top-$k$ frequent keys, we hope that the $k^{th}$ frequent key appears in the first $ns$ records. Assume that the keys are independently and identically distributed, then the generation of the intermediate records can be treated as a Bernoulli trial with success probability $p_k$. It is well known that the expected number of trials until a successful outcome is $\frac{1}{p_k} = k^\alpha H_{m,\alpha}$. Then if

$$ns \geq k^\alpha H_{m,\alpha},$$

we can expect at least one occurrence of the $k^{th}$ frequent key. If we choose a larger value for $s$, then the profiling overhead would be higher and also the time period during which frequency-buffering can be applied is reduced. Therefore, the above inequality gives us a reasonable estimation of $s$.

IV. SPILL MATCHER

Our second optimization improves intra-map parallelism by employing a runtime controller called spill-matcher to balance the work of user code and the MapReduce framework.

A. Parallelism Versus Combine Efficiency

To obtain better intra-map parallelism, we can divide the work of the map task between map threads and support threads. For each map task, a set of map threads read input (key, value) pairs, apply the map() function to them and write the intermediate output to an in-memory circular spill buffer. A spill is produced when the spill buffer occupancy exceeds a predetermined threshold, which we refer to as the spill percentage. Once a spill is produced, one or more support threads sort the data in the spill, apply the combine() function to aggregate tuples with identical keys, and then write the final output to an intermediate local file. Once the map and support threads have processed all the input data assigned to their map task, the map threads proceed to merge-sort all the spilled tuples in the intermediate local files and combines tuples with identical keys using the combine() function.

While the support threads are consuming a spill, the map threads can continue to produce the next spill as long as there is space available in the spill buffer. The map threads need to wait if the buffer is full and the support threads have not yet finished processing the earlier spill. The support threads need to wait if it has finished processing a spill and the map threads are yet to produce the next spill.

To attain high efficiency, it important to balance the pipelined parallelism between map and spill threads and reduce their wait time. Unfortunately, as we discussed in Section II-C2, both map threads and support threads spend a significant fraction of their execution time in the idle state.

For a given memory budget, the spill percentage that determines the size of each spill is the critical parameter that determines the efficiency of pipelined parallelism between map and support threads. Setting the spill percentage to a lower value could create smaller spills exposing fine-grained parallelism between the map and support threads reducing the
functions, sizes of input and intermediate combine() this determines the sorting time in the support thread), and the
wait times. However, smaller spills reduce combine efficiency, because in a smaller spill there will be fewer tuples with the same key that can be combined into one tuple. As a result, there will be more tuples written to the intermediate local file
waiting for the faster. For instance, if the map threads are producing tuples faster than the support threads, ideally the support threads should never be made to wait. Thus, the goal of this technique is to
determine the maximum value for the spill percentage that would not cause the slower of the map and support threads in a map task to wait. Given that there could be significant variability in produce rate (p) of the map threads and consume rate (c) of the support threads, our technique adapts the spill percentage at the granularity of a spill in each map task.

To determine the spill percentage for a spill, our runtime technique predicts the expected values for p and c for the next spill based on the observed p and c for the last spill. Our hypothesis is that the input and system characteristics generally remain the same between two adjacent spills, and the producing and consuming rates measured in the last spill would be accurate enough estimations for the rates of the next spill. Also, measuring p and c for each spill incurs negligible overhead when compared to the time spent in processing each spill. In our implementation, we measure time taken (measured in wall clock time) to produce (T_p) and to consume (T_c) a spill, which are inversely proportional to p and c.

Once we have an estimate for p and c for a spill, our next goal is to determine the spill percentage \( x \) for that spill to maximize the spill size without causing the slower threads to incur any wait time. We formally show in the next section that \( x \) can be derived from \( p \) and \( c \) using the following equation:

\[
x = \max \left\{ \frac{c}{p+c}, \frac{1}{2} \right\}.
\]
We implemented spill-matcher in Hadoop’s single-map
thread single-support-thread system and obtained a significant
performance improvement (Section V-C).

C. Deriving Spill Percentage from Produce/Consume Rates

This section describes how to determine the spill percent-
age \( (x) \) for a spill given the producing \( (p) \) and consuming \( (c) \) rates. Our first-order constraint is to ensure that the slower of
the two groups is never made to wait during the spill. The second-order constraint is to maximize the spill size as much
as possible without violating the first constraint.

Let \( M \) be the size of the spill buffer, \( m_i \) be the size of the
\( i \)th spill, \( p, c \) be the produce and consume rates. For the first
spill, we have \( m_1 = xM \). For the subsequent spills, we have
\[
m_i \geq xM \quad \text{for all } i \geq 2.
\]
The \( i \)th spill could be larger if the support threads are still
processing the \((i - 1)\)th spill when its size reaches \( xM \).

Now, we derive the upper limit of \( m_i \). Since the map
threads start to produce the \( i \)th spill right after it notifies the
support threads to consume the \((i - 1)\)th spill, when the
support threads finish consuming the \((i - 1)\)th spill, the map
treads could have produced either \( \frac{p}{x+c} m_{i-1} \) bytes if there is
enough space in the spill buffer, or only \( M - m_{i-1} \) bytes and
completely fill the spill buffer. If the bytes produced is smaller
than \( xM \), then the map threads need to produce more records
in order to reach the spill threshold and make the support threads
wait for the data.

These conditions yield the following equation:
\[
m_i = \max \left\{ xM, \min \left\{ \frac{p}{x+c} m_{i-1}, M - m_{i-1} \right\} \right\}.
\]
Note that if
\[
M - m_{i-1} \geq \frac{p}{x+c} m_{i-1},
\]
then the spill buffer is not full when the \((i-1)\)th spill is
produced, so the map threads can continue the ith spill without
waiting for the support threads to spill the data; if
\[
\min \left\{ \frac{p}{x+c} m_{i-1}, M - m_{i-1} \right\} \geq xM,
\]
then after the support threads finish the \((i-1)\)th spill, the
amount of data produced by the map threads have already
reached the spill threshold, so the support threads can
immediately process the ith spill without any wait.

As we described earlier, to improve parallelism, our first-
order constraint is to ensure that the slower group never has
to wait. If \( p < c \), then the map threads are slower, and (2) and
(3) give the following inequality to make it wait free:
\[
M \geq \frac{1}{x+c} m_{i-1} \geq \frac{1}{x+c} xM \Rightarrow x \leq \frac{p}{p+c}.
\]
If \( p > c \), then the support threads are slower. To make it wait-
free, we have the following inequality from (2) and (4):
\[
xM \leq M - m_{i-1} \leq M - xM \Rightarrow x \leq \frac{1}{2}.
\]
Since \( \frac{p}{p+c} > \frac{1}{2} \) if and only if \( p < c \), we can combine the two
upper bounds into \( x \leq \max \left\{ \frac{c}{p+c}, \frac{1}{2} \right\} \). It is not hard to prove
that this inequality is a sufficient and necessary condition for
ensuring that the slower group is wait free. Given that our
second constraint is to maximize \( x \) as much as possible, we
get the equation (1) shown in the last section.

D. Discussion

spill-matcher adapts the spill percentage to reduce the wait
time of map and support threads while maximizing spill sizes

V. EXPERIMENTS

We now demonstrate the benefits of frequency-buffering
and spill-matcher on six applications. We show that the tech-
niques we have described can reduce a job’s running time
by up to 39.1%, and that all applications show noticeable
improvement. We also explore why some applications benefit
largely from our optimizations, while others only modestly.

A. Experimental Setup

Before we can validate our work from Sections III and IV,
we detail our experimental setup.

1) System Configurations: We ran all experiments on two
Hadoop equipped clusters, one local Hadoop cluster
running a total of 12 mappers and 12 reducers on 6 machines,
with each one equipped with two quad-core 1.86GHz Xeon
processors, 16GB of RAM, and two to four hard disks.

B. Experimental Setup

We ran all experiments on two
Hadoop equipped clusters, one local Hadoop cluster
running a total of 12 mappers and 12 reducers on 6 machines,
with each one equipped with two quad-core 1.86GHz Xeon
processors, 16GB of RAM, and two to four hard disks.

2) Dataset Generation: The six applications involve sev-
eral different inputs: a text corpus, a web server access log,
and a virtual web crawl.

The text corpus we used for WordCount, InvertedIndex, and
WordPOSTag comes from a dump of the Wikipedia in 2008.2
The text has 139.7M lines and fits into a 8.52GB text file.
It contains 1.45B words, but only 24.7M unique ones. The
frequency distribution of the words is shown in Figure 3.

The access logs we used for AccessLogSum and Access-
LogJoin were generated using the tool3 provided by Pavlo,

The accesses logs we used for AccessLogSum and Access-
LogJoin were generated using the tool3 provided by Pavlo,

\[\text{http://static.wikipedia.org/} \]

\[\text{http://database.cs.brown.edu/projects/mapreduce-vs-dbms/} \]
et al. [19]. We use the default parameters and generated a 18.68GB UserVisit log file containing 155M user-visit records for about 600,000 URLs, plus a 33.92MB Rankings table containing ranking scores for each URL. We modified the script to emit URLs according to a distribution that matches real-world observed data, a Zipfian distribution with parameter 0.8, as suggested by Breslau, et al. [4].

The crawl for PageRank is a synthetic graph of 10M pages, computed using the available tools from Pavlo, et al. We used a Zipfian parameter $\alpha = 1$ according to Adamic and Huberman [2]. The web graph is then represented as a list of URLs with their outgoing links, resulting in a 22.89GB file.

For the experiments running on Amazon EC2, we scaled the input data accordingly, so WordCount, InvertedIndex, and WordPOSTag process a text corpus of 50GB, AccessLogSum and AccessLogJoin process an access log of 110GB, and PageRank processes a crawl of 145GB.

| TABLE III. THE OVERALL LOCAL-CLUSTER TIMING RESULTS AFTER APPLYING OUR OPTIMIZATIONS. |
|---------------------------------|---------------|---------------|-------------|
| Text-Centric & WordCount & InvertedIndex & Word*POSTag |
|Baseline & 571s & 816s & 20,170s |
|FreqOpt & 448s (78.4%) & 634s (77.8%) & 20,057s (99.4%) |
|SpillOpt & 449s (78.7%) & 636s (78.0%) & 20,177s (100.0%) |
|Combined & 347s (69.9%) & 536s (65.7%) & 19,781s (98.1%) |
|Other & AccessLogSum & AccessLogJoin |
|Baseline & 203s & 345s & 694s |
|FreqOpt & 198s (97.4%) & 346s (100.3%) & 645s (92.9%) |
|SpillOpt & 196s (96.6%) & 320s (92.7%) & 665s (96.3%) |
|Combined & 194s (95.4%) & 331s (96.0%) & 613s (88.2%) |

### B. Frequency Buffering

Success of the frequency-buffering optimization relies on the system’s ability to correctly predict popular keys in map()’s intermediate output. However, the real benefit yielded by frequency-buffering also depends on other criteria: the volume of data emitted by map(), the cost of serializing and deserializing this data during multiple sort and merge steps, even possibly the cost of comparing two keys while sorting. We examine frequency-buffering’s prediction success for both the text corpus and access log inputs, as well as its performance when applied to the six test applications.

1) Prediction Accuracy: We used the profiling algorithm from Metwally, et al. [16]. However, a realistic space budget is likely smaller than what that technique requires to guarantee that it finds the true top-k keys. Moreover, frequency-buffering will only profile a small percentage of map()’s output. Therefore, we expect that frequency-buffering will only imperfectly predict the most popular keys in the intermediate dataset.

Figure 7 shows the percentage of the intermediate data values that frequency-buffering removes, with $s$ set to 0.1 (that is, the first 10% of input records are profiled). The percentage of records that can be removed from the intermediate set grows with the size of the buffer we use to store the frequent keys. The frequency-buffering algorithm is compared against two other prediction algorithms, Ideal, which has perfect knowledge of the distribution of keys within the intermediate data. LRU always adds each new tuple to the buffer, expelling the least-recently-used key. As can be seen, using frequency-buffering with Metwally, et al.’s predictor misses only about 6% of the records from the text corpus compared with Ideal, and only about 10% of the records in the access log setting. The LRU

2) Performance Improvement: Figure 8 shows a breakdown of the abstraction costs in all the threads of a MapReduce application on our local cluster. The left-hand column in each pair is the baseline standard version of Hadoop, compared with the frequency-buffering version in the right-hand column. For the three text-processing applications, we used $k=3000$ and $s=0.01$. For the log-processing applications, we used $k=10000$ and $s=0.1$. We devoted 30% of the baseline’s spill buffer to frequency-buffering so the total amount of memory allocated to the application is fixed.

As can be seen, frequency-buffering works well when the baseline application spends considerable time during the sort and emit operations: 40% of the abstraction costs are reduced for WordCount, 30% for InvertedIndex, and 45% for WordPOSTag. Emitting substantial intermediate data and thus increasing the time devoted to sort and emit is characteristic of text-centric applications. WordPOSTag also benefits from frequency-buffering, but the user’s map() for WordPOSTag is so CPU intensive that although frequency-buffering makes a large absolute drop in abstraction overhead, it only yields a small percentage decrease in overall execution time.
The two relational applications spend less time in sort and emit operations, and also remove records from the intermediate data less effectively. frequency-buffering removes just shy of 7% of the abstraction costs in AccessLogSum, and it obtains only 3% reduction for AccessLogJoin. PageRank map tasks are similar to text-centric applications in that they both have a relatively large amount of data to sort and a skewed key distribution; thus, frequency-buffering works better on PageRank than the relational applications. However, PageRank has comparatively more reducer-side shuffle work, and the distribution of URLs is not as skewed as that of words in a text corpus, so the total time reduction is not as significant as with text-centric applications.

Note that time spent in the emit operation decreases for text-centric applications, but slightly increases for the log-processing ones. It is due to the small profiling and hashing overhead of frequency-buffering. This overhead undermines the frequency-buffering gains to some extent for all applications, and does so completely for the minor gains in AccessLogJoin.

C. Spill Matcher

We examine the extent to which spill-matcher can remove wait time from the slower map-phase thread. We ran all of our test applications under the baseline system, under spill-matcher alone, under frequency-buffering alone, and under a combined frequency-buffering and spill-matcher setting. The spill percentage under non-spill-matcher settings is set to 0.8 (default setting in Hadoop). In frequency-optimized settings, we devoted 30% of the baseline’s spill buffer to frequent keys.

Figure 9 shows the amount of time spent in the map and support threads for two applications under four different test scenarios. Our adaptive adjustment technique removes most of the time in the slower thread of a map task: comparing the spill-matcher execution with the baseline, about 90% of wait time has been removed for WordCount, 89% for InvertedIndex, 77% for AccessLogSum, and 83% for AccessLogJoin. WordPOSTag has near-zero wait time in its slowest thread, and spill-matcher yields no improvement. spill-matcher is less effective for PageRank, removing only 42% of the wait time. This happens because the produce rate p is approximately the same as the consume rate c; the cases of p > c and p < c can thus both happen, which reduces the margin for spill-matcher’s predictor inaccuracy that can be tolerated without increasing the busier thread’s wait time.

Figure 9 also shows that applying frequency-buffering alone can reduce the wait time of the map thread. The two optimizations interact with each other. Reducing work via frequency-buffering can reduce the sorting load on the support thread, thus reducing the map thread’s wait time. It can also change the produce rate, yielding a higher optimal spill percentage, and reducing the time spent in merge. Thus, frequency-buffering can open opportunities for spill-matcher to exploit.

D. Overall Performance Improvements

Table III shows the overall runtime results of our optimizations applied to all six applications on the local cluster. Unsurprisingly, they are most effective when applied to WordCount, InvertedIndex, and WordPOSTag. Although WordPOSTag does not show a major percentage drop in runtime due to the huge amount of time spent in the user’s map() code, the absolute reduction in time (389 seconds, on average) is even greater than seen with WordCount and InvertedIndex.

Meanwhile, the relational-style AccessLogSum and AccessLogJoin applications show modest improvements. This result is entirely expected, as they generate comparatively little intermediate data compared to their text-centric brethren. This smaller intermediate set requires less time to sort and emit, so frequency-buffering has less data to remove. It also leaves relatively little for the support thread to do, leaving less potential parallelism for spill-matcher to uncover. PageRank shows a slightly better improvement because frequency-buffering works better, as explained in Section V-B2.

Table IV shows the results of running our optimizations on Amazon EC2. The savings on the running time of WordCount and PageRank are similar to those on the small local cluster, proving that the our optimizations can scale to a larger cluster. The improvement of InvertedIndex is not as good as before, due to the larger overhead of transmitting more data between nodes in the shuffle phase.

Figure 10 shows how much time our combined techniques saved on different settings of a synthetic benchmark, SynText. SynText is a parameterizable benchmark that allows us to explore different points in the possible space of text-centric applications. We can vary SynText in terms of CPU-intensity as well as storage-intensity. CPU-intensity is the volume of...
computation performed in \texttt{map()}, as a multiplicative factor over what \texttt{WordCount} performs. Storage-intensity is measured by the average growth in output size when two records are aggregated in \texttt{combine()} or \texttt{reduce()}. The result also provides a reference for how other benchmark applications should be interpreted along these two dimensions. For example, \texttt{WordCount} is both non-CPU-intensive and non-storage-intensive and so appears in the lower-left corner of the diagram. In contrast, \texttt{InvertedIndex} is comparatively storage-intensive and so appears in the upper-left corner. Note that the optimizations are most effective when the level of CPU activity is moderate and when there is significant benefit from applying \texttt{combine()} on intermediate data.

VI. Related Work

Li et al. [14] proposed a technique roughly similar to \textit{frequency-buffering}, but only in the context of an online analytics system that uses hash-based grouping. Zheng et al. [22] also proposed a simplistic frequency-based cache to obtain better memory usage for reduction-based MapReduce implementation on GPUs. Unlike that work, we examine frequency-buffering’s impact on shuffle, evaluate it on a wide range of workloads, propose a framework for analyzing its performance, and give an accurate mechanism for choosing frequent keys and parameters. The problem of matching rates among discrete processing units that we address in \textit{spill-matcher} has some rough similarity to streaming databases [21], but we are unaware of any work in the MapReduce setting.

A skewed key distribution in text-centric applications brings benefits to \textit{frequency-buffering}, but also incurs skew partitions for reduce tasks and impairs MapReduce performance. Kwon et al. [13] proposed an automatic skew mitigation approach to avoid such problem. Researchers have developed work that should help many programs, including text-centric ones. Jiang, et al. identified five broad areas where MapReduce performance could be improved, such as direct I/O [12]. Herodotou, et al. developed a system to jointly optimize a large number of MapReduce system parameters [10]. Their “cost-based optimization” does not explore the space of relational query plans, but rather the MapReduce parameter space.

VII. Future Work and Conclusion

We have described a system for optimizing text-centric MapReduce applications, a class of programs that has been largely ignored despite a recent flurry of MapReduce work. Our two optimizations, \textit{frequency-buffering} and \textit{spill-matcher}, work on a range of text-centric applications and can make such programs run up to 39.1% faster. They even show a modest improvement when run on non-text applications. Finally, these optimizations require no changes to user code and are very simple to implement in a MapReduce framework such as Hadoop. After applying our work, Figure 8 shows that all applications still spend much time in non-user code. These areas could be targeted by different \texttt{post-map()} grouping procedures, by using more efficient on-disk data representations to minimize I/O, or even by traditional “zero-copy” engineering.

VIII. Acknowledgments

We thank the anonymous reviewers for comments that improved this paper. This work is supported by NSF CAREER-1149773 and General Electric.

REFERENCES


