Hourglass: a Library for Incremental Processing on Hadoop

Matthew Hayes
LinkedIn

Sam Shah
LinkedIn

Abstract—**Hadoop enables processing of large data sets through its relatively easy-to-use semantics. However, jobs are often written inefficiently for tasks that could be computed incrementally due to the burdensome incremental state management for the programmer. This paper introduces Hourglass, a library for developing incremental monoid computations on Hadoop. It runs on unmodified Hadoop and provides an accumulator-based interface for programmers to store and use state across successive runs; the framework ensures that only the necessary subcomputations are performed. It is successfully used at LinkedIn, one of the largest online social networks, for many use cases in dashboarding and machine learning. Hourglass is open source and freely available.**

I. Introduction

The proliferation of Hadoop [25], with its relatively easy-to-use MapReduce [6] semantics, has transformed common descriptive statistics and dashboarding tasks as well as large-scale machine learning inside organizations. At LinkedIn, one of the largest online social networks, Hadoop is used for people, job, and other entity recommendations, ad targeting, news feed updates, analytical dashboards, and ETL, among others [23]. Hadoop is used for similar applications at other organizations [11, 24].

A simple example of a descriptive statistic task may be to daily refresh a list of members who have not logged into the website in the past month, which could be displayed in a dashboard or used as an aggregate in other analysis. The naïve implementation is to compute the set difference of all members and those who logged in the past 30 days by devising a job to process the past 30 days of login event data every day, even though the other 29 days of data is static.

Similarly, in machine learning applications, an example of a feature may be impression discounting: dampening recommendations if they are seen but not acted upon. Again, the naïve implementation is for a job to compute impression counts by re-reading and re-computing data from the beginning of time that was already processed in previous runs. Naturally, these tasks could become incremental. However, writing custom code to make a job incremental is burdensome and error prone.

In this work, we describe Hourglass, LinkedIn's incremental processing library for Hadoop. The library provides an accumulator-based interface [26] for programmers to store and use state across successive runs. This way, only the necessary subcomputations need to be performed and incremental state management and its complexity is hidden from the programmer.

For practical reasons, Hourglass runs on vanilla Hadoop. Hourglass has been successfully running in a production scenario and currently supports several of LinkedIn's use cases. It can easily support monoid [15, 21] computations with append-only sliding windows, where the start of the window is fixed and the end grows as new data arrives (for example, the impression discounting case), or fixed-length sliding windows, where the size of the window is fixed (for example, the last login case). We have found that this library supports many of our use cases.

We evaluated Hourglass using several benchmarks over fixed-length sliding windows for two metrics: total task time, which represents the total cluster resources used, and wall clock time, which is the query latency. Using public datasets and internal workloads at LinkedIn, we show that Hourglass yields a 50–98% reduction in total task time and a 25–50% reduction in wall clock time compared to non-incremental implementations.

Hourglass is open source and freely available under the Apache 2.0 license. As far as the authors know, this is the first practical open source library for incremental processing on Hadoop.

II. Related Work

There are two broad classes of approaches to incremental computation over large data sets. The first class of systems provides abstractions the programmer can use to store and use state across successive runs so that only the necessary subcomputations need be performed. Google's Percolator [19] allows transactional updates to a database through a trigger-based mechanism. Continuous bulk processing (CBP) [16] proposes a new data-parallel programming model for incremental computation. In the Hadoop world, HaLoop [3] supplies a MapReduce-like programming model for incremental computation through extending the Hadoop framework, adding various caching mechanisms and making the task scheduler loop-aware. Hadoop online [4] extends the Hadoop framework to support pipelining between the map and reduce tasks, so that reducers start processing data as soon as it is produced by mappers, enabling continuous queries. Nova [18] is a workflow manager that identifies the subcomputations affected by incremental changes and produces the necessary update operations. It runs on top of

Pig [17], a data-flow programming language for Hadoop, and externalizes its bookkeeping state to a database.

The second class of approaches are systems that attempt to reuse the results of prior computations transparently. DyradInc [20] and Nectar [8] automatically identify redundant computation by caching previously executed tasks in Dyrad [10]. Incoop [1] addresses inefficiencies in task-level memoization on Hadoop through incremental addition support in the Hadoop filesystem (HDFS) [22], controls around task granularity that divide large tasks into smaller subtasks, and a memoization-aware task scheduler. Slider [2] allows the programmer to express computation using a MapReduce-like programming model by assuming a static, unchanging window, and the system guarantees a sliding window. Approaches in this class are currently limited to research systems only.

Our approach borrows techniques from systems in the first class to accommodate incremental processing atop Hadoop. As Hourglass is not changing the underlying MapReduce layer in Hadoop, it does suffer from well-known inefficiencies that many of the described systems are attempting to address. Efficient incremental processing of large data sets is an active area of research.

III. Incremental Model

A. Problem Definition

Hourglass is designed to improve the efficiency of sliding-window computations for Hadoop systems. A *sliding-window* computation uses input data that is partitioned on some variable and reads only a subset of the data. What makes the window *sliding* is that the computation usually happens regularly and the window grows to include new data as it arrives. Often this variable is time, and in this case we say that the dataset is *time-partitioned*. In this paper we focus on processing time-partitioned data, however the ideas extend beyond this.

Consider a dataset consisting of login events collected from a website, where an event is recorded each time a user logs in and contains the user ID and time of login. These login events could be stored in a distributed file system in such a way that they are partitioned by day. For example, there may be a convention that all login events for a particular day are stored under the path /data/login/yyyy/mm/dd. With a partitioning scheme such as this, it is possible to perform computations over date ranges. For example, a job may run daily and compute the number of logins that occurred in the past 30 days. The job only needs to consume 30 days worth of data instead of the full data set.

Suppose that the last login time for each user was required. Figure 1 presents two iterations of a MapReduce job producing this information from the login event data. Without loss of generality, the view is simplified such that there is one map task per input day and one reduce task per block of output.

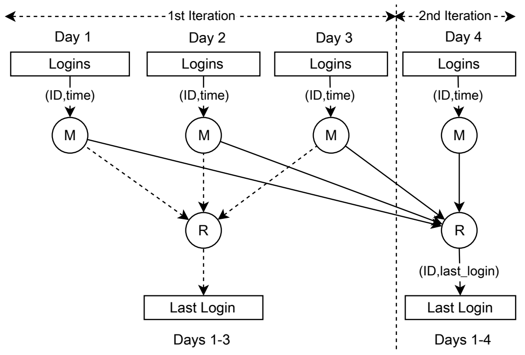


Figure 1. Computing the last login time per user using MapReduce. The input data is partitioned by day. Each map task (M) extracts pairs (ID,login) representing each login event by a user. The reducer (R) receives pairs grouped by user ID and applies max() to the set of login times, which produces the last login time for each user over the time period. It outputs these last login times as (ID,last_login) pairs. The first iteration consumes days 1-3 and produces the last login time per user for that period. The second iteration begins when day 4 data is available, at which time it consumes all 4 days of available data. Consecutive days share much of the same input data. This suggests it may be possible to optimize the task by either saving intermediate state or using the previous output.

Computing the last login time in this way is an example of what we will call an *append-only sliding window* problem. In this case, the start of the window is fixed and the end grows as new data becomes available. As a result, the window length is always increasing.

One inefficiency present in this job is that each iteration consumes data that has been processed previously. If the last login time per user is already known for days 1-3, then this result could be used in place of the input data for days 1-3. This would be more efficient because the output data is smaller than the input data. It is this type of inefficiency that Hourglass addresses.

As another example, suppose there is a recommendation system that recommends items to users. Each time items are recommended to a user, the system records an event consisting of the member ID and item IDs. Impression discounting, a method by which recommendations with repeated views are demoted in favor of unseen ones, is applied to improve the diversity of recommendations. With this in mind, Figure 2 presents three iterations of a MapReduce job computing the impression counts for the last three days. This is similar to the last-login case in Figure 1 except that the input window is limited to the last three days instead of all available data.

Computing the impression counts in this way is an example of what we will call a *fixed-length sliding window problem*. For this type of problem the length of the window is fixed. The start and end of the window both advance as new data becomes available.

As with the previous example, the impression counting job presented in Figure 2 is inefficient. There is significant overlap of the input data consumed by consecutive executions of the job. The overlap becomes greater for larger window sizes.

The inefficiencies presented here are at the core of what Hourglass attempts to address. The challenge is to develop a

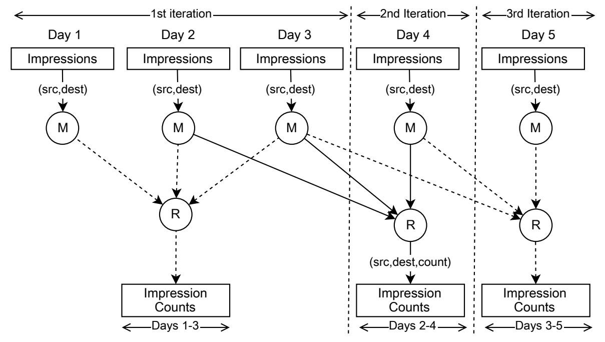


Figure 2. Computing (src,dest) impression counts over a three day sliding window using MapReduce. The input data is partitioned by day. Each map task (M) extracts (src,dest) pairs from its input data. The reducer (R) counts the number of instances of each (src,dest) pair and outputs (src,dest,count). The first iteration consumes days 1-3 and produces counts for that period. The second iteration begins when day 4 data is available, at which time it consumes the most recent 3 days from this point, which are days 2-4. When day 5 data is available, the third iteration executes, consuming days 3-5. Consecutive days share much of the same input data. This suggests it may be possible to optimize the task by saving intermediate data.

programming model for solving these problems efficiently, while not burdening the developer with complexity.

B. Design Goals

This section describes some of the goals we had in mind as we designed Hourglass.

Portability. It should be possible to use Hourglass in a standard Hadoop system without changes to the grid infrastructure or architecture. In other words, it should use out-of-the-box components without external dependencies on other services or databases.

Minimize Total Task Time. The total task time refers to the sum of the execution times of all map and reduce tasks. This represents the compute resources used by the job. A Hadoop cluster has a fixed number of slots that can execute map and reduce tasks. Therefore, minimizing total task time means freeing up slots for other jobs to use to complete work. Some of these jobs can even belong to the same workflow. Minimizing total task time can therefore contribute to greater parallelism and throughput for a workflow and cluster.

Minimize Execution Time. The execution time refers to the wall clock time elapsed while a job completes. This should include all work necessary to turn input data into output data. For example, Hourglass can produce intermediate data to help it process data more efficiently. A MapReduce job producing such intermediate data would be included here. While minimizing job execution time is a goal, in some cases it might be worth trading off slightly worse wall clock time for significant improvements in total task time. Likewise, wall clock time for an individual job might be worse but the overall wall clock time of a workflow might be improved due to better resource usage.

Efficient Use of Storage. Hourglass might require additional storage in the distributed file system to make processing more efficient. There are two metrics that we should be concerned with: total number of bytes and total number of files. The number of files is important because the Hadoop distributed file system maintains an index of the files it stores in memory on a master server, the NameNode [22]. Therefore, it is not only a goal to minimize the additional bytes used, but also the file count.

C. Design Assumptions

There are a few assumptions we make about the environment and how Hourglass might be used to solve problems:

Partitioned input data. We assume that the input data is already partitioned in some way, which is the common method for storing activity data [11, 23, 24]. Without loss of generality, we will assume the data is time-partitioned throughout this paper.

Sliding window data consumption. Input data is consumed either through a fixed-length sliding window or an append-only sliding window. Supporting variable length windows is not a use case we have encountered.

Mutability of input data. We do not assume immutability of input data. While an assumption such as this might mean that more efficient designs are possible for an incremental system, it is a hard assumption to make given the complexity and fallibility of large distributed systems. Systems sometimes have errors and produce invalid data that needs to be corrected.

Immutability of own data. While we do not assume the immutability of input data, we do assume that any intermediate or output data produced by Hourglass will not be changed by any other system or user.

D. Our Approach

In this section we will present our approach to solving *append-only sliding window* and *fixed-length sliding window* problems through MapReduce.

1) Append-only Sliding Window: First we introduce the concept of a *partition-collapsing job*, which reads partitioned data as input and merges the data together, producing a single piece of output data. For example, a job might read the last 30 days of day-partitioned data and produce a count per key that reflects the entire 30 day period.

Figure 3 presents an example of a partition-collapsing job. Here three consecutive blocks of data for three consecutive days have been collapsed into a single block. More formally, a *partition-collapsing* job takes as input a set of *time-consecutive* blocks $I^{[t_1,t_2)}, I^{[t_2,t_3)}, \cdots, I^{[t_{n-1},t_n)}$ and produces output $O^{[t_1,t_n)}$, where $t_i < t_{i+1}$. In Figure 3, blocks $I^{[1,2)}, I^{[2,3)}, I^{[3,4)}$ are processed and $O^{[1,4)}$ is produced.

Figure 3 can be used for the append-only sliding window problem, but it is inefficient. One of the fundamental weaknesses is that each execution consumes data that was previously processed. Suppose that the reduce step can be represented as a sequence of binary operations on the values of a

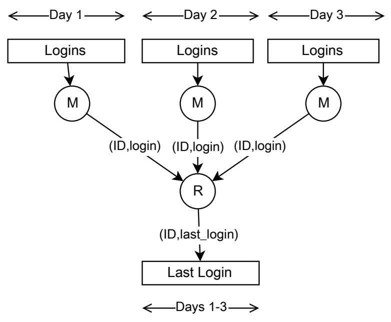


Figure 3. An example of a partition-collapsing job. The job consumes three consecutive days of day-partitioned input data and produces a block of output data spanning those three days. This particular job consumes login events partitioned by day. Each map task outputs (ID,login) pairs representing the time each user logged in. The reducer receives a set of login times for each ID and applies max() to determine the last login time for each user, which it outputs as (ID,last_login) pairs. The job has therefore collapsed three consecutive day-partitioned blocks of data into a single block representing that time span.

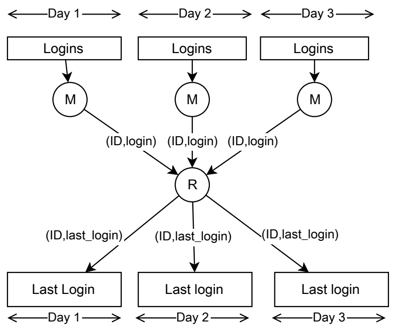


Figure 4. An example of a partition-preserving job. The job consumes three days of day-partitioned input data and produces three days of day-partitioned output data. Here the reducer keeps the input data partitioned as it applies the reduce operation. As a result the output is partitioned by day as well. The (ID,last_login) pairs for a particular day of output are only derived from the (ID,login) pairs in the corresponding day of input.

particular key: $a \oplus b \oplus c \oplus d$. Assuming the reducer processes these values in the order they are received, then the operation can be represented as $(((a \oplus b) \oplus c) \oplus d)$. However, if the data and operation have the *associativity* property, then the same result could be achieved with $(a \oplus b) \oplus (c \oplus d)$. This means that one reducer could compute $(a \oplus b)$, a second reducer could compute $(c \oplus d)$, and a third could apply $\oplus$ to the two resulting values. If the intermediate results are saved, then the computations do not need to be repeated. When new data e arrives, we can compute $(a \oplus b) \oplus (c \oplus d) \oplus e$ without having to recompute $(a \oplus b)$ and $(c \oplus d)$. This is the same principle behind memoization [5], which has been applied to intermediate data produced in Hadoop for other incremental systems [2].

An example of a job applying this principle is presented in Figure 4. We refer to this as a *partition-preserving* job. Here the reducer maintains the partitions from the input data as it applies the reduce operation. As a result, the output is partitioned by day as well. This achieves the same result as running a separate MapReduce job on each day of input without the scheduling overhead.

More formally, a partition-preserving job takes as input a set of time-partitioned blocks $I^{[t_1,t_2)}$, $I^{[t_3,t_4)}$, $\cdots$, $I^{[t_{n-1},t_n)}$ and

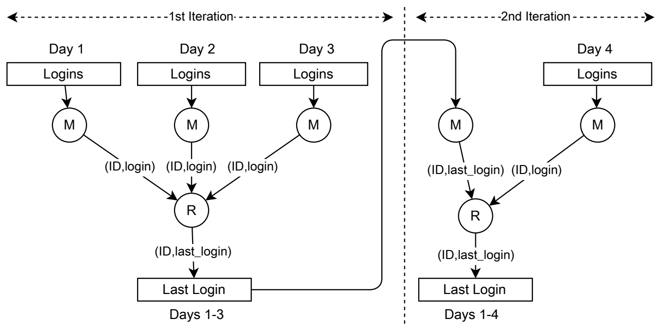


Figure 5. An example of a *partition-collapsing* job solving the append-only sliding window problem by reusing previous output. Here the first iteration has already produced the last login times for each user for days 1-3. The second iteration uses this output instead of consuming the input data for days 1-3.

produces time-partitioned output $O^{[t_1,t_2)}$, $O^{[t_3,t_4)}$, $\cdots$, $O^{[t_{n-1},t_n)}$, where $O^{[t_i,t_j)}$ is derived from $I^{[t_i,t_j)}$.

Partition-preserving jobs provide one way to address the inefficiency of the append-only sliding window problem presented in Figure 1. Assuming the last login times are first computed for each day as in Figure 4, the results can serve as a substitute for the original login data. This idea is presented in Figure 6.

One interesting property of the last-login problem is that the previous output can be reused. For example, given output $O^{[t_{i-1},t_i)}$, the output $O^{[t_i,t_{i+1})}$ can be derived with just $I^{[t_i,t_{i+1})}$. This suggests that the problem can be solved with a single partition-collapsing job that reuses output, as shown in Figure 5. This has two advantages over the previous two-pass version. First, the output data is usually smaller than both the input data and the intermediate data, so it should be more efficient to consume the output instead of either. Second, it avoids scheduling overhead and increased wall clock time from having two sequentially executed MapReduce jobs.

Two techniques have been presented for solving the append-only sliding window case more efficiently. One uses a sequence of two jobs, where the first is partition-preserving and the second is partition-collapsing. The second uses a single partition-collapsing job with feedback from the previous output.

2) Fixed-length Sliding Window: Similar to the append-only sliding window case, this problem can be solved using a sequence of two jobs, the first partition-preserving and the second partition-collapsing. The idea is no different here except that the partition-collapsing job only consumes a subset of the intermediate data. This has the same benefits as it did for the append-only sliding window problem.

For append-only sliding windows it was shown that in some cases it is possible to apply an optimization where only the single partition-collapsing job is used. If the previous output can be reused, then the partition-preserving job can be dropped. In some cases a similar optimization can be applied to fixed-length sliding windows. The idea is presented in Figure 7 for a 100 day sliding window on impression counts. Here the previous output is used and combined with the

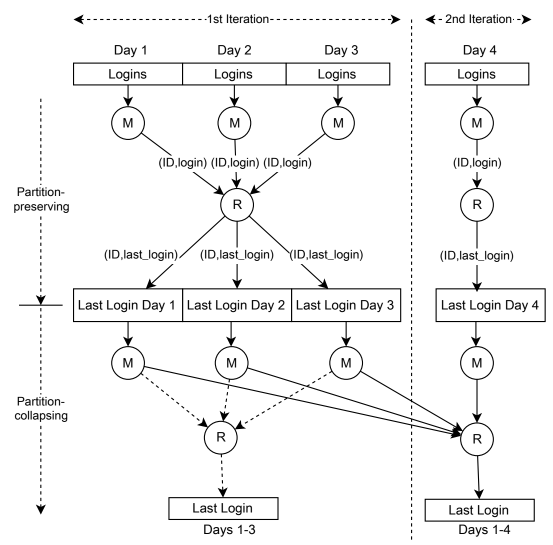


Figure 6. An example of an append-only sliding window computation of the last login time per user through the use of a partition-preserving job followed by a partition-collapsing job. The first job's map task reads login events from day-partitioned data and outputs (ID,login) pairs representing the login times for each user. The reducer receives the login times per user but maintains their partitioning. It computes the last login time for each day separately, producing day-partitioned output. The second job's map task reads in the last login time for each user for each of the days being consumed and sends the (ID,last_login) to the reducer grouped by ID. The reducer applies max() to the login times to produce the last login time over the period. For the first iteration, the first pass processes three days of input data and produces three days of intermediate data. For the second iteration it only processes one day of input data because the previous three have already been processed. The second pass for the second iteration therefore consumes one block of new data and three blocks that were produced in a previous iteration.

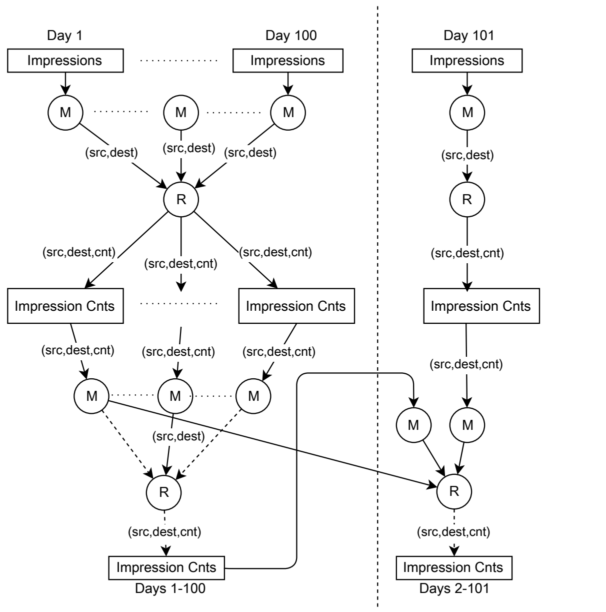


Figure 7. An example of a 100 day sliding window computation of impression counts through the use of a partition-preserving job followed by a partition-collapsing job. The first job's map task reads impressions from day-partitioned data and outputs (src,dest) pairs representing instances of src being recommended dest. The reducer receives these grouped by (src,dest). It maintains the partitioning and computes the counts of each (src,dest) separately per day, producing day-partitioned output. For the first iteration, the second pass consumes 100 days of intermediate data. Each map task outputs (src,dest,cnt) pairs read from the intermediate data. The reducer receives these grouped by (src,dest) and sums the counts for each pair, producing (src,dest,cnt) tuples representing the 100 day period. For the second iteration, the first pass only needs to produce intermediate data for day 101. The second pass consumes this new intermediate data for day 101, the intermediate data for day 1, and the output for the previous iteration. Because these are counts, arithmetic can be applied to subtract counts from day 1 from the counts in the previous output, producing counts for days 2-100. Adding counts from the intermediate data for day 101 results in counts for days 2-101.

newest day of intermediate data; however, in addition, the oldest day that the previous output was derived from is also consumed so that it can be subtracted out. This still requires two jobs, but the partition-collapsing job consumes far less intermediate data.

E. Programming Model

One of the goals of Hourglass is to provide a simple programming model that enables a developer to construct an incremental workflow for sliding window consumption without having to be concerned with the complexity of implementing an incremental system. The previous section showed that it is possible to solve append-only and fixed-length sliding window problems using two job types:

Partition-preserving job. This job consumes partitioned input data and produces output data having the same partitions. The reduce operation is therefore performed separately on the data derived from each partition so that the output has the same partitions as the input.

Partition-collapsing job. This job consumes partitioned input data and produces output that is *not partitioned* – the partitions are essentially collapsed together. This is similar to a standard MapReduce job; however, the partition-collapsing job can reuse its previous output to improve efficiency.

Consider some of the implications these features have on the mapper. For the reducer of the partition-preserving job to maintain the same partitions for the output, some type of identifier for the partition must be included in the key produced by the mapper. For example, an impression $(src, dest)$ would have a key $(src, dest, pid)$, where pid is an identifier for the partition from which this $(src, dest)$ was derived. This ensures that `reduce` only operates on $(src, dest)$ from the same partition. The partition-collapsing job can reuse its previous output, which means that the previous output has to pass through the mapper. The mapper has to deal with two different data types.

There are implications for the reducers too. For the partition-preserving job, the reducer must write multiple outputs, one for each partition in the input data. For the partition-collapsing job, the reducer must not only perform its normal reduce operation but also combine the result with the previous output.

Hourglass hides these details from the developers so they can focus on the core logic, as would normally express in a standard MapReduce job. It achieves this by making some changes to the MapReduce programming model.

First, let us review the MapReduce programming model [6, 14], which can be expressed functionally as:

- **map:** $(v_1) \rightarrow [(k, v_2)]$
- **reduce:** $(k, [(v_2)]) \rightarrow [(v_3)]$

```
function MAP($impr$)
    EMIT($impr$, 1)                                   ▷ $impr \equiv (src, dest)$
end function
function REDUCE($impr, counts$)
    $sum \leftarrow 0$
    for $c$ in $counts$ do
        $sum = sum + c$
    end for
    $output \leftarrow (impr.src, impr.dest, sum)$
    EMIT($output$)
end function
```

Figure 8. Impression counting using a traditional MapReduce implementation. The mapper emits each (src,dest) impression with a count of 1 for the value. The reducer is *iterator-based*. It receives the counts grouped by (src,dest) and sums them to arrive at the total number of impressions for each (src,dest).

The map takes a value of type v_1 and outputs a list of intermediate key-value pairs having types k and v_2. The reduce function receives all values for a particular key and outputs a list of values having type v_3.

Figure 8 presents an example implementation for counting (src,dest) impressions using MapReduce. The map function emits (src,dest) as the key and 1 as the value. The reduce function receives the values grouped by each (src,dest) and simply sums them, emitting (src,dest,count).

This example uses an *iterator-based* interface for the reduce implementation. In this approach, an interface representing the list of values is provided to the user code. The user code then *iterates* through all values present in the list. An alternative to this is the *accumulator-based* interface, which has the same expressiveness as the iterator-based interface [26]. An example of the accumulator-based approach is shown in Figure 9.

Next, we will present how the programming model differs in Hourglass. Hourglass uses an accumulator-based interface for the reduce implementation. Additionally, the functional expression of reduce is slightly different from that of general MapReduce:

- **map:** $(v_1) \rightarrow [(k, v_2)]$
- **reduce:** $(k, [(v_2)]) \rightarrow (k, v_3)$

The map function here is the same as in the MapReduce programming model. The reduce function is less general because it can output at most one record and each record must consist of a key-value pair. In fact, the key k is implicitly included in the output of the reducer by Hourglass so the user code only needs to return the output value. An example of a finalize implementation is shown in Figure 10.

The map operation retains the same functional definition because Hourglass hides the underlying details and only invokes the user's mapper on input data. The mapper for the partition-collapsing job passes the previous output to the reducer without user code being involved. Likewise, the mapper for the partition-preserving job attaches the partition identifier to the output of the user's map function before sending it to the reducer.

The reduce operation differs principally because the partition-collapsing job is more efficient if it reuses the previous output. Reusing the previous output implies that it

```
function INITIALIZE()
    return 0
end function
function ACCUMULATE($sum, impr, count$)
    return $sum + count$
end function
function FINALIZE($impr, sum$)
    $output \leftarrow (impr.src, impr.dest, sum)$
    EMIT($output$)
end function
```

Figure 9. Impression counting using an *accumulator-based* interface for the reduce operation. To sum the counts for a particular (src,dest) pair, initialize is called first to set the initial sum to zero. Then, accumulate is called for each count emitted by the mapper, where for each call, the current sum is passed in and a new sum is returned. When all counts have been processed, the final sum is passed to the finalize method, which emits the output.

```
function FINALIZE($impr, sum$)
    return $sum$
end function
```

Figure 10. A simplified finalize method for the accumulator-based interface used in Hourglass. Here, finalize does not need to return the *src* and *dest* values because they are implicitly paired with the return value.

must pass through the mapper. By forcing the output to be in the form (key,value), it is possible to pass the data through the mapper without the developer having to implement any custom code. Otherwise, the developer would have to implement a map operation for the previous output as well, making the implementation more complicated and exposing more of the underlying details of the incremental code. This conflicts with the goal of having a simple programming model.

To reuse the previous output, Hourglass requires that a merge operation be implemented if it is an append-only sliding window job. The fixed-length sliding window job in addition requires that an unmerge operation be implemented in order to reuse the previous output.

- **merge:** $(v_3, v_3) \rightarrow (v_3)$
- **unmerge:** $(v_3, v_3) \rightarrow (v_3)$

These functions take two parameters of type v_3, the output value type of the reducer function. merge combines two output values together. unmerge effectively is an *undo* for this operation. Given an output value, it can *subtract* another output value from it.

Figure 11a shows an example of merge for the last login problem described previously. Given the previous last login and the last login for the new set of data just processed, it computes the maximum of the two and outputs this as the new last login.

Figure 11b shows an example of merge and unmerge for computing impression counts. Given the previous output count and the count from the new intermediate data, merge sums them to produce a new count. Using this count and the oldest intermediate count corresponding to the previous window, unmerge subtracts the latter from the former to produce the count over the new window.

F. Capabilities

Hourglass can be used to incrementalize a wide class of sliding window problems. Recall that sliding window

```
function MERGE($prev\_last\_login, new\_last\_login$)
    $last\_login = max(prev\_last\_login, new\_last\_login)$
    return $last\_login$
end function
```

(a)

```
function MERGE($prev\_count, new\_count$)
    $curr\_count = prev\_count + new\_count$
    return $curr\_count$
end function
function UNMERGE($curr\_count, old\_count$)
    $curr\_count = curr\_count - old\_count$
    return $curr\_count$
end function
```

(b)

Figure 11. Examples of `merge` and `unmerge` functions for computing (a) last-login, and (b) impression counts.

problems have the property that the input data is partitioned and the computation is performed on a consecutive sequence of these partitions. We can express the reduce operation as $reduce(x_i + \!\!+ \, x_{i+1} + \!\!+ \ldots + \!\!+ \, x_j)$, where x_i is the list of map output data derived from one of the input partitions and $+\!\!+$ represents concatenation. If the reduce operation can be represented as an associative binary operation $\oplus$ on two data elements of type M, then the previous reduce computation can be replaced with the equivalent $reduce(x_i) \oplus reduce(x_{i+1}) \oplus \cdots \oplus reduce(x_j)$. Assuming that $\oplus$ has the closure property and that an identity element $i_\oplus$ also exists , then together $(M, \oplus)$ form a *monoid* [15, 21].

Splitting the reduce operation in this way translates directly to the first and second passes described earlier for Hourglass, where the first pass is partition-preserving and the second pass is partition-collapsing. The first pass produces partial results and saves these as intermediate data. The second pass computes the final result from the intermediate data. A binary operation $\oplus$ with identity $i_\oplus$ is easily expressible using an accumulator-based interface. Therefore, if the reduce operation for a sliding-window problem can be represented using a monoid, then it can be incrementalized as two passes with Hourglass. Either type of sliding-window problem can be incrementalized this way.

There are many problems that can be expressed using monoid structures. For example, integers along with any of the min, max, addition, and multiplication operations form monoids. Average can also be computed using a monoid structure. There are also many approximation algorithms that can be implemented using monoid structures, such as Bloom filters, count-min sketches, and HyperLogLog [13].

Assuming the reduce operation can be represented as a monoid consisting of $(M, \oplus)$, then the `merge` operation described earlier can also be represented using the same monoid with binary operation $\oplus$. This means that an append-only sliding window job can be implemented with just a single partition-collapsing job, as `merge` enables reuse of the previous output, making an intermediate state unnecessary.

Recall that for the fixed-length sliding window, the second pass partition-collapsing job can only reuse the previous output if an `unmerge` operation is implemented. Unfortunately, having the monoid property does not by itself mean that the `unmerge` operation can be implemented. However, if the monoid also has the invertibility property, then the monoid is actually a group and `unmerge` can easily be implemented from `merge` by inverting one of the two elements. For example, for the addition of integers, we can define $merge(x, y) \to x + y$. Using the invertibility property, we can define $unmerge(x, y) \to x - y$. Therefore, if the reduce operation for a fixed-length sliding window problem can be represented using a group, the problem can not only be incrementalized using two passes, but the second pass partition-collapsing job can reuse the previous output.

Addition and multiplication of integers and rational numbers form a group. It is also possible to compute the average using a group structure. This makes Hourglass well suited for certain counting and statistics jobs operating over fixed-length sliding windows.

IV. Evaluation

Four benchmarks were used to evaluate the performance of Hourglass. All used fixed-length sliding windows. The first benchmark evaluated aggregation of impression data from a Weibo recommendation training set [12] on a local single-machine Hadoop 1.0.4 installation. The remaining three benchmarks were run on a LinkedIn Hadoop 1.0.4 grid that has hundreds of machines. The second benchmark evaluated aggregation of impressions collected on the LinkedIn website for the "People You May Know" (PYMK) feature, which recommends connections to members. The third evaluated aggregation of page views per member from data collected on the LinkedIn website. The fourth evaluated cardinality estimation for the same page view data.

Two metrics were collected for each benchmark. *Total task time* is the sum of all map and reduce task execution times. It represents the amount of compute resources used by the job. *Wall clock time* is the execution time of the job from the time setup begins until the time cleanup finishes. Because Hourglass uses two MapReduce jobs for fixed-length sliding windows, the metrics for the two jobs were summed together.

A. Weibo Impressions Benchmark

The Weibo recommendation training data [12] consists of a set of item recommendations. These were partitioned by day according to their timestamp, producing data spanning a month in time. For this benchmark we evaluated several lengths of sliding windows over the data. In addition we evaluated Hourglass with and without output reuse enabled in order to evaluate its impact. A simple MapReduce job was also created as a baseline. The mapper and accumulator used for Hourglass are shown in Figure 12a and Figure 12b.

Figure 13a presents a comparison of the total task time as the sliding window advanced one day at a time. Although the 7 day window results for the Hourglass jobs are intermittently

```
public static class Mapper extends AbstractMapper {
   private final GenericRecord key, value;
   public Mapper() {
     key = new GenericData.Record(KEY_SCHEMA);
     value = new GenericData.Record(VALUE_SCHEMA);
     value.put("count", 1L);
   }
   public void map(GenericRecord record,
                KeyValueCollector collector)
      throws IOException, InterruptedException {
     key.put("src", (Long)record.get("userId"));
     key.put("dest",(Long)record.get("itemId"));
     collector.collect(key, value);
   }
}
```

(a)

```
public static class Counter implements Accumulator {
  private long count;
  public void accumulate(GenericRecord value) {
    count += (Long)value.get("count");
  }
  public boolean finalize(GenericRecord newValue) {
    if (count > 0) {
      newValue.put("count", count);
      return true; // true means output record
    }
    return false; // false means do not output record
  }
  public void cleanup() {
    this.count = 0L;
  }
}
```

(b)

Figure 12. Weibo task Java implementations in Hourglass for (a) the mapper and (b) the combiner and reducer accumulators. The key is (src,dest) and the value is the count for that impressed pair.

better and worse, for larger window lengths, the Hourglass jobs consistently perform better after the initial iteration. The job that reuses the previous output performs best; for the 28 day window it yields a further 20% reduction on top of the 30% reduction already achieved.

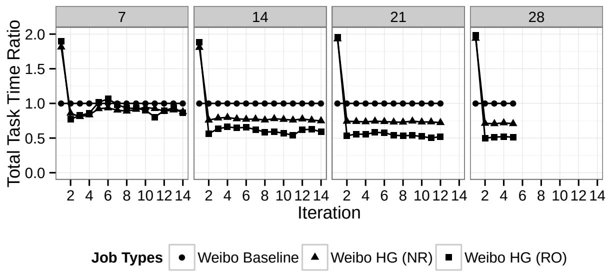
Figure 13b presents a comparison of the wall clock time for the three jobs across multiple iterations. While the wall clock time is worse for the 7 day window, there is a trend of improved wall clock time as the window length increases. For a 28 day window, the wall clock time is reduced to 67% for the job which reuses the previous output.

The size of the intermediate data storing the per-day aggregates ranged in size from 110% to 125% of the final output size. Therefore, it slightly more than doubles the storage space required for a given piece of output data.

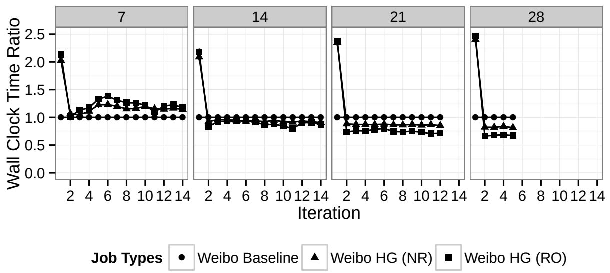
B. PYMK Impressions Benchmark

"People You May Know" (PYMK) is a recommendation system at LinkedIn that suggests connections to members. To improve the quality of its recommendations, it tracks which suggestions have been shown. This data is recorded as a sequence of (src,destIds) pairs, partitioned by day.

The task for this benchmark was to count (src,dest) impression pairs over a 30 day sliding window. This is very similar to the task of the previous benchmark. However, we found that flattening the data into (src,dest) pairs was very inefficient for this data set, as it increased the number of records significantly. Therefore, the data was kept grouped by src and the output value was a list of (dest,count) pairs.



(a)



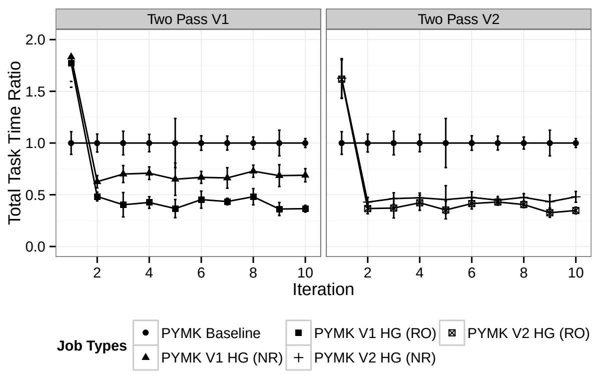
(b)

Figure 13. Comparing (a) the total task time, and (b) the total wall clock time of two Hourglass jobs against a baseline MapReduce job for the Weibo task. Values were averaged over three runs and were normalized against baseline. One Hourglass job reuses the previous output (RO) and the other job does not (NR). Fixed-length sliding windows of 7, 14, 21, and 28 days are shown. Reusing output generally performed better. Total task time for the first iteration is roughly twice that of baseline's; however, for larger window sizes it is significantly smaller, reaching 50% of baseline for the 28 day window. Wall clock time is also larger for the first iteration. For the 7 day window it remains worse than baseline for subsequent iterations, however for the 21 and 28 day windows it is significantly smaller, reaching 67% that of baseline's wall clock time for the 28 day window.

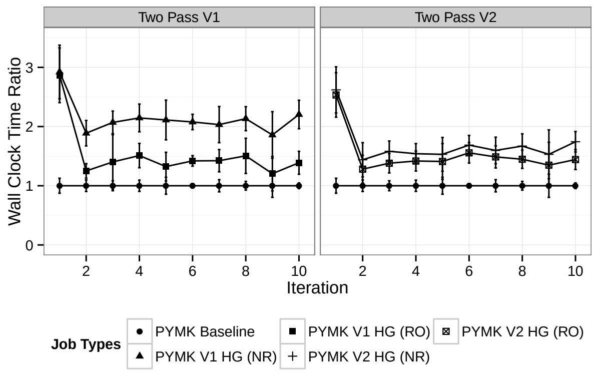
A basic MapReduce job was created for a baseline comparison. The mapper for this job was an identity operation, producing the exact destIds read from each input record. The combiner concatenated dest IDs together into one list and the reducer aggregated these lists to produce the count per dest ID.

There were two variations of Hourglass jobs created for this benchmark. This being a fixed-length sliding window problem, a partition-preserving job served as the first pass and a partition-collapsing job served as the second pass. For the first variation, the partition-preserving job did not perform any count aggregation; the combiner and reducer each produced a list of dest IDs as their output values. All count aggregation occurred in the reducer belonging to the partition-collapsing job.

In the second variation, the partition-preserving job performed count aggregation in the reducer. This made it similar to the basic MapReduce job, except that its output was partitioned by day. The partition-collapsing job was no different from the first variation except for the fact that it consumed counts which were already aggregated by day.



(a)



(b)

Figure 14. Comparing (a) the total task time, and (b) the total wall clock time for Hourglass jobs against a baseline MapReduce job for the PYMK task using a 30 day fixed-length sliding window. Values have been normalized against the baseline. The *Two Pass V1* variation stores a list of dest IDs in the intermediate state as the value. The *Two Pass V2* variation instead stores a list of (dest,count) pairs, therefore performing intermediate aggregation. Both variations were evaluated when previous output was reused (RO) and when it was not reused (NR). For total task time in (a), there was not a significant difference between the two variations when reusing output (RO). The total task time for each variation averaged about 40% of baseline's. For *Two Pass V2*, the job which does not reuse output (NR) is improved over the corresponding *Two Pass V1* variation. It improved so much that it was almost as good as the version which reuses output (RO). For the wall clock time in (b), the Hourglass jobs consistently have a higher wall clock time than baseline. The output reuse cases (RO) perform better than the cases that do not reuse output (NR), with about a 40% higher wall clock time than baseline.

Figure 14a presents comparisons of the total task time for the two variations against the baseline MapReduce job. The Hourglass jobs consistently perform better for both variations. The job that reuses output performs best, showing a clear improvement over the job that does not reuse output. For the second variation, the performance of the two jobs is similar. This implies that, considering total task time alone, after the intermediate data is aggregated there is only a small benefit in reusing the previous output.

Figure 14b presents comparisons of the wall clock times for the two variations against the baseline MapReduce job. Unlike total task time, for this metric, the Hourglass jobs consistently performed worse. Between the two, the variation that reused output performed best. This is different than the results of the Weibo benchmarks, where Hourglass improved both metrics for large window sizes. However, the difference between the Weibo and PYMK benchmarks is that the latter ran on a cluster with hundreds of machines, which allowed for better parallelism. In this particular case there is a tradeoff between total task time and wall clock time.

It is worth considering though that reducing total task time reduces load on the cluster, which in turn improves cluster throughput. So although there may be a tradeoff between the two metrics for this job in isolation, in a multitenancy scenario, jobs might complete more quickly as a result of more compute resources being made available.

C. Page Views Benchmark

At LinkedIn, page views are recorded in an event stream, where for each event, the member ID and page that was viewed is recorded [23]. For this benchmark we computed several metrics from this event stream over a 30 day sliding window. The goal was to generate, for each member over the last 30 days, the total number of pages viewed, the total number of days that the member visited the site, and page view counts for that member across several page categories. Examples of page categories include “profile”, “company”, “group”, etc. Computing aggregates such as these over fixed-length sliding windows is a very common task for a large-scale website such as LinkedIn.

As the task computes a fixed-length sliding window, partition-preserving and partition-collapsing Hourglass jobs were used. The key used was the member ID and the value was a tuple consisting of a page view count, a days visited count, and a map of page category counts. To evaluate the performance of Hourglass, a baseline MapReduce job was also created.

Figure 15 compares the performance of Hourglass against the baseline job. As in previous examples, the first iteration of Hourglass is the most time consuming. Although total task time for the first iteration was about the same as baseline's, wall clock time was twice as much. For subsequent iterations, however, total task time and wall clock time were substantially lower than baseline's. Total task time was between 2% and 4% of baseline's; wall clock time was between 56% and 80% of baseline's.

D. Cardinality Estimation Benchmark

The page views data from the previous example can also be used to determine the total number of LinkedIn members who accessed the website over a span of time. Although this number can be computed exactly, for this benchmark we use the HyperLogLog algorithm [7] to produce an estimate. With HyperLogLog, one can estimate cardinality with good accuracy for very large data sets using a relatively small amount of space. For example, in a test we performed using a HyperLogLog++ implementation [9] that includes improvements to the original HyperLogLog algorithm, cardinalities in the hundreds of millions were

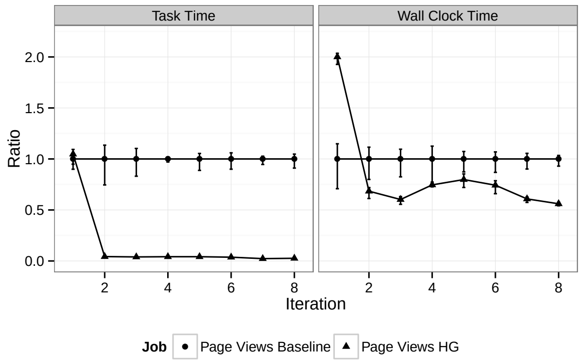


Figure 15. Comparing the Hourglass jobs against a baseline MapReduce job for the page views aggregation task. The window length used was 30 days. Several runs were performed for each iteration. The minimum values of total task time and wall clock time across all runs are plotted, normalized by the baseline value of each iteration. Total task time for the first iteration of Hourglass is about 5% greater than baseline's, within the margin of error. Subsequent iterations, however, have total task times which are in the range of 2% and 4% of baseline's, which reflects a substantial reduction in resource usage. Wall clock time for the first iteration of Hourglass is about double that of baseline's, a result of having to run two MapReduce jobs sequentially. However, subsequent iterations have wall clock times between 56% and 80% of baseline's.

estimated to within 0.1% accuracy using only about 700 kB of storage. HyperLogLog also parallelizes well, making it suitable for a sliding window [7]. For this benchmark an implementation based on HyperLogLog++ was used.

We estimated the total number of members visiting the LinkedIn website over the last 30 days using a sliding window which advanced one day at a time. Since only a single statistic was computed, the key is unimportant; we used a single key with value "members". The value used was a tuple of the form (data,count). The count here is a member count. The data is a union type which can be either a long value or a byte array. When the type is a long it represent a single member ID, and therefore the corresponding count is 1L. When the type is byte array it represents the serialized form of the HyperLogLog++ estimator, and the count is the cardinality estimate. The advantage of this design is that the same value type can be used throughout the system. It can also be used to form a monoid.

The implementation of the mapper and accumulator is straightforward. The mapper emits a tuple (memberId,1L) for each page view event. The accumulator utilizes an instance of a HyperLogLog++ estimator. When the accumulator receives a member ID, it offers it to the estimator. When it receives a byte array it deserializes an estimator instance from the bytes and merges it with the current estimator. For the output it serializes the current estimator to a byte array and includes the cardinality estimate.

The partition-preserving job used this mapper and in addition used the accumulator for its combiner and reducer. The partition-collapsing job used an identity mapper which passed the data through unchanged. It also used the accumulator for its combiner and reducer. Using the accumulator in the combiner greatly reduces the amount of data which has to be sent from the mapper to the reducer. A basic MapReduce

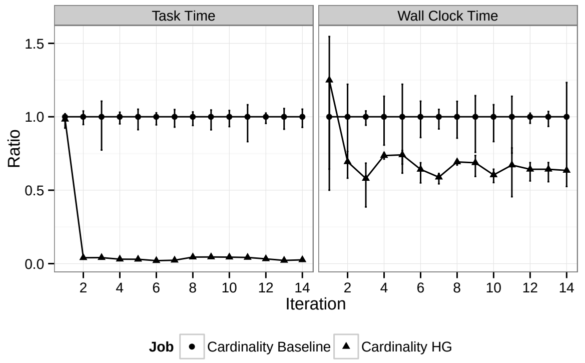


Figure 16. Comparing the Hourglass jobs against a baseline MapReduce job for the cardinality estimation task. Several runs were performed for each iteration. The minimum values of total task time and wall clock time across all runs are plotted, normalized by the baseline value of each iteration. Total task time for the first iteration of Hourglass is roughly the same as baseline's, within the margin of error. Subsequent iterations, however, have total task times which are in the range of 2% and 5% of baseline's, which reflects a substantial reduction in resource usage. Wall clock time for the first iteration of Hourglass is about 25% greater than that of baseline's. However, subsequent iterations have wall clock times between 58% and 74% of baseline's.

job was also created to establish a baseline for comparison. It used the same mapper and accumulator implementations. The HyperLogLog++ estimator was configured with a m value of 20 bits, which is intended to provide an accuracy of 0.1%.

Figure 16 compares the performance of Hourglass against the baseline job for a 30 day sliding window computed over a period of 14 days. Similar to the previous benchmark, total task time is significantly reduced for subsequent iterations. For this benchmark it is reduced to between 2% and 5% of baseline's. This reflects a significant reduction in resource usage. Wall clock times for subsequent iterations is reduced as well by a significant amount.

V. Conclusion

In this paper we presented Hourglass, a framework for efficiently processing data incrementally on Hadoop by providing an easy accumulator-based interface for the programmer. We evaluated the framework using several benchmarks over fixed-length sliding windows for two metrics: total task time, representing the cluster resources used; and wall clock time, which is the query latency. Using real-world use cases and data from LinkedIn, we show that a 50–98% reduction in total task time and a 25–50% reduction in wall clock time are possible compared to baseline non-incremental implementations. Hourglass is in use at LinkedIn and is freely available under the Apache 2.0 open source license.

References

[1] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *SoCC*, pages 7:1–7:14, 2011.

[2] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental sliding-window computations for large-scale data analysis. Technical Report 2012-004, MPI-SWS, 2012.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The HaLoop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169–190, Apr. 2012.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, 2010.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

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

[7] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, (1), 2008.

[8] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI*, pages 1–8, 2010.

[9] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[11] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at Twitter. *Proc. VLDB Endow.*, 5(12):1771–1780, Aug. 2012.

[12] Y. Li and Y. Zhang. Generating ordered list of recommended items: a hybrid recommender system of microblog. *arXiv preprint arXiv:1208.4147*, 2012.

[13] J. Lin. Monoidify! monoids as a design principle for efficient mapreduce algorithms. *CoRR*, abs/1304.7544, 2013.

[14] Y. Liu, Z. Hu, and K. Matsuzaki. Towards systematic parallel programming over mapreduce. In *Euro-Par 2011 Parallel Processing*, pages 39–50. Springer, 2011.

[15] Y. Liu, K. Emoto, and Z. Hu. A generate-test-aggregate parallel programming library. 2013.

[16] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, pages 51–62, 2010.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[18] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous Pig/Hadoop workflows. In *SIGMOD*, pages 1081–1090, 2011.

[19] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 1–15, 2010.

[20] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: reusing work in large-scale computations. In *HotCloud*, 2009.

[21] D. Saile. Mapreduce with deltas. Master's thesis, Universitt Koblenz-Landau, Campus Koblenz, 2011.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[23] R. Sumbaly, J. Kreps, and S. Shah. The "Big Data" ecosystem at LinkedIn. In *SIGMOD*, 2013.

[24] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, pages 1013–1020, 2010.

[25] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2010.

[26] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.