C-MR: Continuously Executing MapReduce Workflows on Multi-Core Processors

Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, Uğur Çetintemel
Department of Computer Science
Brown University
{backman,karthikp,rfonseca,ugur}@cs.brown.edu

ABSTRACT

The widespread appeal of MapReduce is due, in part, to its simple programming model. Programmers provide only application logic while the MapReduce framework handles the logistics of data distribution and parallel task management.

We present the Continuous-MapReduce (C-MR) framework which implements a modified MapReduce processing model to continuously execute workflows of MapReduce jobs on unbounded data streams. In keeping with the philosophy of MapReduce, C-MR abstracts away the complexities of parallel stream processing and workflow scheduling while providing the simple and familiar MapReduce programming interface with the addition of stream window semantics.

Modifying the MapReduce processing model allowed us to: (1) maintain correct stream order and execution semantics in the presence of parallel and asynchronous processing elements; (2) implement an operator scheduler framework to facilitate latency-oriented scheduling policies for executing complex workflows of MapReduce jobs; and (3) leverage much of the work that has gone into the last decade of stream processing research including: pipelined parallelism, incremental processing for both Map and Reduce operations, minimizing redundant computations, sharing of sub-queries, and adaptive query processing.

C-MR was developed for use on a multiprocessor architecture, where we demonstrate its effectiveness at supporting high-performance stream processing even in the presence of load spikes and external workloads.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – query processing, parallel databases

General Terms

Design, Performance, Experimentation

Keywords

Stream processing, MapReduce, Multi-core

1. INTRODUCTION

MapReduce [5] has become quite popular since its debut, largely due to its simplistic programming model and automatic handling of parallelization, scheduling, and communication for distributed batch processing. Hiding these intricacies lowers the barrier to entry for application programmers to begin large-scale data processing.

Recently, there has also been interest in leveraging MapReduce to continuously process unbounded streams of data [4, 6, 8]. Such streams include event logs, click streams, image/video streams, network traffic, and various other data feeds. Stream applications are often time-critical such that their utility is proportional to the promptness of the results (e.g., network intrusion or fraud detection) thus latency minimization is an objective of many stream applications.

Enabling stream support for MapReduce jobs is simple for Map operations (which will continuously consume and emit data) but requires the addition of the *window* stream construct for Reduce operators as they expect to process a finite collection of data. Windows are temporal subdivisions of a stream described by their *size* (the amount of the stream they span) and their *slide* (the interval between windows) to provide a basis for analyzing data across the dimension of time (eg., determining the frequency of a trending web-search over the past hour at 5-minute intervals).

While continuously executing a single MapReduce job can done with relative ease (by invoking a MapReduce job repeatedly over newly arriving windows), the challenge is achieving low-latencies when continuously scheduling complex workflows of MapReduce jobs. Batch-oriented workflows of MapReduce jobs can be executed in a serialized and bottom-up manner (from inputs to outputs), but continuously executing workflows requires a great deal of coordination between jobs as data will never stop arriving and results must be produced continuously. The continuous scheduling of such complex workflows has not been previously addressed by continuous MapReduce frameworks.

We developed the Continuous-MapReduce framework (C-MR), for use on multiprocessor architectures, to address these needs while leveraging the benefits of both MapReduce and data stream management systems (DSMS). C-MR uses the simplicity and flexibility of the MapReduce programming model and adapts the MapReduce processing model to define and execute complex stream workflows. We also leverage principles common to existing DSMSs [2] to utilize latency- and resource-aware operator scheduling policies, incremental processing for Map and Reduce operators, and stream sharing for common sub-workflows.

1.1 Related Work

The Hadoop Online Prototype [4] (HOP) supports stream processing for a single MapReduce job by running Map threads and Reduce threads continuously with data pipelined between them. HOP supports only primitive, non-overlapping windows constructed from each node's local wall-clock time and can not guarantee stream order preservation.

In-situ MapReduce (iMR) [8] uses the MapReduce programming interface to deploy a single MapReduce job onto an existing DSMS where the inputs are read only from disk. The DSMS allows for count- or time-based sliding windows, pipelining between Map and Reduce operations, and in-network, multi-level aggregation trees for Reduce operations.

IBM's Deduce [6] modularizes the functionality of MapReduce into an operator within a DSMS. This operator consumes a delimited list of files/directories (each tuple likened to a window definition) to invoke MapReduce on. Therefore, The stream is a layer of indirection to execute MapReduce jobs where the scheduling of resources for the batch and stream processing workloads are separate. Also, the burden of window management is placed on the application developer to insert window definitions into the stream and removes the possibility for well known stream processing optimizations such as incremental processing and the reduction of redundant computations in overlapping windows.

1.2 Contributions

In contrast to such previous work, we focused on creating a continuous MapReduce framework suitable for developing real-world stream applications which have low-latency objectives, stream integrity requirements, and are generally represented as workflows of operators.

In summary, we make the following contributions:

- **Presenting the C-MR framework and programming interface** for creating and executing complex workflows of continuous MapReduce jobs on unbounded data streams.

- **Automatic stream order preservation and window management** in the presence of parallel and asynchronous stream processors.

- **A workflow-wide operator scheduling framework** which supports the progressive transition between scheduling policies (based on resource-availability) to meet application objectives.

- **The application of classic query optimizations to MapReduce workflows**, including sub-query sharing, incremental sub-window processing, and adaptive query processing.

2. C-MR PROGRAMMING INTERFACE

In defining the C-MR programming interface, we made a conscious effort to mimic the original MapReduce interface so that porting applications between frameworks would be trivial. Only minor interface differences are present and application logic need not be modified. Our framework is written in C++ and supports sliding window definitions, connections to input/output streams, the creation of complex workflows of MapReduce jobs, and the sharing of common sub-workflows. An example workflow of continuous MapReduce jobs, as specified by the C-MR programming interface, might look like the one seen in Figure 1.

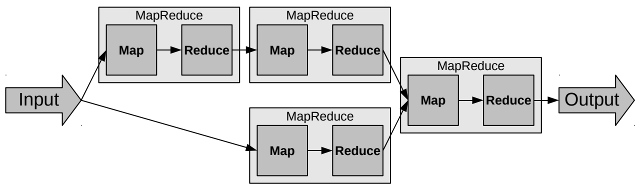


Figure 1: An example C-MR workflow.

Like batch-oriented MapReduce jobs, the continuous MapReduce jobs in C-MR are defined by a Map operation and a Reduce operation which the application programmer specifies. Similarly to Google's MapReduce [5], classes deriving from `Map` and `Reduce` superclasses implement corresponding `map` and `reduce` functions to facilitate application logic. These functions may then produce results using the provided `emit` function. The function signatures for the MapReduce interface are defined in Table 1.

void **map**(void* key, int keySize, void* val, int valSize, timeval timestamp);
void **reduce**(void* key, int keySize, vector<void*> val, vector<int> valSize, timeval timestamp);
void **emit**(void* key, int keySize, void* val, int valSize, timeval timestamp);

Table 1: MapReduce interface function prototypes

For a MapReduce job to continuously process data and produce results, it must be attached to input and output streams. The input and output streams of the workflow must also be defined by the application programmer to insert data into the workflow and to make use of the results. These streams are represented by user-defined functions. Input streams are associated with a file descriptor (e.g., standard input, TCP socket connection, opened file) and the input function will continually fetch key/value information from the stream and return the results (encapsulated into the `Data` format) to the workflow. Stream output functions will continually receive and handle `Data` as specified by the application programmer. Examples of these function formats can be found in Table 2.

Data*	myStreamInputFunc(FILE* inStream) { . . . }
void	myStreamOutputFunc(Data* data) { . . . }

Table 2: Stream input/output function formats

To create complex workflows of continuous MapReduce jobs, we define a `Query` for which we add inputs, outputs, and intermediate MapReduce operations. Each intermediate operator and output takes a unique identifier to direct data to them. Similarly, each input and intermediate operator defines the number of downstream locations it will forward data to and then lists those locations by their unique identifiers. The `addInput` function takes an input stream, input parsing function, and a list of attached operators as input. The `addMapReduce` function defines a unique ID, instances of a pair of `Map` and `Reduce` subclasses (contained within a `MapReduce` object), an instance of the `Window` class (specifying the window size and window slide of the Reduce operation), and a list of downstream locations to deliver data to. The `addOutput` function simply denotes its unique ID and the user-defined function that will handle the results. These function signatures are specified in Table 3. Similarly, C-MR supports the addition of individual Map or Reduce operators to a Query but the `addOperator` function signatures have been omitted due to space constraints.

void Query::**addInput**(FILE* stream, Data* (*inputFunc)(FILE*), int numOpsConnected, ...);
void Query::**addMapReduce**(uint16_t id, MapReduce mapReduce, Window window, int numOutputs, ...);
void Query::**addOutput**(uint16_t id, void (*outputFunc)(Data*));

Table 3: Workflow creation interface function prototypes

A standard MapReduce job consumes a set of input key/value pairs and produces a set of output key/value pairs. The intermediate data produced by Map and consumed by Reduce are a set of keys with corresponding lists of values. Since the input and output of the MapReduce job can also be represented in this way, we define the input and output schema of each workflow operator in C-MR to similarly consume and produce a set of keys with corresponding lists of values. Therefore, the output of any MapReduce job can be sent to any other MapReduce job with the impetus on the programmer to process the data accordingly.

3. C-MR ARCHITECTURE

C-MR allows for continuous execution of complex MapReduce workflows on a multi-processor architecture. We define each independent processing element on a computer (i.e., processor, core) to be a *computing node* capable of executing any defined Map or Reduce operation.

C-MR computing nodes have a different execution strategy than is seen in MapReduce. Traditionally, computing nodes receive a set of Map or Reduce tasks and each node must wait for all other nodes to complete their tasks before being allocated additional tasks. C-MR uses pull-based data acquisition allowing computing nodes to execute any Map or Reduce workload as they are able. Thus, straggling nodes will not hinder the progress of the other nodes if there is data available to process elsewhere in the workflow. Nodes are instead free to fetch data from the shared memory buffers and can do so at their respective rates (given the

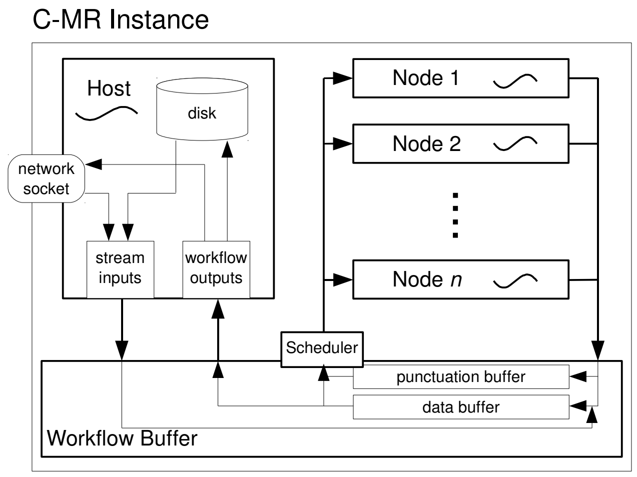


Figure 2: The host invokes a node thread for each local processor/core and then manages the workflow inputs and outputs while the nodes asynchronously execute data from the workflow buffer as directed by the scheduler.

possibilities of external workloads). We have outlined the physical architecture of our system in Figure 2 and describe its components below.

Host Process. A computer running an instance of C-MR will launch a C-MR `Host` process. This process is responsible for determining the number of available processors/cores on the computer and launching computing node threads for those we elect to use. The `Host` will then instantiate the workflow operators – specified by the application programmer – from which nodes will find code to process data according to the application. With the workflow instantiated as a directed acyclic graph of Map and Reduce operations, the `Host` will attach input streams to the workflow.

Workflow Buffer. Intermediate data in the workflow are stored in a shared-memory staging area, known as the `Workflow Buffer`. It is here that data are materialized into windows and/or wait to be consumed by computing nodes. To ensure that the temporally aware Reduce operators consume an ordered stream, we collect and sort the data here prior to window materialization.

To maintain a sorted stream as we materialize windows for Reduce operations, we insert punctuations [10] into the intermediate workflow buffers. The punctuations denote window boundaries within the stream and are inserted at intervals corresponding to a window's slide value and at a location where the stream is already ordered (e.g., at the input to the workflow or at an upstream sorting point). When a node fetches a punctuation from the stream, we replicate the punctuation and issues a copy to each node. Nodes will continue processing their data and pass any replicas they find downstream. Once all replicas are received downstream at the `Workflow Buffer`, we have the guarantee that all data relevant to the applicable window has arrived and may be materialized. We elaborate on this procedure in Section 4.1.

Scheduler. A `Scheduler` routine acts as the interface between `Nodes` and the `Workflow Buffer` when a `Node` requests data to process. The `Scheduler` may use any operator scheduling technique, such as those defined in Sections 4.2 and 4.3, to determine which data from the `Workflow Buffer` a `Node` should process next.

Additionally, the `Scheduler` moderates the volume of data that a `Node` consumes on each request. In this way we batch a set of data for a specific operator that the `Node` can process in sequence. This assists `Nodes` in minimizing context switches to improve cache utilization. We empirically found a batch size that we've applied to all intermediate streams in the workflow. We hope to, in the future, automatically derive appropriate batch sizes for stream segments, which may vary over time, but such an optimization is currently beyond the scope of this paper.

Generic Computing Nodes. The `Host` launches a `Node` thread for each CPU core on the computer that is "generic" in that it is capable of processing data on behalf of any workflow operator. Each thread is confined to run on its own core via the Linux `sched_setaffinity` function to prevent the operating system from scheduling `Node` threads on the same computing resource. `Node` threads will continually attempt to fetch data to process from the `Workflow Buffer` as directed by the `Scheduler`. When data are available, they will be processed with respect to their corresponding operations with the results forwarded downstream. A `Node` will give priority to handling replicated punctuations as they arrive and push them back into the `Workflow Buffer` to encourage the materialization of downstream windows and thereby free up additional data to be processed.

4. IMPLEMENTATION DETAILS

With the C-MR architecture in place, we implemented procedures to facilitate stream ordering with window management and resource-aware operator scheduling for workflows of MapReduce operators.

4.1 Stream and Window Management

Windows, by definition, consist of temporally contiguous data and can therefore only be formed onto sorted streams. Streams that are delimited by data-parallel operations pose a complication because the merged output streams are not guaranteed to retain their original orderings. We address this problem by replicating window-bounding *punctuations* [10] once they are retrieved by a node from an operator's input buffer. A replica is sent to each of the active computing nodes to maintain the property that the window is bounded on each parallel stream. Once the nodes have returned all replicas to the merged output stream, which suffices to bound the window within the merged stream, we can materialize the window with its relevant data so that it can be processed.

An illustration of this entire process is depicted in Figure 3. Note that punctuations are only necessary for windowed (Reduce) operators and that they can only be inserted into sorted streams. Therefore, to allow for punctuation insertion, we require application inputs to be sorted by either system or application timestamp. When windows are materialized, and the stream is subsequently re-ordered, punctuations can be inserted yet again to preserve order for a downstream operator and so on for any additional downstream operators.

As an alternative to requiring sorted inputs, one could employ a commonly used technique to tolerate tuple delays until a threshold period of "slack" time has passed at which point punctuation insertion would occur. However, this goal is not pertinent to the focus of our paper.

4.2 Operator Scheduling

To determine which operator a node should execute, C-MR uses a scheduling framework that enacts one of any number of scheduling policies that may be defined. The framework was developed with the goal of being a "progressive" scheduling framework, capable of executing multiple policies simultaneously and transitioning between those policies based on resource availability. Our progressive scheduler, which we expand on in section 4.3, draws from the following basic pair of scheduling policies.

1. **Oldest Data First (ODF):** Schedule the operator that has the data with the oldest timestamp.
2. **Best Memory Trade-off (MEM):** Schedule the operator that is expected to free up the most amount of memory.

These policies evaluate each operator, giving it a rank, to determine which will be the best candidate for a node to consume data from. Ranks are represented as a signed integer with preference being given to operators with higher ranks. Operator ranks are only compared to other ranks assigned by the same policy. Ranks produced by differing policies need not be comparable as these policies are executed and compared in isolation of each other. Table 4 defines the methods that the policies use to rank an individual operator.

	Ranking Expression applied to Operator
ODF	-(timestamp of data at front of queue)
MEM	$\frac{\text{avgInputSize-(selectivity)(avgOutputSize)}}{\text{avgProcessingTime}}$

Table 4: Operator scheduling policies

ODF uses the negated timestamp of the data at the front of an operator's queue, resulting in old data (with lower timestamps) earning higher ranks. MEM observes the expected difference between the average size of an input tuple and the average size of an output tuple for an operator with consideration of the operator's selectivity. This difference (the expected decrease in memory usage) is then divided by the processing time to assess the expected number of bytes reduced per unit of time.

While these scheduling policies themselves are not particularly novel, our scheduling framework, in conjunction with the use of generic processing nodes, allows for two substantial advantages. The first advantage is that these nodes will now have the opportunity to execute the operator with the highest priority in the workflow. Thus, generic processing nodes are able to, at each step, schedule the data that is most beneficial to the current scheduling objective.

4.3 Progressive Scheduling

The second advantage afforded to us by our scheduling framework is the ability to enact different scheduling policies on each node's request for data. The benefits for this were not immediately obvious but, with further investigation, we were able to develop hybrid scheduling strategies which utilize multiple scheduling policies simultaneously. This can be beneficial when certain precious system resources become scarce. To this effect, we instantiated a hybrid, latency-oriented and memory-aware scheduling policy that aims to progressively become more memory-efficient as the available

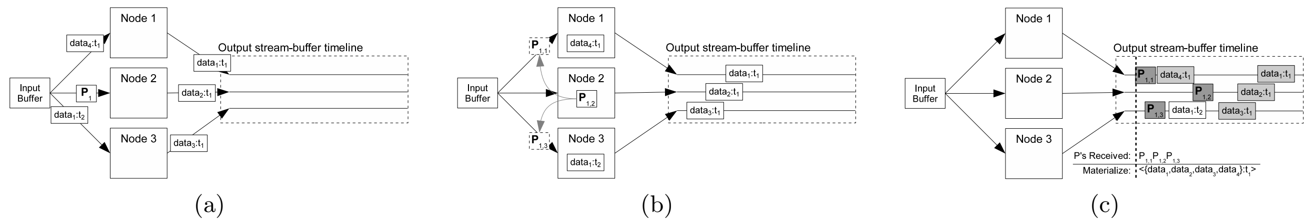


Figure 3: Punctuations, denoting window boundaries, are inserted into the workflow prior to temporally sensitive operators at a point where the stream has been sorted. A node will consume the punctuation from the sorted input stream-buffer (a) and then replicate that punctuation to the other nodes (b). After all replicas are received at the intermediate buffer, we collect data whose timestamps fall into the applicable interval and materialize them as our window (c).

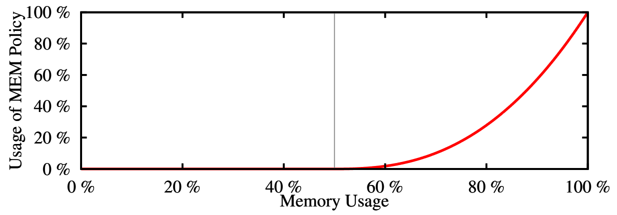


Figure 4: This CDF shows the rate at which our hybrid progressive scheduler transitions to the MEM policy as available memory decreases. Here we use a transition threshold of 50% with the remaining interval represented as a Beta CDF with parameters $\alpha = 2.5$ and $\beta = 1$.

system memory depletes. In this way, as the system memory becomes increasingly scarce, a higher probability of requests for data will use the MEM scheduling policy.

We use this *progressive* scheduling arrangement to improve end-to-end latency when conditions in our system change such that one specific policy is no longer advantageous. For instance, we generally find that ODF does a very good job of producing low end-to-end latencies. However, when bursty workloads deplete available memory, causing data to move into swap space, continuing with the ODF policy results in extremely poor performance. For this reason, we pair the ODF and MEM policies in the progressive scheduler to leverage ODF when memory is plentiful and progressively transition over to the MEM policy when in danger of hitting swap space. Our goal is to achieve the best of both worlds — trying to maintain a focus on minimizing latency while protecting ourselves from exhausting memory (which would hurt end-to-end latency objectives).

To model this progressive transition, we used the cumulative distribution function (CDF) of the Beta probability distribution. The use of a CDF allows for the complete transition between two scheduling policies (on the interval $[0, 1]$) relative to resource availability. The Beta distribution is parameterized by α and β which enable the application programmer to specify a policy transition rate that fits the requirements of the application. The curve can be parameterized to be as linear or as exponential as desired. We use this curve to transition from the ODF policy to the MEM policy once memory usage exceeds a threshold, T. In Figure 4 we show that we stretch this Beta CDF over the interval corresponding to $[T = 50\%, 100\%]$ memory usage with Beta parameters $\alpha = 2.5$ and $\beta = 1$.

To decide whether or not we will use the MEM policy for a particular node's data request, we take the percentage of memory currently in use, x, and pass it into the CDF depicted in Figure 4 which is expressed mathematically in Equation 1. The resulting value corresponds to the portion of data requests that will use the MEM policy for operator scheduling. For each node's request to the scheduler, we can then generate a random number on the interval $[0, 1]$ to probabilistically determine which policy to use. If $\texttt{rand}(0, 1) < F(x)$, then we use MEM and otherwise we use ODF.

$$F(x) = \begin{cases} 0, & x < T \\ \text{BetaCDF}(\frac{x-T}{1.0-T}, \alpha, \beta), & \text{otherwise} \end{cases} \tag{1}$$

Of course, any scheduling policies can be substituted into this progressive scheduling framework and it is also possible to leverage more than two policies concurrently for such a framework. All of this is possible due to the ability to enact differing workload-wide scheduling policies on each new generic processing node's request for data. Experimental results for such a hybrid scheduling strategy are discussed in section 5.2.

4.3.1 Re-Purposing the Combine Phase

In C-MR we re-purpose the Combine phase of MapReduce to incrementally process sub-windows of the windows that a Reduce operation will process. This notion of precomputing sub-windows was initially studied for stream processing by [7] to encourage incremental window processing and reduce redundant computations due to overlapping sliding windows. We first extended this method to continuous MapReduce execution, through the use of the Combine phase [3], and the method has since been adapted for distributed frameworks [8].

We represent the Combine phase as an extra Reduce-like operator in the workflow, situated between Map and Reduce (as seen in Figure 5). It has a window size that is a common factor of the Reduce window's size and slide. The sub-window size and slide values are equal such that there is no overlap between sub-windows. The aggregated results of these sub-windows are computed only once and can be integrated with corresponding downstream Reduce windows. The only redundant computation remaining after such an optimization is the aggregation of the results of the sub-windows at the Reduce operator. The cost of this redundancy equates to aggregating aggregates.

To determine whether or not re-purposing the Combine phase will be worthwhile for a particular Reduce operator in a C-MR workflow, we compare the expected number of values a Reduce window would aggregate over both with and without the Combine phase. If we denote a Reduce window's size as w and the slide as s with the size and slide of the Combine's sub-window to be the highest common factor

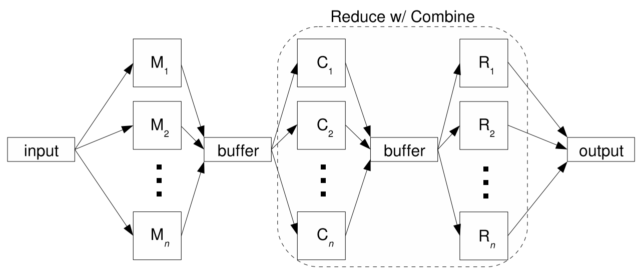


Figure 5: C-MR uses Combine to process sub-windows of Reduce operations. This encourages incremental processing of Reduce windows, data parallelism within Reduce windows, and decreases redundant processing of overlapping Reduce windows.

of these values, k, then we can predict the work spent for both strategies in the following manner using t to denote the average number of values observed in the stream per unit of time.

$$\begin{aligned} \text{Cost of No-Combine} &= wt \\ \text{Cost of Combine} &= st + \tfrac{w}{k} \end{aligned}$$

The cost of No-Combine is equal to the value density of the window (wt), whereas the average cost of using Combine is equal to the value density of the slide amount of the window that has not been observed yet (st) plus the cost to aggregate the sub-window aggregates (w/k) which simply corresponds to the number of sub-windows in a window. Thus, the repurposed Combine phase will be beneficial if the following is true:

$$\begin{aligned} wt &> st + \tfrac{w}{k} \\ wt - st &> \tfrac{w}{k} \\ t(w-s) &> \tfrac{w}{k} = \tfrac{w}{HCF(w,s)} \end{aligned}$$

The result is intuitive. That is, if t (the volume of data per unit of time) is sufficiently large then the Combine phase will be beneficial in aggregating over the bulk of this volume only once. Similarly, if $(w - s)$ (the degree of overlap between adjacent Reduce windows) is sufficiently large then the Combine phase will aid us in not recomputing this significant portion for each window. In all other cases, the benefit of the Combine phase will not outweigh its cost. A simple check of this expression allows us to make an informed decision regarding whether or not to implement the Combine phase, although it does not consider the potential benefit that the Combine phase provides by encouraging incremental processing of the Reduce workload which may improve latency.

Additionally, there is an extra trade-off to consider should we pick the sub-window size to be a common factor of the Reduce window's size and slide that is smaller than their highest common factor. Picking smaller common factors will encourage additional incremental processing but at the expense of overhead incurred due to managing extra sub-windows.

5. EXPERIMENTAL RESULTS

The tests in this section were performed on a computer with an Intel®Core^{TM}2 Quad Processor Q6600. Some of the following tests use a stream application that determines the moving average of stock prices over time. This application uses both transformation (Map) and aggregation (Reduce) operations to parse data into stock symbols and prices and to calculate the average of stock prices observed within a window for each stock symbol present.

To facilitate this application, we replayed NYSE stock data from the TAQ3 data release of January 2006 [1]. The stream contains records representing stock trades which include a symbol, price, and a time stamp of 1-second granularity. For each second, anywhere from 100 to 1703 trades were captured with a range of 71 to 794 unique symbols appearing per second. This provided a large amount of skew in both volume and stock symbol. The stream was played back at an accelerated rate to provide an increased workload.

5.1 Continuously Executing a MapReduce Job

It is possible to take an existing, non-continuous, MapReduce framework and periodically invoke MapReduce jobs over a stream as new windows arrive to facilitate continuous execution. However, this places the burden on the application programmer to perform stream and window management and does not provide opportunities to perform intra-job optimizations such as enabling pipelining, latency-oriented scheduling, or the reduction of redundant computations. Here, we compare the repeated invocation of Phoenix++ [9] (a single-host, multi-core MapReduce framework) jobs over a data stream to a C-MR MapReduce job. At the same time, we also compare these two to a C-MR MapReduce job with a Combine operation interposed between Map and Reduce as an optimization to decrease the amount of redundant reduce computations that occur due to overlapping, sliding windows.

This experiment uses a single MapReduce job which determines the moving average of stock symbol prices over a data stream. Figure 6 shows the results of executing each strategy using various window sizes (all with a window slide of 1 second) while replaying a finite stream of stock data. The results show that redundant computations in-

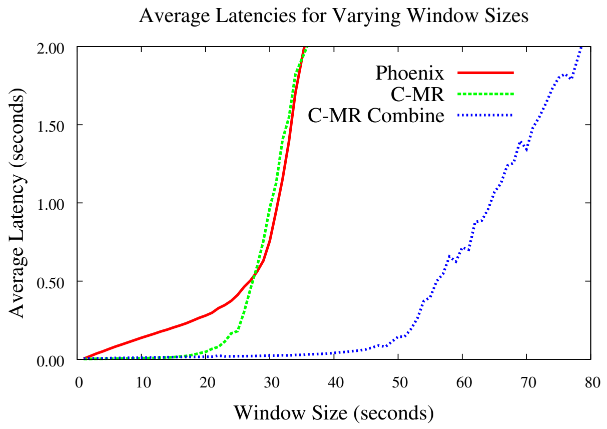
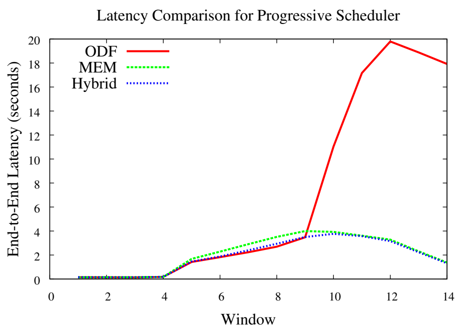
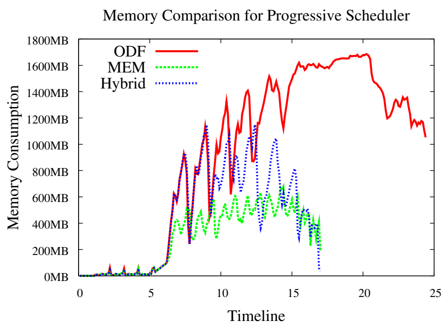


Figure 6: Repeatedly invoking a Phoenix++ MapReduce job over a stream results in many redundant computations (at both Map and Reduce operations). C-MR allows data to be processed only once by Map and the inclusion of the Combine operator significantly decreases redundant work performed at the Reduce operator.



(a)



(b)

Figure 7: The Hybrid strategy initially mirrors ODF, with good latency results. As memory becomes limited, given a burst in stream volume, Hybrid is able to progressively transition towards additional MEM policy invocations, avoid swap space, and outperform the others. ODF ultimately performs terribly as it is unable to avoid swap space.

curred by Phoenix++ ultimately hurt latency; as the size of the workload increases (with larger windows), the latency of Phoenix++ increases more rapidly than the C-MR variants. C-MR hits the overload point just prior to Phoenix++, due to additional storage and organization requirements of intermediate data. C-MR with Combine, however, supports processing significantly larger workloads before reaching saturation.

It is also worth noting, that the Combine strategy only begins to outperform the standard C-MR strategy after a window size of 15 is reached. Prior to this point, the inclusion of an intermediate operation (along with bucketing keys, ordering streams, and materializing windows) is more expensive than the savings in redundant Reduce computations it spares us from.

5.2 Operator Scheduling

The combination of generic computing nodes and our scheduler allows us to consciously schedule complex workflows with latency objectives in mind. We evaluate our scheduling policies on the workflow depicted in Figure 1. The Map and Reduce operators have been assigned varying selectivities, resulting output value sizes, and processing times.

Scheduling policies like Oldest Data First (ODF) are commonly used to reduce the average end-to-end latencies of workflow results. Memory-conservative policies (MEM) also happen to be useful for low-latency applications; they lower memory footprints and reduce the chance of latency spikes caused by spilling into swap space. We test a progressive scheduling policy which allows for a hybrid utilization of both policies.

In this test a burst in the stream is initiated at the 4 second mark and we observe how the scheduling policies (ODF, MEM, and Hybrid) are able to cope with the significant volume given a meager 2GB of available RAM. Figure 7 shows the results for both latency and memory usage.

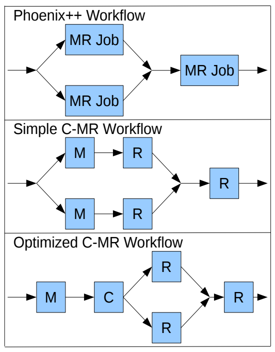
Initially, we find that the burst of data most negatively affects the MEM policy while Hybrid mimics ODF since plenty of memory is available. It is only once the memory footprint starts to significantly increase that we see Hybrid deviate and become more memory-conscious while taking a small latency penalty. Soon, ODF hits swap space and its latencies skyrocket while Hybrid, with a more conservative approach, avoids hitting swap space while still allowing latency-minimization to be a priority. This results in the Hybrid policy consistently providing good (but not always best) latencies throughout the experiment.

5.3 Workflow Optimizations

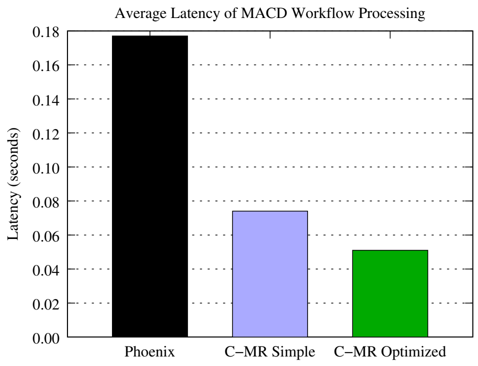
By enabling the creation of complex workflows of MapReduce jobs, C-MR supports the ability to perform a variety of workflow optimizations which includes sharing common sub-workflows. To show this, we used a financial analysis application which performs a moving average convergence/divergence (MACD) query. This query, common to financial trading applications, performs two moving averages of differing window sizes at similar slide intervals over the same stream. The difference of the two moving averages is returned as the result. In Figure 8a, we depict three different workflow implementations – one using a wrapper interface to pipeline data to multiple Phoenix++ instances and the other two being a simple C-MR workflow and an optimized C-MR workflow.

With C-MR we can fork output streams to decrease redundant computations through stream sharing; C-MR handles the generation and propagation of window-boundary punctuations through these forks towards their specific downstream operators. In the optimized C-MR workflow, we allowed the two Reduce steps to share both the Map operator and an introduced Combine operator. Even though the two Reduce window sizes are different, the Combine operator produces sub-window aggregates which both Reduce operators can consume. This prevents a large amount of redundant computations. The Phoenix++ workflow required a considerable amount of work to pipeline data between its MapReduce jobs and to also perform stream synchronization at the input for the MapReduce job which merges and processes its two input streams.

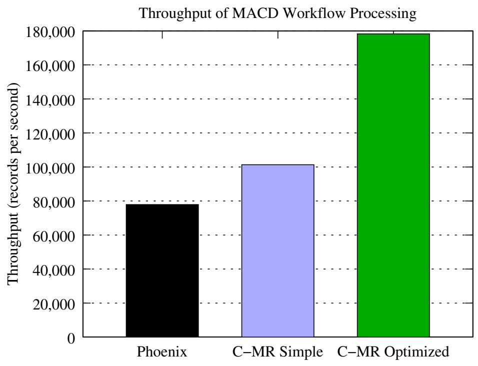
This test performed a MACD analysis on a replayed stock data stream with window sizes of 5 minutes and 10 minutes and a common window slide of 1 minute. In the optimized workflow, the Combine operation produced sub-window aggregates of 1 minute window sizes to the parallel Reduce operations. The latency results of this test are shown in Figure 8b. We see that the Phoenix++ workflow performs the worst because it incurs a large amount of redundant processing and because of its inability to facilitate latency-oriented scheduling. Also, the optimized C-MR workflow



(a)



(b)



(c)

Figure 8: We characterize the ability to execute MapReduce workflows with a sample MACD application which was implemented with two levels of optimization in C-MR as well as through repeated invocations to a Phoenix++ wrapper interface through which we performed stream and window management ourselves. Workflows are depicted in Figure (a) and latency and throughput results are shown in Figures (b) and (c) respectively.

outperforms the simple C-MR workflow with a 31% decrease in average latency.

We also replayed the same stock data stream as 1 large batch which arrives instantaneously to analyze throughput. The results of this test can be seen in Figure 8c. Given this particular workload, we see that both of the C-MR strategies outperform the Phoenix++ workflow with regard to the volume of data they can process. The performance gap for throughput is somewhat smaller than we saw for latency as Phoenix++ is quite optimized for throughput performance. C-MR, on the other hand, currently employs a latency-oriented scheduling policy and incurs a higher per-tuple overhead for doing so. In spite of this, the computation savings and workflow optimizations provided by C-MR allows for better performance.

6. CONCLUSIONS

We presented the C-MR framework which supports the continuous execution of complex workflows of MapReduce jobs on unbounded data streams. By modifying the underlying MapReduce processing model, we were able to preserve stream order and execution semantics while providing a progressive, end-to-end latency-oriented scheduling framework.

Unlike batch-processing applications, the unbounded nature of data-streams and end-to-end latency objectives of stream applications prevent the possibility of simple bottom-up workflow processing. To support stream applications, it is necessary to support scheduling workflows of stream operators and facilitate the interactions between them. Doing so opens up the possibility for us to employ end-to-end latency-oriented scheduling policies and many of the workflow optimization techniques that the stream processing community has exploited in the past such as sub-query sharing, incremental sub-window processing, and adaptive query processing.

7. ACKNOWLEDGEMENTS

This work has been supported by NSF Grants IIS-0448284 and IIS-0905553.

8. REFERENCES

[1] Monthly TAQ, http://www.nyxdata.com, New York Stock Exchange, Inc., 2010.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[3] N. Backman, K. Pattabiraman, and U. Çetintemel. C-MR: A continuous MapReduce processing model for low latency stream processing on multi-core architectures. Technical Report CS-10-01, Computer Science Department, Brown University, Feb 2010.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, pages 313–328. USENIX Association, 2010.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. Deduce: at the intersection of MapReduce and stream processing. In *EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology*, pages 657–662, New York, NY, USA, 2010. ACM.

[7] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34:39–44, March 2005.

[8] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[9] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[10] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, 2003.