Abstract of “C-MR: Continuous Execution of MapReduce Workflows for Stream Processing” by Nathan Backman, Ph.D., Brown University, May, 2013.

Data processing frameworks provide application programmers an interface to manipulate and analyze data. This thesis studies a novel parallel stream processing model, designed for workflow-based data processing frameworks, that leverages application performance requirements to motivate the flexible scheduling and fine-grained allocation of data to computing nodes.

We feature this processing model through the design and implementation of the Continuous-MapReduce (C-MR) data processing framework. 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. Its novel processing model enables: 1) fine-grained, workflow-wide load balancing across computing nodes; 2) the evolving application of data and task parallelism models as guided by application performance requirements; and 3) a novel scheduling framework which supports gradual transitions between scheduling policies relative to application performance and/or resource availability.

This work explores the potential of the C-MR processing model by studying our single-host implementation of C-MR that supports parallel execution on non-dedicated and heterogeneous computing nodes (both multi-core CPUs and GPUs). We then study this processing model through the implementation of a distributed version of C-MR that supports execution on multiple hosts. This endeavor involved the generalizable strategy of employing hierarchical instances of the C-MR processing model while requiring modifications to the data acquisition and load balancing strategies. Experimental results from these studies show that the C-MR processing model can effectively support the continuous execution of workflows of MapReduce jobs for stream processing while being resilient to stream and resource fluctuations due to the processing model’s flexibility and diversification of processing responsibilities.

C-MR: Continuous Execution of MapReduce Workflows for Stream Processing

by

Nathan Backman

B. S., Whitworth University, 2006

Sc. M., Brown University, 2009

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May, 2013

This dissertation by Nathan Backman is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ____

Uğur Çetintemel, Director

Recommended to the Graduate Council

Date ______________ ____

Rodrigo Fonseca, Reader

Date ______________ ____

Stanley B. Zdonik, Reader

Approved by the Graduate Council

Date ______________ ____

Peter M. Weber
Dean of the Graduate School

Acknowledgements

This dissertation is the result of many years of work, during which I was supported by a large number of people to whom I will forever be grateful. First and foremost, Uğur Çetintemel was an invaluable advisor. His guidance, encouragement, and patience lead me to be the researcher that I am today and his general cheerfulness and compassion was more than any grad student could ask for in the midst of such a trying process.

My committee members, Rodrigo Fonseca and Stan Zdonik, provided their wisdom and insight on many occasions to help me analyze difficult problems and to help me better communicate ideas as a researcher whether through discussion, writing, or presentations.

Much of the implementation for this work was not done alone. Karthik Pattabiraman and Zikai Wang worked alongside of me for many, many hours at the computer and at the whiteboard to develop and implement some of the concepts within this document.

Also, I can't thank my friends at Brown enough. To Irina Calciu, Jennie Duggan, David McClosky, Anna Ritz, and Ryan Tarpine, thanks for the positive distractions, discussions, and games. The time we spent together was definitely a highlight of my stay at Brown.

I would not have found my way to graduate school without my prior research advisors and professors from my undergraduate years. To Jacob Furst, Kent Jones, Susan Mabry, and Peter Tucker, I give thanks for introducing me to research and Computer Science and for fueling the passion they found within me.

I would also like to thank my parents, Keith and Gail Backman, and my sister, Hope, for shaping and fostering my intellectual curiosities from an early age. And again, to my parents, for the contributions and sacrifices they made to further my education. They have been an unconditional source of support throughout my life.

Finally, I thank my wonderful wife, Megan, and our daughters, Lily and Brooke, for being the brightest parts of my life. Megan has been at my side through all of the highs and lows and has been the best partner in life that I could ask for.

Contents

List of Tables

List of Figures

Chapter 1

An introduction to data processing frameworks

Often times, a programmer is tasked with the job of processing data. Data processing frameworks are designed to facilitate the analysis and/or transformation of data while minimizing implementation complexities for the programmer.

In the most trivial of cases, the job of the programmer may be quite easy. For instance, if a person wanted to simply take the total word count of all text files within a directory they could simply execute the following command in a shell: `wc *.txt`. However, data processing applications can also be substantially more sophisticated. It may be that the application has numerous inputs and that there are a variety of processing steps (perhaps some conditionally, dependent on the data) which are required to occur in a specific sequence. Organizing the flow of data through a workflow of computational operations can take quite a bit of effort regarding the communication, synchronization, and assembling of data and intermediate results.

The difficulty of managing such applications will continue to increase as the complexity of the workflow of processing operations increases and as parallelism is leveraged to include many computing nodes in the case that the computational demands of a job cannot be satisfied on a single computer. A programmer might require results within a certain period of time and a single computer alone may not be able to produce the results within the required time constraints. Likewise, a single computer may not have the memory capacity required to perform the task at hand and the work might therefore need be distributed to multiple computers. Performing this management by hand, for each data processing application, is time consuming, error prone, and not fulfilling work. Furthermore, for applications of a similar type, there may be a large degree of commonalities between the management of those applications. What a programmer will find themselves with, after managing their own suite of data processing applications, is that they will have a large amount of boilerplate code and framework whereas the application-specific code is quite small in comparison.

The goals of data processing frameworks is to relieve programmers from the burden of managing the execution of their applications. Such frameworks will handle parallelization of code onto clusters of computers, manage network communications, handle issues regarding fault tolerance and high availability, and provide a simple means for the application developer to outline the actual data analysis that is to occur. In this way, the programmer will only be responsible for writing application-specific code and specifying the inputs to the application. This is usually accomplished by distilling the application logic into a sequence of succinct operations and defining a directed acyclic graph denoting in what order to apply those operations on the data. Data will then flow through graph from the inputs of the application and be processed by the intermediate operators before being produced on the outputs of the graph as the application results. The specification of such graphs can be manual or derived from a higher-level language such as SQL or Pig[34].

1.1 Requirements

The requirements of data processing frameworks vary depending on the types of applications they serve.

Batch processing frameworks are intended to support applications which analyze or transform a fixed volume of data. Examples of such processing include analyzing collections of web logs, sorting large amounts of data, document clustering, applying statistical machine learning algorithms, and even the application of SQL queries to databases. The goal, when executing these types of jobs, is to return the results of the task as quickly as possible. In other words, the goal is to maximize throughput (accomplishing as much processing per unit of time as possible) in order to minimize the latency of the job at hand.

Stream processing frameworks support a different class of applications. These applications, by definition, continuously consume and process data while continuously producing results. Such continuous flows of data are called *s*treams. Examples of streams include event logs, user-click streams, image/video feeds, network traffic, sensor readings (temperature, GPS coordinates, accelerometers, traffic movement), and various other data feeds.

The requirements of stream processing applications are different than those of batch processing applications. Stream processing applications are temporally sensitive – the order of data within the stream can have an impact on the results of the application. They are generally time-critical such that their utility is proportional to the promptness of the results. For example, stream applications which seek out network intrusions or financial fraud patterns are meant to respond quickly to an observed threat. Other examples of stream applications include automated stock trading, real-time video processing, geo-spatial trajectory modification, and vital-signs monitoring. Results produced by such applications are often urgent and require immediate attention. As the outputs of these applications become more and more delayed, their worth and applicability rapidly decrease. This is apparent in the financial sector regarding stock analysis – firms that are able to observe important trends in stock ticker streams, before their competing firms, will be able to reap financial rewards. In order to support time-critical stream processing applications, it is important to minimize the average latency of the continuously emitted results instead of throughput. On the other hand, stream applications which are not time-critical exist such that their aim is to simply process as large a stream as possible with the goal of maximizing throughput.

In addition to latency and throughput requirements, batch and stream applications also have usability requirements such as requiring a simple programming interface and abstracting the programmer away from the demanding jobs of parallel task management, network communication, fault recovery, and load balancing.

1.2 Challenges

The computational complexity and/or data volume of batch jobs and stream applications can be quite substantial. In order to meet their performance requirements, it is often necessary to distribute their workloads to many computing nodes to be processed in parallel.

Doing so effectively, however, requires:

- Balancing workloads across the given set of computing nodes such that the contributing nodes are being efficiently utilized
- Scheduling workflow operators in a manner that is conducive to meeting performance expectations
- Ensuring high-availability to provide correct results and speedy recoveries in the midst of faults

While there have long been policies in place to perform load balancing, operator scheduling, and provide high-availability, it is important to provide solutions for these challenges that are mindful of the users performance expectations.

1.2.1 Load balancing

In the case of batch jobs, all participating computing nodes should ideally finish their alloted portion of work as quickly as possible with the allocation distributed in such a way that all hosts will finish their allocations simultaneously to eliminate the presence of stragglers. For stream applications, the processing power of the computing nodes should be distributed across the workflow to provide minimal end-to-end workflow latencies.

Unfortunately a number of factors, relating to both the computing hardware and the workload itself, increase the difficulty of finding such optimal workload allocations.

- **Hardware Volatility:** Leveraging increasingly large clusters of computing nodes can will increase the probability of failures as well as the impact of stragglers due to various software and/or hardware issues. Given the sheer number of participating computing nodes in some modern-day compute centers and the size of the infrastructures required to support them, these problems are to be expected. In non-dedicated computing environments, we may also expect the computing potential of nodes to vary due to external workloads. These days, with the emergence of the cloud and virtualization, truly non-dedicated computing nodes are increasingly difficult to find.

- **Workload Volatility:** It is common to experience volatility in the data for both batch and streaming workloads. Data skew can cause significant imbalances for operations that performs aggregate operations over like data items. In a parallel processing environment, data of the same type must ultimately be aggregated at the same physical site and therefore skew in the data distribution among data types can result in hot spots at computing nodes. A different kind of skew can be observed in the computational effort required to process individual data items leading some nodes to spend more processing time on data than others while perhaps being allocated the same amount of data. Data bursts, in the case of streams, can also cause load balancing troubles when we incorrectly predict future processing requirements of operations in a workflow.

These uncertainties, with respect to both hardware and data, require processing frameworks (both batch and stream-oriented) to be dynamic and adapt to such volatility in order to maintain high-efficiency of computing resources and to meet application requirements.

1.2.2 Scheduling

Load balancing alone is not enough to ensure efficient utilization of computing nodes. We must also consider dependencies between operators, the computing node on which data should be processed, and the order of data execution. In other words, it is important to determine what balance or mixture of data, task, and pipelined parallelism will be appropriate for an application given its performance requirements. Such decisions have a significant impact on throughput and end-to-end latency.

The challenge here arises in how to maximize the use of the computing nodes in a way that is beneficial to the goals of the application. For instance, in batch processing, where individual data latency is unimportant, we should aim to minimize expensive disk reads and context switches. For stream processing applications, however, it can be beneficial to incur additional context switches in the effort to quickly progress data items down the workflow and towards the output in order to minimize the average latency of the results.

In non-ideal circumstances, this becomes even more challenging due to volatility in both the hardware and workload (just as in load balancing). As precious resources such as available main memory deplete with bursts in the workload, it will be necessary to adapt our scheduling policies to avoid catastrophic events such as running into swap space or buffering data to disk.

1.2.3 High-availability

As mentioned earlier, hardware failures are inevitable and should be expected when using clusters of computing nodes. It is vital to ensure that correct results are produced in the midst of failures and that we con recover from these failures quickly and with little cost.

Batch processing frameworks are often used to process such large volumes of data that much of the intermediate data that are generated must be stored on disk. By using distributed file systems with fault tolerant properties[21, 1], batch processing frameworks can be protected from some degree of hardware failures. Any data lost after one of the intermediate checkpoints can be replayed to working computing nodes.

Stream processing frameworks, on the other hand, have such strict end-to-end latency requirements that buffering data to disk is unacceptable. Disk seek and read/write times result in such significant time penalties that shedding stream load (and sacrificing result correctness) can often be a preferred alternative[40]. For this reason, intermediate buffers are generally managed in main memory. To then preserve data in the event of faults, it is necessary to resort to replicating the effort of computing nodes (to instantly replace failed nodes) or replicating data (to replay a copy of prior data to other working nodes).

Coordinating replications and points of aggregation in elastic computing architectures is a challenge in itself. As nodes come and go (or fail), it is necessary for the remaining nodes to synchronously be aware of updated aggregation points and replication destinations.

1.3 Background

Batch and stream processing frameworks have both been around for quite some time. We have leveraged previous work and concepts from both types of frameworks.

1.3.1 Distributed batch processing

Batch processing frameworks and database management systems have been using data parallelism to process large workloads for quite some time. Volcano[22] was proposed in the early 90's as a means of encapsulating the tasks of distributing data and gathering results for partitioned operator parallelism in a workflow. Other tools such as MPI[32] implementations and OpenMP[35] require the programmer to invoke specialized libraries to parallelize specific sections of their applications.

MapReduce[20], and associated implementations such as Hadoop's MapReduce[2], have isolated these portions of parallel code in order to free application developers from the management of their execution. In this way, programmers do little more than simply define transformation and aggregation operations which are then automatically replicated across many computing nodes for the purpose of data parallel processing. Languages have been created to provide SQL-like expressiveness, such as Pig[34], in order to define workflows consisting of MapReduce jobs. Schedulers such as Oozie[3] (Hadoop's workflow scheduler) then try to optimize the execution of these jobs over the available set of computing nodes in order to increase the throughput of these batch jobs.

1.3.2 Distributed stream processing

Stream applications are expressed as a set of computations performed over streams. Each basic computation is represented as an operator in the stream and receives input data from upstream operators or stream sources while producing output data to downstream operators or application outputs. In this way, application programmers would construct a directed acyclic graph (DAG) which represents how an application processes a stream. It is the responsibility of stream processing frameworks to facilitate the execution of such a workflow of operators for each application it supports.

Early distributed stream processing engines did not support data parallelism but did support pipelined and task parallelism[12, 17]. Workflows of operators were generally partitioned into subsections with each computing node handling a different set of consecutive operations. Fault tolerance in these systems was usually handled through either: 1) a mirrored set of secondary computing nodes on standby which would be utilized if a primary node went down[25]; or 2) by holding onto upstream backup copies of data in which such case data could be replayed downstream to a still working node[38, 25].

Later, partitioned parallelism was integrated into stream processing engines via concepts laid out by RiverDQ [14]. This strategy involved interposing load balancing operators within a workflow to direct data to a set of pre-defined computing nodes that were allocated to a parallelized operator. Flux[37] would enable this strategy to be used for content-sensitive data partitioning to support aggregations with appropriate intra-operator load balancing and would later be expanded to provide fault tolerance[38].

Chapter 2

A Generic-Node Processing Model

Distributed stream processing engines often employ rigid operator allocation strategies: they allocate the entire workload of a subset of operators [45, 36, 44] and/or a portion of the workload from a single but expensive operator [14, 37] to each computing node. "Pinning" nodes to operators, in this way, results in coarse-grained workload balancing — on the scale of nodes to operators. Additionally, pinning nodes to a specific set of operators prevents those nodes from contributing to other operators across the workflow in times of need to withstand stream or resource volatility. In effect, these nodes' operator schedulers (which determine the order of execution for the operators that have been assigned to a node) have been restricted to choosing the best plan of execution for only a subset of the workflow instead of the whole.

Statically employing data, task, and/or pipeline parallelism by pinning operators to nodes may not efficiently meet end-to-end latency objectives. The forms of parallelism enacted by nodes, at any moment, should be motivated by application performance objectives (such as minimizing latency). To this end, we present a distributed stream processing model that supports workload allocation and scheduling frameworks that are able to manage parallelism in real-time. Instead of pinning nodes to operators, we *partition* all operator workloads for parallel processing while supplying a novel strategy to maintaining the sorted integrity of streams for temporally sensitive operators amidst data parallel processing. We then distribute those partitions across the set of all computing nodes, to encourage forms of parallelism which reduce latency, and employ a latency-oriented, workflow-wide operator scheduler to concentrate the nodes' collective efforts on the portions of the workflow that are most important to minimizing latency at any given time.

We claim that supporting the processing of volatile and fluctuating stream workloads with heterogeneous and non-dedicated computing nodes requires a flexible processing model.

2.1 Generic computing nodes

We have coined the term *G*eneric computing nodes to represent computing nodes that have not been restricted to participating only in a specific subsection of the workflow; instead they have the flexibility to receive and process data from any workflow operator[15].

Our processing model uses this concept to allow generic computing nodes to flexibly move throughout the workflow to support workflow-wide scheduling, to withstand stream and workload volatility by diversifying processing efforts of computing nodes, and to provide fine-grained distribution of the entire application's load across the available computing resources. The remainder of this section discusses these benefits and how generic computing nodes enable them.

2.1.1 Pinned vs. non-pinned computing nodes

Both batch processing frameworks and traditional parallel stream processing engines generally restrict the participation of a computing node to a specific operation.

Computing nodes used in batch processing frameworks such as MapReduce are assigned ("pinned" to) large batches of data to process and may not process data from another operation until *all* participating nodes have finished the entirety of their allocated workloads.

In distributed stream processing engines, operator workloads are generally assigned ("pinned" to) computing nodes using one of the following policies: 1) nodes can be statically assigned to operators in an arrangement which is meant to be resilient to observed patterns of fluctuation within a data stream[26, 44]; 2) the entire workload of an operator may be migrated from one computing node to another[12, 36, 45]; and 3) load-balancing operators may be interposed into the workflow to migrate portions of an operator's data parallel workload between the computing nodes assigned to that operation[14, 37].

These strategies all constrain the computing nodes to participate with only a specific subset of the operators within a workflow. This inflexibility results in an inability to perform both workflow-wide load balancing and workflow-wide scheduling in the midst of resource volatility and stream volatility caused by bursts and skew.

The generic node processing model, on the other hand, removes these restrictions. By allowing computing nodes to consume data from any operator in a workflow, it is possible to attempt to balance the entire workload of the application across all available nodes. Likewise, all nodes can be scheduled with respect to the entire workflow of operators. Therefore, if the workload of a particular operator increases significantly, all computing nodes can be scheduled to execute that operator in order to alleviate its burden. In the case of a pinned assortment of computing nodes, on the other hand, the nodes assigned to the burdened operator will likely be overloaded while the remainder of the computing nodes might be underutilized.

Load distribution comparison

To show this, we provide a simple C-MR workflow (leveraging our C-MR framework which is defined in the following chapter) containing a single MapReduce job (with no Combine step) and compare C-MR's use of generic, non-pinned computing nodes to all possible pinned node allocation strategies. We used an Amazon EC2 High-CPU Extra Large instance and supplied 6 cores to be allocated to the workflow. The cost of the Map and Reduce operators were derived artificially with busy waits and the Reduce operation consumed windows with a size and slide of 1 second.

To test the effects of volatility on the two strategies, we inserted two anomalies over the testing interval. The first is a burst at the 10 second mark which lasts for 10 seconds and nearly doubles the input rate. The second anomaly is a cycle-stealing process, simulating external workloads, that is collocated with one of the cores for 10 seconds starting at the 40 second mark. The results of this experiment are depicted in Figure 2.1.

The assignment of 3 nodes to the Map and 3 nodes to the Reduce is the best possible allocation the pinned strategy can provide given the specified workflow and available resources. The generic nodes used by C-MR, however, consistently outperform the best Pinned strategy as generic nodes experienced better utilization and were able to collectively contribute to hot-spots in the workflow as they occurred. The fine-grained load balancing of all workflow data onto all computing nodes allowed the generic node strategy to have better utilization across the cores while allowing the cores to collectively assist at operators experiencing bursts or take load from cores that are burdened.

2.1.2 Data vs. task vs. pipelined parallelism

There are three types of parallelism common to data stream processing. Data parallelism is the act of partitioning the workload of a single operator among multiple computing nodes. Task parallelism enables the concurrent processing of separate workflow operations. Pipelined parallelism dedicates nodes to differing regions of a pipeline in order to reduce the overhead of each node (reducing context switches and improving cache utilization).

The simple application of any these parallelism concepts with pinned computing nodes, however, is not necessarily beneficial for low-latency stream processing. Strict pipelined parallelism can result in poor node utilization and an inability to prioritize differing areas in the workflow on-the-fly. Distributing computational effort temporally across a workflow can be counter-productive to minimizing the average latency of results. Pipelined parallelism implies de-coupling efforts across

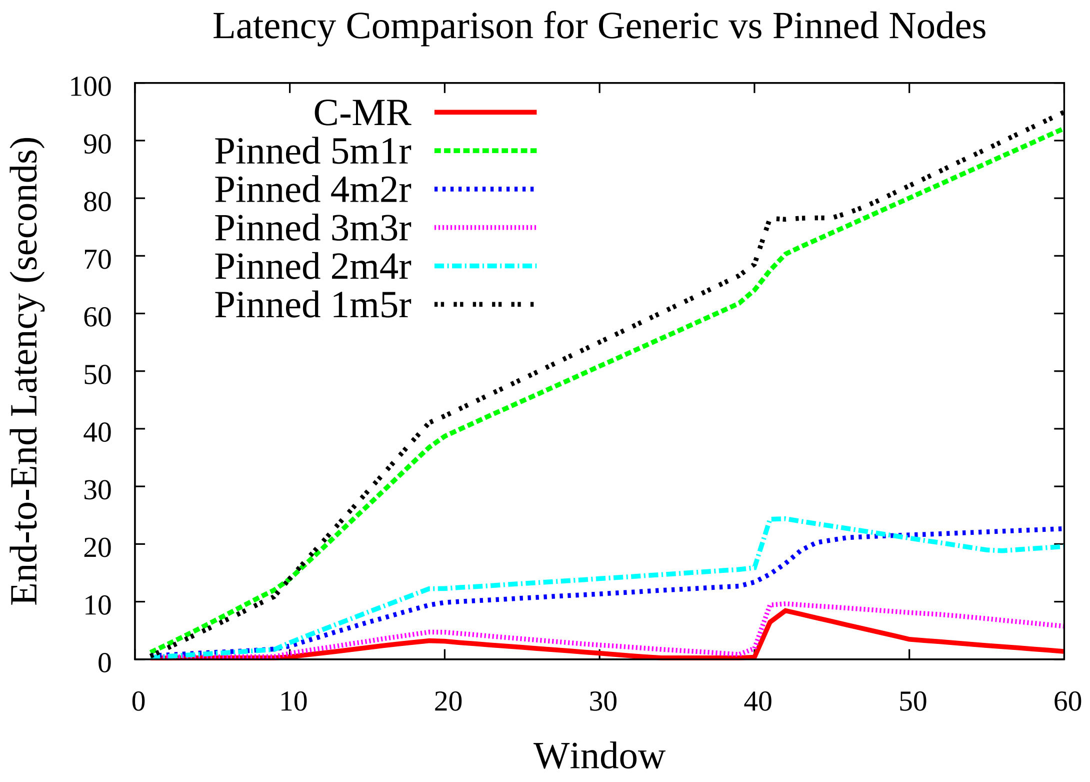


Figure 2.1: Pinned vs. Non-Pinned computing nodes

time (and therefore provides the concurrent execution of data of differing priorities), but it may be beneficial to instead keep efforts coupled (i.e., to process oldest data first). These same problems can also be experienced with the fixed provisioning of data and parallel parallelism within a workflow.

The goal of a processing framework is not simply to achieve parallelism, but to use parallelism to meet the performance requirements of the application. This may require the right balance of these different kinds of parallelism as identified by an operator scheduling policy. Data, task, and pipeline parallelism are therefore not themselves targets to aim for but they are instead the tools we should use to do the work of our scheduling policies.

2.1.3 Fitting heterogeneous architectures

The use of generic, non-pinned computing nodes, in contrast to pinned computing nodes, results in the benefit that an additional node will supplement the entire workflow and not just a subset of it. Our processing model treats computing nodes as black boxes — they simply consume, process, and emit data and any underlying implementation details are completely abstracted away. This affords us the opportunity to utilize fundamentally different architectures such that computing nodes can be any individual processing elements including CPUs/cores and even GPUs. Many computers are currently being packaged with GPUs capable of performing general purpose computations instrumented via parallel processing frameworks such as CUDA[4] and OpenCL[5] and therefore the utilization of these processors can provide substantial benefits.

To test the ability of our processing model to handle the utilization of differing architectures, we adapted and integrated Mars[24], a MapReduce framework for graphics processors using NVIDIA's CUDA architecture, into our C-MR framework (defined in the following chapter), in which we've implemented our processing model, to support the inclusion of GPU computing nodes. Since computing nodes are abstracted away as black boxes, when they are issued data to process data, we simply invoke an instance of Mars underneath the abstraction and return the results.

GPUs are excellent at performing SIMD operations but are not well suited for operations where the threads executing on the GPU's many cores will take differing execution paths. Therefore, GPUs will not be appropriate for all types of streaming applications but they certainly have their place.

Since GPUs are meant to process numerous data simultaneously, our GPU computing node performed an internal layer of batching under the black-box abstraction. In this way, it would fetch many data items before internally processing the batch of them. Thus, GPUs are beneficial for batch-heavy, or high volume, streams which includes processing replayed streams that are stored on disk.

To show the combined benefits of utilizing both CPU and GPU computing nodes, we performed an experiment varying the number of CPUs and GPUs used for a stream application.

This experiment was performed on an Intel®Core^{TM}2 Quad Processor Q6600 with a GeForce 9800 GX2. In these tests, a single CPU core was dedicated to the purposes of GPU kernel invocation and other supplemental work for Mars. We delivered a high volume stream (too large of a volume for CPUs to handle alone) to a simple MapReduce workflow which performed a series of mathematical operations suited for SIMD processing on GPUs. The results of this experiment are shown in Figure 2.2 as we show the benefit of the cumulative effort of heterogeneous nodes both with and without the GPU.

The GPU node is certainly a power-house and can quickly work through much of the data. When used alone, the CPU nodes struggle under the load of the stream. However, the addition of CPU nodes when the GPU node *is* operating does provide a valuable incremental benefit. Therefore, we find that heterogeneous nodes are able to complement each other in C-MR due to the use of a generic-node, pull-based processing model which allow nodes to consume data as quickly as they are able to.

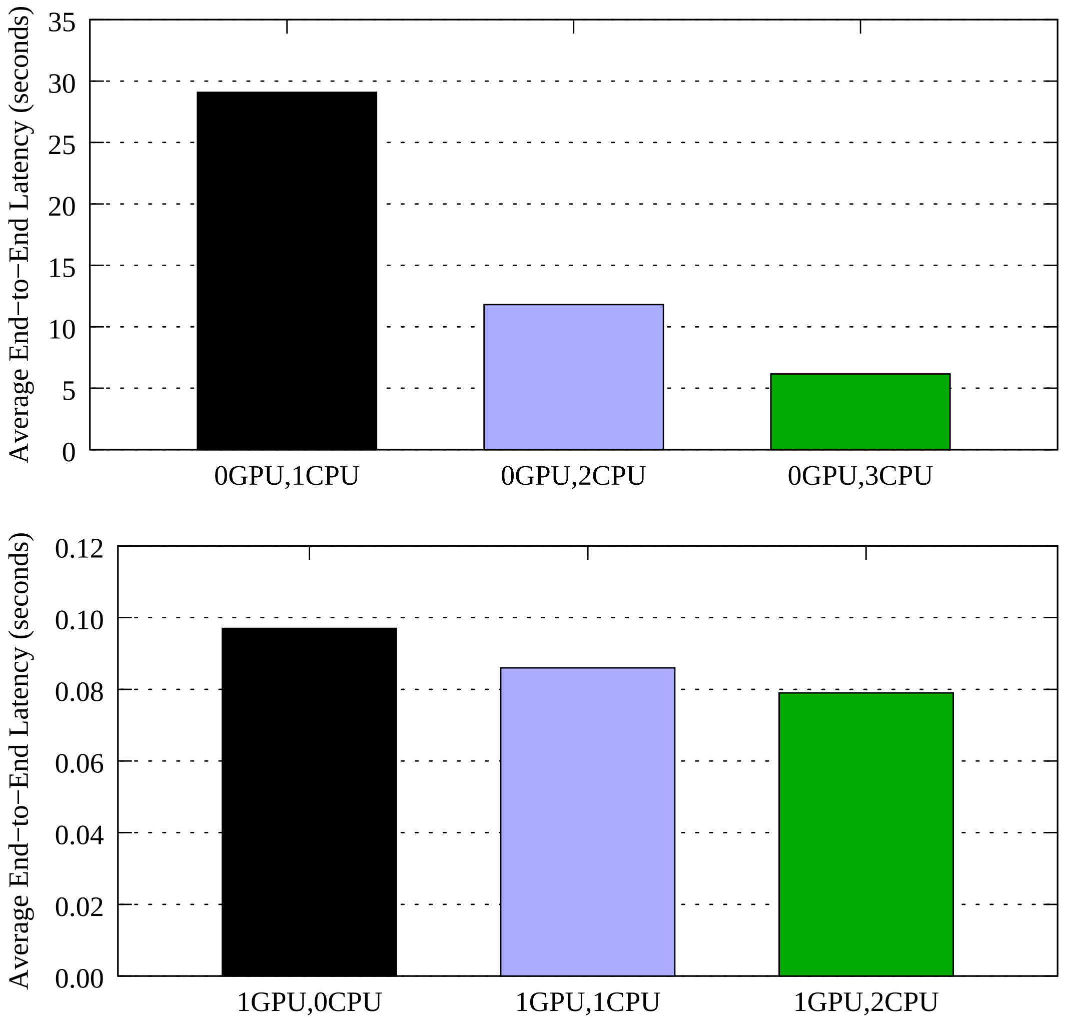


Figure 2.2: Contributions of heterogeneous computing nodes

2.1.4 Resilience to volatility

This arrangement of non-pinned computing nodes encourages node participation at all points in the query plan. If there is a sudden hotspot developing at the input to an operator then many computing nodes will be able to collectively alleviate these bottlenecks. Otherwise, a node that is constrained to only participate in a specific region of the workflow may sit idly during a lull as other nodes are overloaded due to a burst in data. Non-pinned nodes are better able to share variances in stream volume and skew.

2.1.5 Workflow-wide scheduling opportunities

The scheduler is able to sway the collective set of computing nodes together such that each node consumes the data item with the highest priority, according to the scheduling policy, out of the entire workflow. For instance, for an end-to-end latency oriented policy, each node might be told to process the oldest data item in the workflow.

Also, since the computing nodes come to the scheduler independently, as they are ready to process new data, the scheduler can use different policies for different nodes or even toggle between policies for the same node. An example of this might be when the goal is to minimize end-to-end latency and the amount of free main memory drops significantly. In such a case it can be beneficial to leverage a memory-aware scheduling policy that reduces the memory used in order to not run into swap space. Running into swap space would incur tremendously large latency penalties and therefore it may be beneficial to the goal of latency minimization to prevent depleting main memory. By toggling only a few of those nodes that request data to instead process data that will provide a memory reduction benefit, we can keep the rest of the nodes using a latency minimization policy while still protecting ourselves from hitting swap space.

2.2 Preservation of stream order

Some stream operators process individual data items from the stream independently from all other data in the stream. Others perform an analysis or transformation on a range within a stream. Stream processing applications therefore define *windows* which identify these ranges for processing.

Windows are a natural concept to stream processing. They facilitate concepts such as temporal aggregation — for instance, a window would support a query which computing the moving average of a stock price over the last 5 minutes while being evaluated every 1 minute. As identified in this example, windows have a *size* to define the span of the window and a *slide* to define the frequency of evaluation.

When operators consume windows for processing, their input streams must maintain a sorted order so that windows enacted on them will capture the appropriate data. While windowed data streams have been studied by the stream processing community for years, the application of windows to data parallel stream processing has been less closely examined.

As windows can only be formed on sorted streams, we find that streams delimited by data-parallel operations pose a complication in that data parallel processing does not guarantee the preservation of stream order for the results produced. It is therefore necessary to put in place provisions to re-sort the stream prior to temporally-sensitive operations.

2.2.1 Punctuation insertion and replication

As mentioned previously, the inclusion of data-parallel operations in a workflow violates the integrity of a stream's temporal ordering. We cannot be assured in which order parallel computing nodes will return results, therefore, we must assume that the intermediate results produced by a data-parallel operation will arrive out-of-order.

To correct the temporal order of our streams, we use a basic form of *punctuations*[43]. These punctuations are inserted into sorted streams at some interval relative to an application or system attribute (i.e., application timestamp or system time) to denote the ends of windows within a stream. They are, however, only useful when the stream is sorted. Punctuations would become invalidated if, for example, one arrives prior to the end of the window it is meant to terminate. In the case of a data-parallel operation, the merged stream that the computing nodes collectively produce may be unsorted, however, the streams which they consume are sorted. With our processing model, we exploit this characteristic and replicate window boundary punctuations and insert them into each of those parallel streams (one for each active computing node). When a node receives a replicated punctuation, it performs no operation but instead simply passes on the punctuation to the merged downstream buffer.

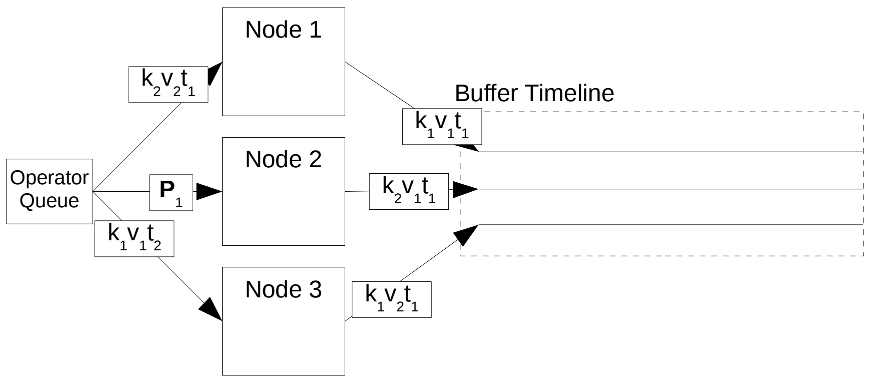
2.2.2 Window materialization

After collecting all replicated punctuations at the downstream buffer, we can be assured that all of the window's corresponding data has been received. This triggers a materialization of the window which can be sent to the downstream window-consuming operator. The data within each materialized window is not required to be sorted since DSMSs only require windows to be a relation of temporally adjacent records, however it is trivial to also sort the data within the window at this time.

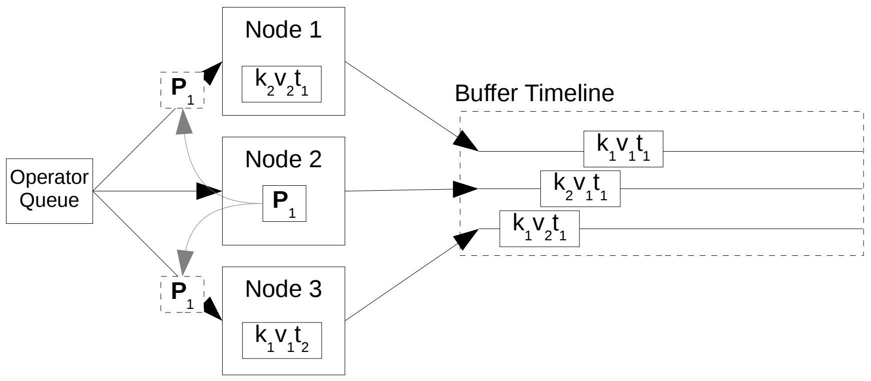
An example of punctuation replication, nodes passing punctuations, and punctuation collection for window materialization is depicted in Figure 2.3.

Notice that punctuations are only necessary for aggregate operations that require windowing, and that punctuations can only be inserted into streams that have already been sorted. Therefore, we require the application's input stream to be sorted by some timestamp, whether application timestamp or system timestamp. This constraint allows us to punctuate the sorted input stream for downstream windowing operations. Likewise, we can insert punctuations at the next downstream window materialization point after the stream has been sorted and so on for additional downstream operations.

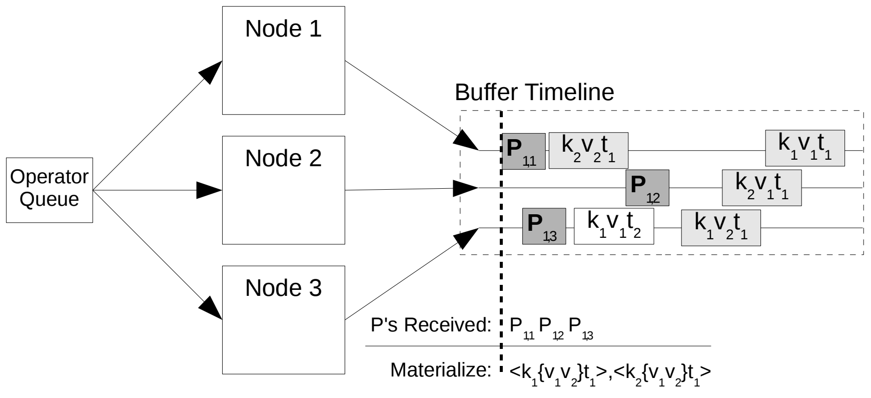
As an alternative to requiring sorted inputs, one could employ a commonly used technique to



(a)



(b)



(c)

Figure 2.3: Punctuation management and window materialization
Punctuations denoting window slide boundaries are inserted into the workflow prior to aggregation at a point where the stream has been sorted. A node will consume the punctuation from the sorted stream (2.3a) and then replicate that punctuation to the other nodes (2.3b). After all replications are received at the intermediate buffer, we collect values of like keys, whose timestamps falls into the window interval, and materialize them as our window.

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 and has been covered by previous stream processing systems [13, 30, 27].

Chapter 3

The C-MR Framework

We developed a new stream processing framework, Continuous-MapReduce (C-MR), intended for execution on a single host to demonstrate the benefits of our generic-node processing model. To effectively leverage the generic computing nodes in our processing model, we decided to use a programming interface for our framework that supported the simple definition of parallelized operators and resulted in the creation of intermediate data that could be consumed by any computing node. The MapReduce programming model provided these properties so we based our framework around the notion of stream-oriented MapReduce jobs to process unbounded streams.

MapReduce[20] has become very popular since its debut, largely due to its simplistic programming model and automatic handling of the parallelization, scheduling, and communication associated with parallel data processing. Hiding these intricacies has lowered the barrier to entry for application programmers to begin parallelized, batched data processing.

Recently, programmers have found it useful to leverage MapReduce to process data streams, going so far as to write scripts to periodically invoke MapReduce implementations over subsets of streams as they arrive. This approach, however, becomes complicated when faced with elaborate workflows of MapReduce jobs dependencies, the need to evaluate overlapping windows (which incurs redundant computations), and the desire to optimize the execution of the workflow.

To facilitate its easy adoption and efficiently support the continuous application of MapReduce to data streams using heterogeneous parallel hardware platforms, it was necessary to provide the following key features and functionalities:

- Little or no changes to the standard MapReduce programming model
- Support for complex workflows of multiple MapReduce jobs
- Automatic stream window management under parallelized operation

- A workflow-wide, latency-oriented scheduler that would effectively utilize the available parallel resources and deal with time-varying load and spikes common in real-world streams

Continuous-MapReduce (C-MR) extends the traditional MapReduce processing model to support continuous execution over unbounded data streams. To understand C-MR, it is important to first understand the traditional MapReduce processing model.

3.1 MapReduce processing model

Google developed the MapReduce framework[20] to leverage clusters of commodity computers for large-scale, data-parallel processing while abstracting away the difficulties of managing the computing resources from the application programmers. These programmers are supplied with a notion of *Map* and *Reduce* operations whose functionality they implement to manipulate their data.

The MapReduce processing model begins with a **Map Phase** in which data are read from a distributed file system, partitioned among a set of available computing nodes, and consumed by those nodes as `(key, value)` pairs. The Map tasks, as implemented by the application programmer, will process each of its input records, independently of the other records, and for each record generate intermediate results of the form `list(key, value)`. The intermediate data is buffered to the local disk.

After the Map tasks on *all* computing nodes have been completed, the **Reduce Phase** begins in which all intermediate data with the same key are collected at the same node to perform an aggregation specified by the application programmer. As computing nodes are allocated to become reducers, they will contact a centralized oracle to be assigned a set of keys to aggregate over. They then fetch the corresponding data from all nodes that produced those keys as output during the Map phase. This data is retrieved and sorted locally after which the data is processed and results of the form `list(value)` are produced for each key.

Optionally, there is a **Combine Phase** which enables a mapper to perform a Reduce-like aggregation over the output values of a Map task that have identical keys before they are written to disk and subsequently sent to the Reduce nodes. This phase serves to conserve network bandwidth.

3.2 Continuous-MapReduce

Continuous-MapReduce[16], like MapReduce, allows programmers to define Map and Reduce operations to manipulate streaming data while being free of the burdens associated with parallelized task and resource management.

3.2.1 Window semantics

The most fundamental difference between C-MR and MapReduce lies in the scope by which the operations execute. Streams, by definition, are unbounded and can therefore not be aggregated by standard blocking operations such as Reduce. A Reduce operation will never be able to produce output because the end of a stream will never arrive. To deal with such blocking operations, the act of stream aggregation is performed as the periodic analysis of finite temporal ranges of data. These ranges are known as *windows* and slide over the stream with respect to time.

The boundaries of such a temporal window are parameterized by a *size* and *slide*. A window with a size of 60 seconds and a slide of 15 seconds will allow us to process all data within the past minute at 15 second intervals. In this way, Continuous-MapReduce provides an analysis of temporal relations within an infinite stream.

With this in mind, it is of the utmost importance that these temporal relations be sorted with respect to time to provide a correct analysis. Parallelized operations (the foundation of MapReduce) do not preserve stream order. We previously discussed the solution to this problem, which our processing model implements, in Section 2.2.

3.2.2 Asynchronous Processing

C-MR differs from MapReduce regarding when and where data are scheduled to be processed. MapReduce is concerned only with the scheduling of a single batch of data through a single MapReduce job. The data is first processed entirely by the Map operation and then by the Reduce operation. Continuous-MapReduce, on the other hand, involves an infinite number of "batches," in the form of windows, as well as potentially many MapReduce jobs for large workflows. Such an arrangement requires the concurrent execution of many operations.

Streaming input sources, as well as intermediate connection between MapReduce jobs within a workflow, result in the asynchronous arrival of data. The input queues to such operators can then regularly be filled with data. For Map operations, this data can be processed immediately. For Reduce operations, on the other hand, the data must be batched until all data relevant to the window of interest has been received. Once such a window has been materialized and is ready to be processed, we have data in the form of discrete computable units. These units, whether Map or Reduce data, can be processed at any computing node.

If computing nodes are logically removed from the burden of preparing these computable units, then they need only worry about processing the data that is available. This opens a wide range of scheduling opportunities as each computing node is able to actively process data from any operation in a workflow. With an assortment of data ready to be processed at their respective operators in the workflow, we have the ability to enact scheduling policies which emphasize a data consumption order that is designed to meet application performance requirements.

In a multi-core system, for instance, each computing node would have available to it the same data as the other computing nodes. The C-MR scheduler can identify the data each node should consume to process. In this way, nodes consume as they are able to and at their corresponding rates in a pull-based manner. The scheduler also has the opportunity to flexibly enact different scheduling policies for individual computing nodes. These characteristics of our processing model allowed for the creation of novel operator scheduling strategies that we will explain in Section 3.7.

3.3 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 3.1.

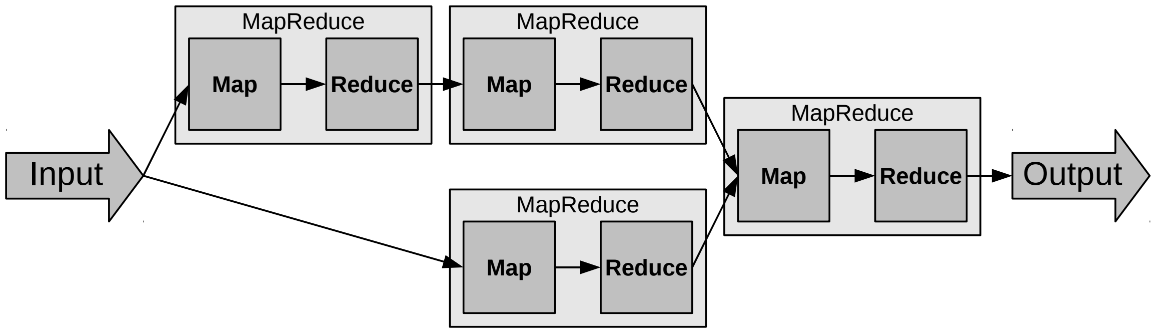


Figure 3.1: An example C-MR workflow.

3.3.1 MapReduce interface

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 [20], 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 3.1.

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 3.1: MapReduce interface function prototypes

3.3.2 Stream interface

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 3.2.

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

Table 3.2: Stream input/output function formats

3.3.3 Workflow construction interface

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 so that data can be directed 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 3.3. Similarly, C-MR supports the addition of individual Map or Reduce operators to a Query through an `addOperator` function.

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 3.3: Workflow creation interface function prototypes

A sample windowed moving average stock application can be seen in Appendix A.

3.3.4 Schema

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.

3.4 Architecture design

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

Our computing nodes have a different execution strategy than is seen in MapReduce. Traditionally, a computing node is scheduled to execute a set of Map or Reduce tasks and once all nodes processing these like-tasks have finished the node may be re-allocated to other tasks. The single-host implementation of C-MR uses a, pull-based data acquisition mechanism allowing computing nodes to execute any Map or Reduce workload as they are able. In this way, they fetch data from the intermediate workflow buffers, process the data, and deposit the data into downstream intermediate workflow buffers. Therefore, the computing nodes communicate with each other via shared memory and avoid buffering data to disk . We have outlined the physical architecture of our system in Figure 3.2.

3.4.1 Host

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 and/or GPUs on the computer and launching threads for each of those we wish to use. The `Host` will then instantiate a variety of Map and Reduce operators, as specified by the application programmer, which are accessible by the nodes to execute tasks via the corresponding application code. With the workflow instantiated as a directed acyclic graph of Map and Reduce operations, the `Host` will proceed to attach input streams to the workflow.

3.4.2 Computing nodes (CPU & GPU)

The `Host` launches a computing `Node` thread for each core (or GPU) available on the computer. The thread is confined to run only on that core with the use of the Linux `sched_setaffinity` function to prevent the operating system from scheduling `Node` threads on the same computing resource – we do this because we want to manage the scheduling of our processing resources ourselves. Each `Node` will constantly try to consume data as it is directed by a scheduler. When data are issued, `Node`s will process them with respect to the corresponding operation and forward the results to downstream workflow buffers. A `Node` will give priority to handling replicated punctuations, pushing them back into the workflow as they arrive, to encourage the materialization of windows which may contain

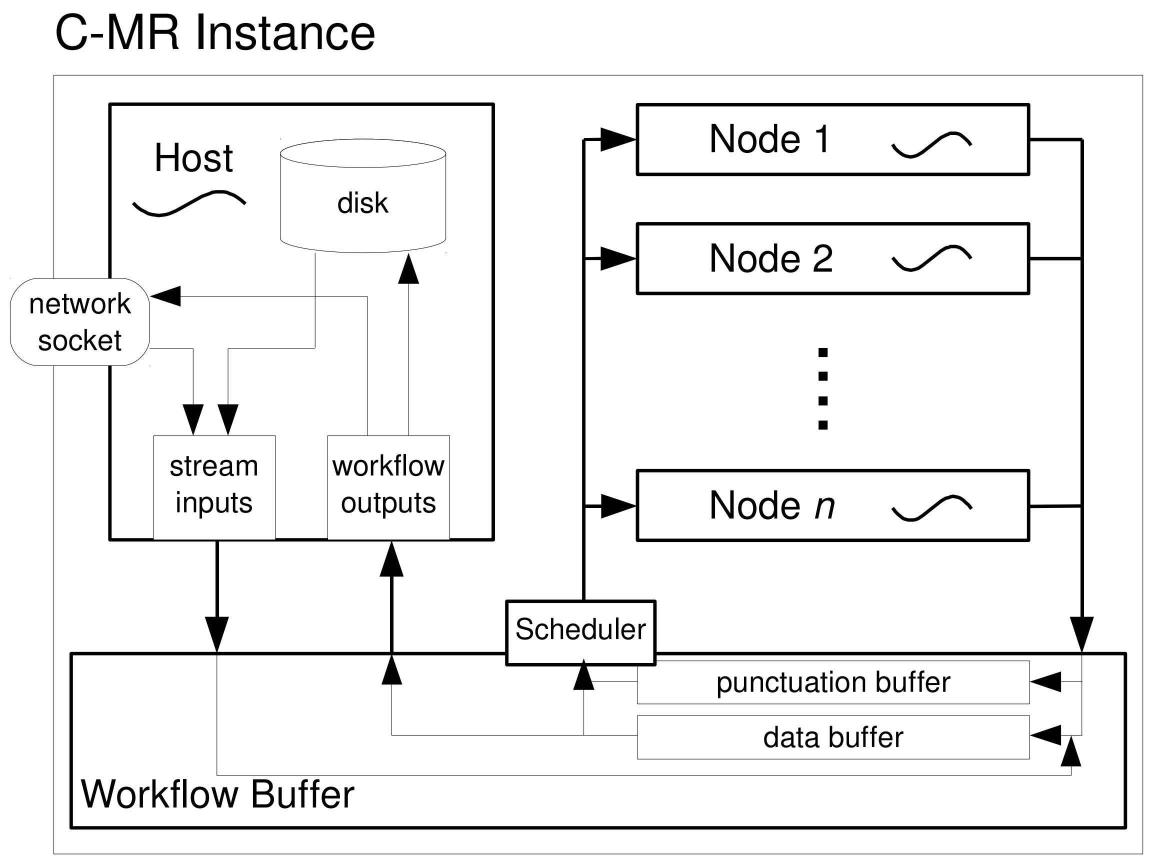


Figure 3.2: C-MR architecture
The host invokes a node thread for each local processor/core and then manages the workflow inputs and outputs while the nodes asynchronously execute the arbitrary tasks they are issued by the scheduler.

high-priority data to be made available to the scheduler.

3.4.3 Workflow buffer

Intermediate data in the workflow are stored in a staging area, known as the `Workflow Buffer`, to be materialized into windows and/or consumed by `Nodes`. To ensure that the temporally aware Reduce operators consume an ordered stream, it is necessary to collect and sort their data in these intermediate buffers in anticipation of window materialization.

To sort the stream, punctuations are inserted into the workflow by the `Workflow Buffer` at a location upstream from each Reduce operation. The punctuations are inserted at intervals corresponding to the Reduce window's slide value and at a location where the stream is already sorted (e.g., at the input to the workflow or just prior to an upstream Reduce operation whose input has just been sorted). Punctuations move along the stream like any other data and once they leave the sorted intermediate streams they are replicated to all `Nodes`. They are then consumed by the `Nodes`, and delivered back to the `Workflow Buffer`. The receipt of the last punctuation denotes that all data has arrived for the window that is to be reconstructed. Upon receiving all of these punctuations at the `Workflow Buffer`, the corresponding window will be materialized and made eligible to the `Scheduler` for propagation along the workflow.

3.4.4 Scheduler

As `Nodes` go to the `Workflow Buffer` to consume data, they first interact with a `Scheduler` routine. The `Scheduler` returns data to a `Node` from the `Workflow Buffer` based on the current scheduling policy in place. The `Scheduler` and its related scheduling policies are discussed in more detail in Section 3.7.

3.5 Workload partitioning

As streams are connected to a C-MR host, key-value pairs will begin to arrive off the stream as defined by the programmers input parsing function. This workload must then be divided among the computing nodes that will be processing it.

3.5.1 Partitioning types

Data can be partitioned in multiple ways and across multiple dimensions. When partitioning data for parallel processing however, we must be conscious of any relationships between data and operations which require some data to be grouped together (aggregations of like data, for instance). In this way, a sub-stream containing Map data can be partitioned arbitrarily among computing nodes as the data for Map tasks are processed independently. On the other hand, Reduce data must be partitioned along two dimensions. First, data must be partitioned into sub-streams which consist of like keys. Then, the resulting sub-streams must be partitioned along the time dimensions to support windowed analysis as defined by the application programmer.

Content-based

Whenever the grouping of data, as we partition a stream, is a result of the contents of the data itself, then we are performing content-based partitioning. In the case of MapReduce, Reduce data is partitioned by key. While Google's MapReduce stores keys as strings, C-MR allows keys to be any type and simply stores a pointer to data of an arbitrary type. We simply ensure that the keys containing identical data are grouped together. Again, for Map, the partitioning type can be arbitrary as the data can all be processed independently.

Context-based

We define the event of partitioning data by its metadata as context-based partitioning. For instance, in the parallel processing of video streams, an individual video frame in a sequence of image can often be partitioned into subregions. In this way, the data within each subregion itself does not dictate how the data is partitioned. Instead we use auxiliary information to determine how to partition each frame within the stream. Likewise, the partitioning of a stream into temporal regions, for the purpose of temporal aggregation via windows, is another form of context-based partitioning. In C-MR we use punctuations to denote the boundaries of these partitions in the stream.

3.6 Load balancing

After the workload has been partitioned into pieces that are consumable by computing nodes, we must make the decision regarding where each of these data should be processed. There are two fundamental forms by which a computing node can consume data. Data can be allocated to a computing based on the node's current workload, or they can be retrieved by the nodes as the nodes become available to process new data.

3.6.1 Push-based data allocation

If data are to be delivered to computing nodes as they arrive, then the data allocator should be aware of the cost of the data to be allocated and the cost of the data that the node has already been allocated but has yet to finish processing. These data costs are relative to the computing potential of the computing nodes. For instance, a data item may be quickly processed at a dedicated node and processed more slowly at a non-dedicated node.

For the sake of load balancing, and the minimization of the latency of results, we would ideally like to deliver a new data item to the computing node who will be able to finish processing it soonest. This requires a significant amount of profiling of both data and computing nodes in order to determine the average processing cost of data in addition to the current processing potential of a computing node. Even if the computing nodes on a C-MR host are identical, it may be possible that external workloads are running in the background on some of those computing nodes and are contending for computing resources.

When dealing with advanced allocation, it is required to make assumptions about the future. One must assume how long it will take to process the remaining data a node has been allocated, assume how long it will take to process the new amount of data, and hope that the computing potential of the node doesn't change in the meantime. If we are dealing with a static environment with homogeneous, dedicated nodes and no complexity skew in the workload, then we can be at ease with some of these assumptions. Otherwise, if these assumptions are wrong then we will experience disparities in what we would have hoped were balanced workloads.

3.6.2 Pull-based data consumption

Given that this implementation C-MR is for a single-host architectures, we have another option available to us for balancing load between computing nodes. As described in Section 3.4.3, the Workflow Buffer, which stores all intermediate workflow data, is stored in shared memory. This results in quick access to the buffer by all nodes. In the previous scenario, for advanced data allocation, data allocated to a node is similarly located in the shared memory buffer thus access times are similar. There is, however, a small degree of contention as the centralized Workload Buffer is accessed by the nodes through a locking mechanism to allow for thread safety.

The benefits of on-demand data consumption lie in the ability for multiple computing nodes to collectively consume from the same hypothetical queue. Using this method, it is not possible to to accidentally over or under-allocate data to any computing node. Instead, a computing node will naturally consume data at the rate which it is able to. In this way, slower computing nodes will less frequently visit the Workflow Buffer for new data while the faster ones will. In our observations, contention over the Workflow Buffer locks for operator queues did not present a problem given the relatively small number of cores on the commodity hardware we used and due to the cost of processing and managing data exceeding that of scheduling.

3.7 Operator scheduling

The C-MR processing model uses generic computing nodes to consume data, in a pull-based manner, from the centralized Workflow Buffer as directed by the scheduler. We have structured our scheduler in a way that it can leverage a number of different *scheduling policies* based on a variety of metrics.

3.7.1 Scheduling Policies

Nodes invoke a scheduler routine that is responsible for selecting prepared data to process from the intermediate workflow buffers as defined by a scheduling policy.

These policies evaluate the input queue to each workflow 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 operator queues with higher ranks. Table 3.4 defines the methods that the policies will use to rank an individual operator.

	Ranking Expression applied to Operator
ODF	-(timestamp of data at front of queue)
YDF	(timestamp of data at front of queue)
CTO	-(breadth-first-traversal order from sink)
MEM	$\frac{\text{avgInputSize-(selectivity)(avgOutputSize)}}{\text{avgProcessingTime}}$

Table 3.4: 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. Conversely, YDF uses the timestamp as-is to give newer data (with higher timestamps) the higher rank. The CTO strategy assigns the rank to each operator according to its ordinal value in a breadth-first-traversal of the workflow starting from the output. These values are negated, such that the values closer to 0 were visited earlier in the traversal and would therefore be ranked highest. Lastly, MEM observes the expected difference between the average size of an input tuple and the average size of an output tuple of an operator with consideration of the operators selectivity. This difference (the expected loss in memory usage) is then divided by the processing time to assess the expected number of bytes reduced per unit of time.

One advantage of having generic computing nodes with access to all data in a workflow is that scheduling policies will allow nodes to collectively concentrate their efforts on hot spots within the workflow. They will all collectively process the area in the workflow that the scheduler deems most important. In contrast, strategies which pin continuous Map and Reduce operators to nodes[19, 42] can find their computing nodes under or over-burdened due to variations in the stream, workload, or system resources. For instance, if a burst of data targets a single operator in the workflow, only the nodes which the operator has been pinned to will be able to face this burden. In our system, however, any computing node will be capable of contributing to alleviate this burden. Our operator scheduler assures us that a computing node will not sit idle if there is ever data available to be processed anywhere in the workflow.

3.7.2 Hybrid & probabilistic scheduling

The arrangement of generic computing nodes and a centralized scheduler affords us the ability to use multiple scheduling policies concurrently by executing new policies on each node's request for data. We present a novel hybrid and probabilistic scheduling framework which enables the gradual transition between scheduling policies based on resource availability. This strategy allows the application to benefit from the benefits of multiple scheduling policies as the need arises. For instance, we instantiated a hybrid ODF and MEM scheduling policy that aims to keep latency minimization a priority while also trying to prevent the depletion of available main memory. When the amount of memory remaining becomes sufficiently limited, the hybrid policy begins to increase the probability that a node's request for data will be satisfied with the use of the MEM strategy.

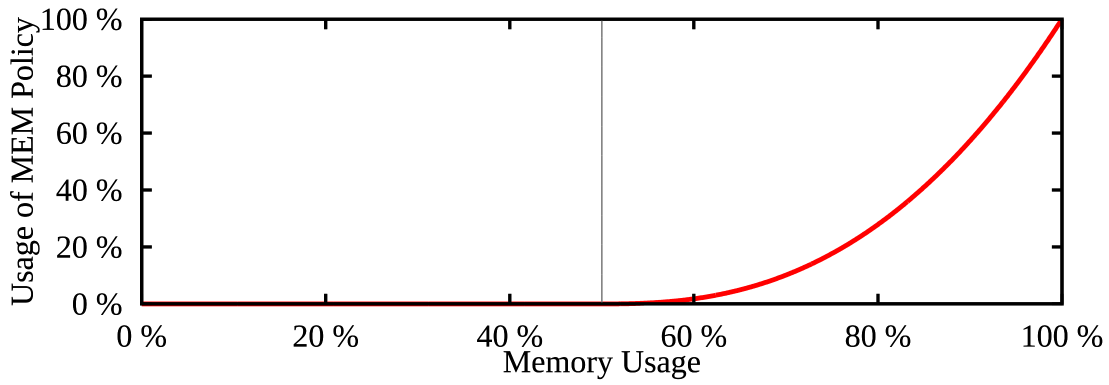


Figure 3.3: Progressive scheduling policy
This CDF shows the rate at which our hybrid progressive scheduler transitions to the MEM policy as available memory becomes more scarce. Here we use a transition threshold of 50% resulting in the remaining interval being represented as a beta CDF with parameters $\alpha = 2.5$ and $\beta = 1$.

We used this progressive scheduling arrangement in order to improve end-to-end latency when conditions in our system change in a way such that one specific policy is no longer advantageous. We generally find that ODF does a very good job of minimizing end-to-end latency, however, when experiencing bursty loads resulting in the sudden depletion of available main memory, the ODF policy begins to perform extremely poorly as swap space begins to be used. For this reason, we pair the ODF and MEM policies in the progressive scheduler in a way that leverages ODF when memory is not limited and gradually transitions over to the MEM policy when in danger of hitting swap space. Here our goal is to achieve the best of both worlds — trying to maintain a focus on minimizing latency while protecting ourself from exhausting the memory which would result in an increase in end-to-end latency.

To model this transition, we used the cumulative distribution function of the beta probability distribution. The beta distribution provides us with a continuous probability distribution on the interval $[0, 1]$. We use this interval to define a transition to the MEM policy once memory usage exceeds a threshold, T, signifying that it would be beneficial to begin the transition between policies. In Figure 3.3 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 3.3 which is expressed mathematically in Equation 3.1. The result is the probability that we 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 $\mathrm{rand}(0, 1) < F(x)$, then we use MEM and we would use ODF otherwise.

$$F(x) = \begin{cases} 0, & x < T \\ \mathrm{BetaCDF}(\frac{x-T}{1.0-T}, \alpha, \beta), & \text{otherwise} \end{cases} \tag{3.1}$$

We used the Beta CDF, which can be parameterized to be as linear or as exponential as desired, to allow application programmers to define for themselves the rate at which they would like to transition between scheduling policies. This desired rate of the transition may depend on the type of application being used or may simply depend on the personal preference of the application programmer – an especially risky person may wish to use an ODF policy for as long as possible before resorting to a sudden increase in the MEM policy upon expiration of main memory.

3.7.3 Analysis

The combination of generic computing nodes and our scheduler allows us to consciously schedule complex workflows with regard to end-to-end application performance requirements. We evaluate our scheduling policies on the workflow depicted in Figure 3.4. The Map and Reduce operators have been assigned varying selectivities, resulting output value sizes, and processing times.

The results of this test, including a burst in the stream at time 10 (like the previous test), are shown in Figure 3.5 for both latency and memory consumption.

The results show that ODF is the clear winner with respect to minimizing latency. The remaining policies all suffer substantially at the onset of the burst. It is interesting to note that, with this particular workflow, the ODF policy also results in the minimum memory consumption during the burst time, beating out the MEM policy. This is likely due to the fact that ODF operates to push data through the workflow which can result flushing data and punctuations to materialization points

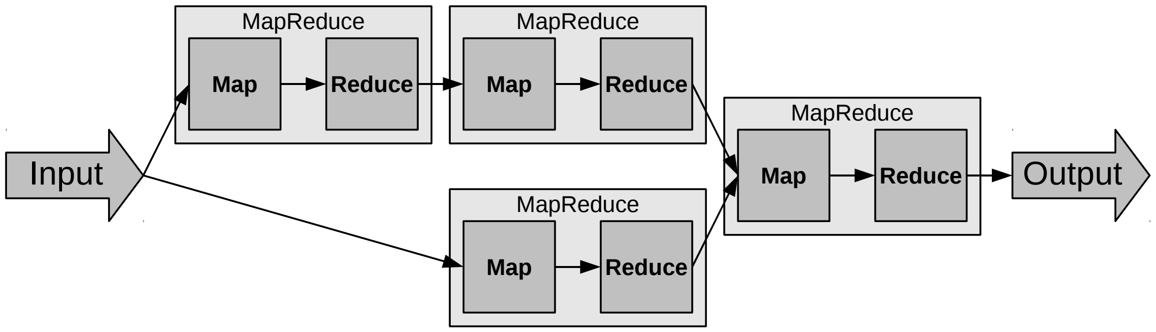


Figure 3.4: C-MR workflow to evaluate scheduling policies

where better memory trade-offs can be achieved. A policy that considers cumulative downstream efforts could help remedy this effect.

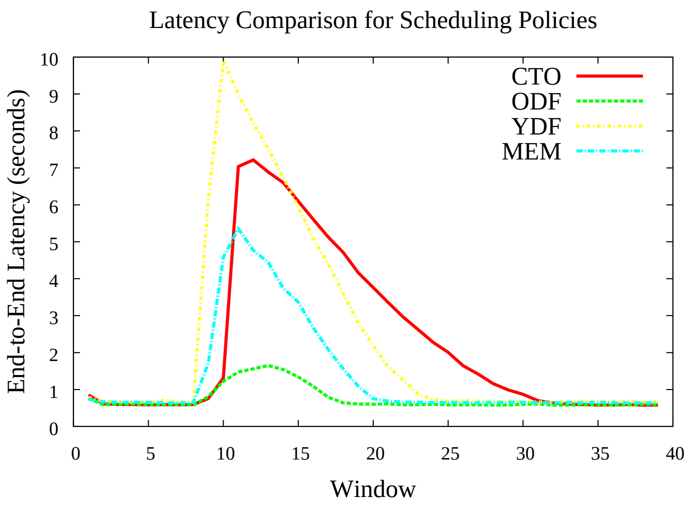
The MEM policy actually excels for many other workflow configurations, including those where highly selective/filtering operations are located early in the workflow which is often the case. For such a workflow, we tested our hybrid scheduling policy which transitions between the ODF policy (with the goal of general latency minimization) and the MEM policy (to prevent hitting swap space) in the hopes of preventing huge spikes in latency.

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 only 2GB of available RAM. Figure 3.6 shows the results for both latency and memory usage.

Initially, we find that the Hybrid policy mimics that of ODF with respect to both latency and memory consumption. It is only once the memory footprint of the stream application starts to significantly increase that we see Hybrid deviate and become more memory concious. From the latency graph, we find that, after the burst hits, the Hybrid policy always outperforms MEM with respect to latency and is only outdone by ODF just before ODF goes into swap space due to ODF's recklessness. The hybrid policy consistently provides good latencies throughout the experiment.

3.8 Re-purposed Combine phase

In C-MR we provide an optimization which re-purposes MapReduce's optional Combine phase to process sub-windows which are common to overlapping Reduce windows. This concept was initially explored for traditional stream processing window evaluation in [29]. We represent the C-MR Combine phase as an extra Reduce operator in the workflow, situated between Map and Reduce operators, with a window size that is a common factor of the Reduce window's size and slide. An



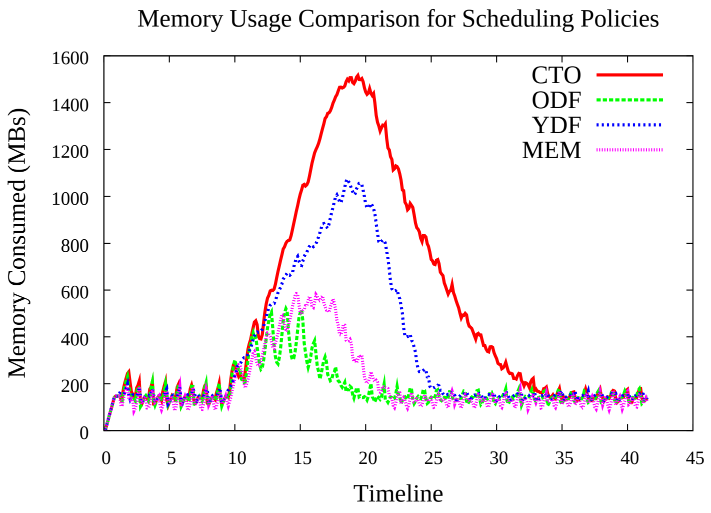
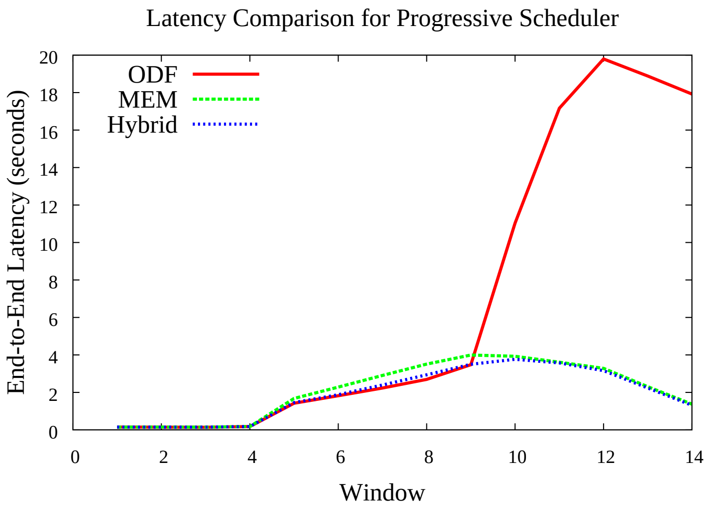


Figure 3.5: Operator scheduling policy comparison

example of this type of workflow can be seen in Figure 3.7. The sub-window size and slide values are equal such that there is no overlap between sub-windows. Such a window is referred to as a *tumbling* windows.

These tumbling windows allow for the incremental aggregation of the full Reduce windows as processing occurs before the full reduce window has materialized. The results are computed only once and can be integrated with the corresponding downstream Reduce windows. The only redundant computation remaining is the aggregation of the results of the sub-windows at the Reduce operator. The cost of this redundancy equates to simply aggregating over a value per sub-window in the Reduce window – aggregating aggregates.



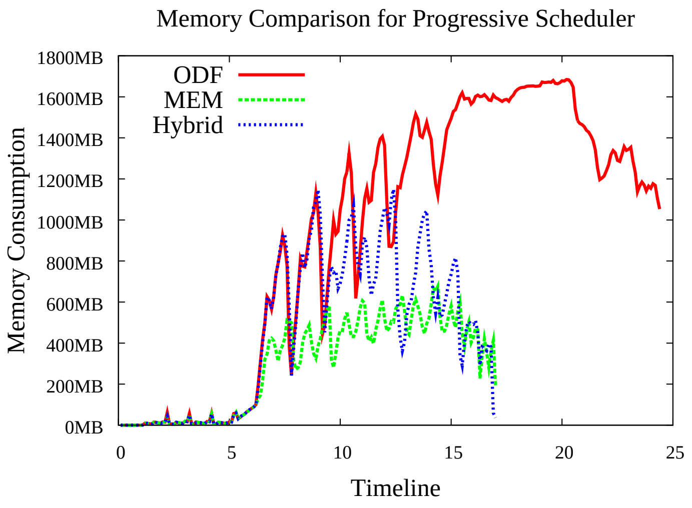


Figure 3.6: Progressive scheduling results

3.8.1 Sharing sub-windows

The precomputed sub-windows produced by Combine can be shared among all other Reduce windows that happen to share the sub-window. For instance, if a Reduce operator aggregates over 60 minutes worth of data every 5 minutes and Combine sub-windows have a size and slide of 5 minutes, then after the workflow has been primed with data each new Reduce window can be computed with the results of the previous pre-computed 11 sub-windows provided by Combine and the latest freshly computed sub-window produced by Combine.

C-MR also allows for the sharing of sub-windows between Reduce operators of similar types but have differing window sizes. For instance, a Reduce operator that has a size of 60 minutes and a

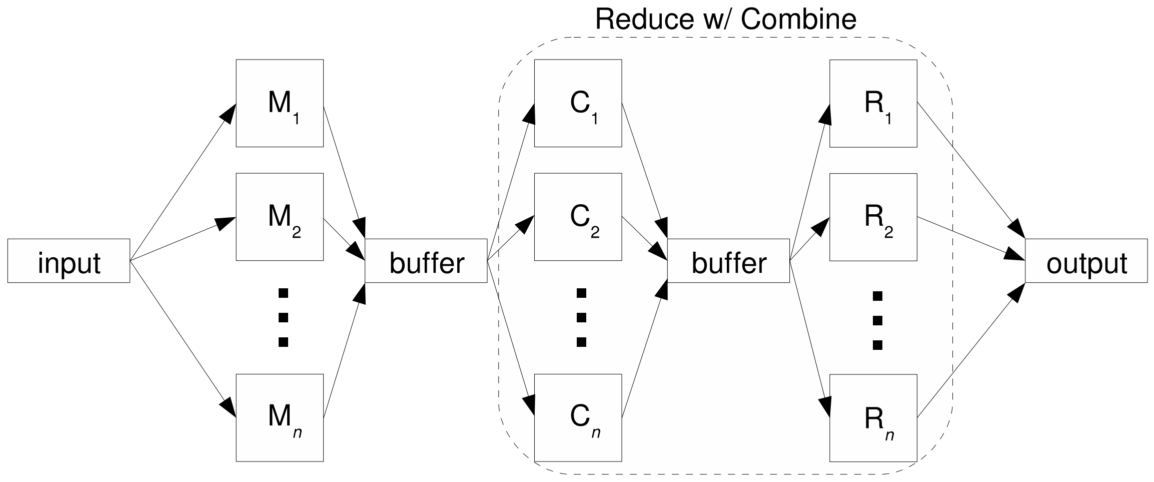


Figure 3.7: Re-purposed Combine phase
C-MR can use Combine to process sub-windows of Reduce operations. This encourages incremental processing of Reduce windows, data parallelism within Reduce windows, and mitigates redundant processing of shared sub-window regions across temporally overlapping Reduce windows.

slide of 5 minutes can share Combine sub-windows with another Reduce operator that has a size of 45 minutes and a slide of 5 minutes as long as those operators both have the same input sources and have a common factor for the slide values.

3.8.2 Overhead analysis

To determine if a C-MR Combine phase will be worthwhile in a workflow, we compare the expected average 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 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 within such windows.

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

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

$$\begin{aligned} wt &> st + \tfrac{w}{k} \\ (wt) - (st + \tfrac{w}{k}) &> 0 \\ t(w - s) &> \tfrac{w}{k} = \tfrac{w}{HCF(w,s)} \end{aligned}$$

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 it's 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.

3.9 Experimental results

The tests in this section were performed on a computer with an Intel®Core^{TM}2 Quad Processor Q6600. The following tests involve stream applications that determines the moving average of stock prices over time. The applications use 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. Example code can be seen in Appendix A.

To facilitate these application, we replayed NYSE stock data from the TAQ3 data release of January 2006 [6]. 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.

3.9.1 Continuously execute a MapReduce job

Naïve windowing can easily be integrated into traditional batch-oriented implementations of MapReduce. Without making changes to these implementations, and by treating them as narrow MapReduce interfaces, we can manually construct time-based windows and then invoke the batch-oriented MapReduce framework to process them. We would then shift the input buffer by the slide-amount of the window. This results in large amounts of data replication (to maintain data both outside the framework, for stream management, and within), redundant processing (at both Map and Reduce operations due to overlapping windows), and an inability to incrementally process data (due to having to wait for the entire window to arrive before being able to invoke the MapReduce job) which all contribute to a higher latency.

Additionally, this method of supporting stream processing with MapReduce places the burden on the application programmer to perform stream and window management themselves. We implemented exactly this kind of stream and window management to invoke MapReduce jobs with the Phoenix++ framework. Phoenix++ Is a single-host, shared memory, multi-core implementation of MapReduce in its third major revision [39]. We compare repeated invocations of Phoenix++, supporting stream processing, to a C-MR MapReduce job and a C-MR MapReduce job which leverages a Combine operation, interposed between the Map and Reduce operators, to decrease that amount of redundant Reduce computations that occur as a result of overlapping, sliding windows.

This experiment uses a single MapReduce job which determines the moving average of stock symbol prices over a data stream. Figure 3.8 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 incurred by Phoenix++ ultimately hurt latency; as the size of the workload increased (with larger windows), the latency of Phoenix++ increased more rapidly than the C-MR variants. C-MR hit the overload point (where the CPUs became saturated with work to do) just prior to Phoenix++, due to additional storage and organization requirements of intermediate data. C-MR with Combine, however, supported the processing of significantly larger workloads before reaching saturation.

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

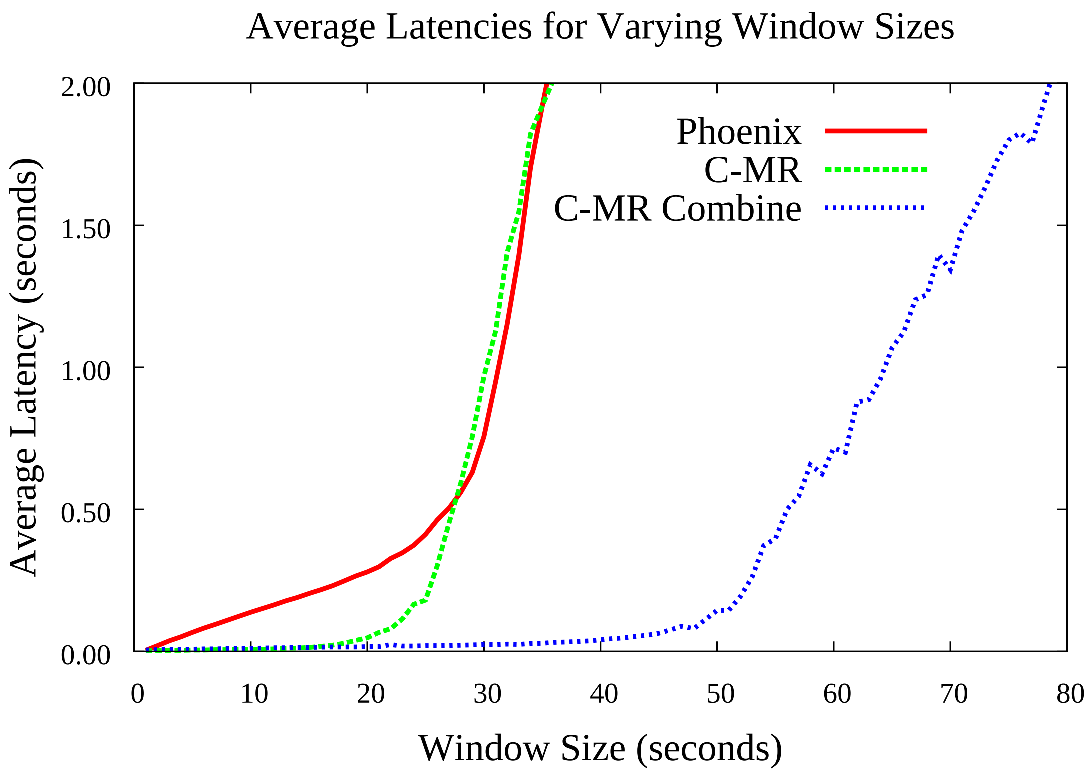


Figure 3.8: C-MR vs streaming Phoenix++ implementation
Repeatedly invoking a Phoenix++ MapReduce job over a stream results in many redundant computations (at both Map and Reduce operations). C-MR allows data to be processed only once by Map and the inclusion of the Combine operator significantly decreases redundant work performed at the Reduce operator.

3.9.2 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 3.9, 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

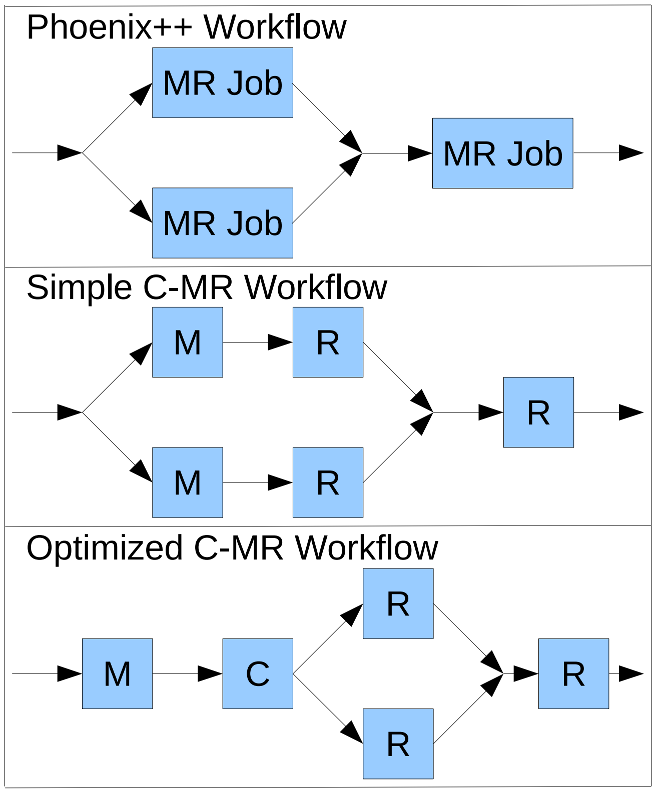


Figure 3.9: MACD workflow implementation strategies

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 to populate their windows. 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 merged and processed 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 3.10. 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 outperforms the simple C-MR workflow with a 31% decrease in average latency.

We also replayed the same stock data stream as one large batch which arrived instantaneously

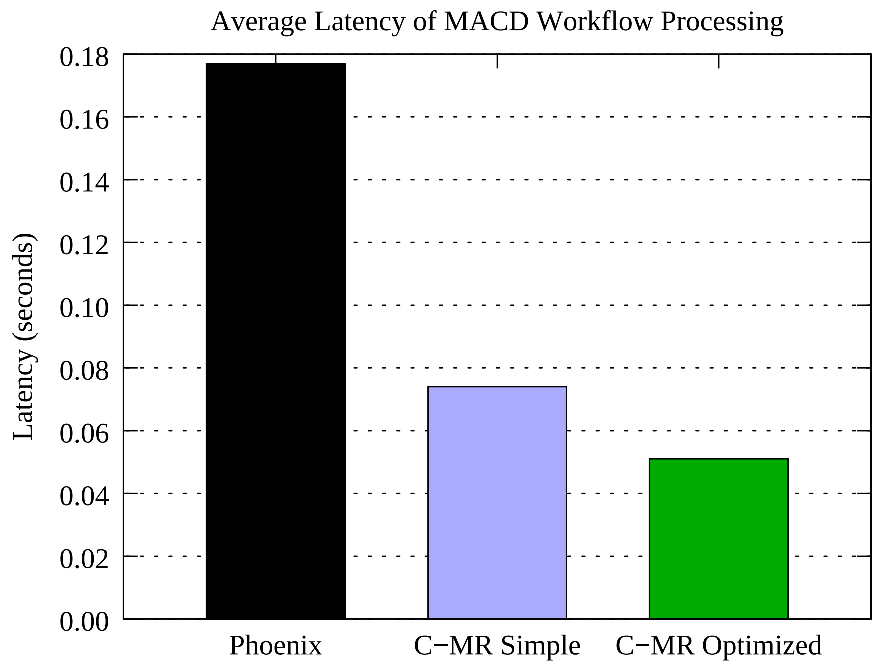


Figure 3.10: Latency of MACD workflow implementations

to perform a throughput. The results of this test can be seen in Figure 3.11. 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.

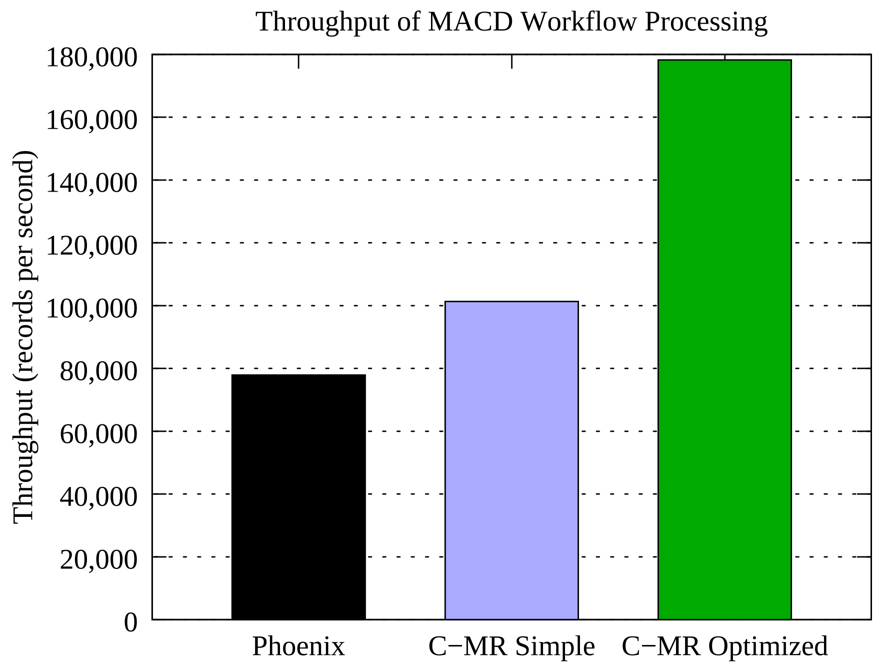


Figure 3.11: Throughput of MACD workflow implementations

Chapter 4

Distributing C-MR

The C-MR framework presented thus far has been suitable for supporting stream applications whose computational requirements do not exceed that of a single high-performance computer. The next natural step is to support applications which leverage larger amounts of data and data with higher processing costs to achieve higher throughputs while maintaining low latencies.

We have extended the C-MR framework to support its distribution onto multiple computing resources. This chapter explains the requirements and challenges that were faced to accomplish this endeavor as well as our implementation details and an analysis of the benefits achieved.

4.1 Requirements

Distributing the C-MR framework to multiple physical computers (Hosts) was a task that required both extending the C-MR architecture and considering the general requirements of large-scale distributed applications. This section discusses both required extensions to the C-MR architecture and the requirements posed by modern distributed applications (as motivated by the prevalence of the cloud computing and infrastructure-as-a-service models).

4.1.1 Distributed application requirements

For most of the past decade, there has been a tremendous increase in the cloud computing movement. This movement champions the concept of delegating the management of complex infrastructures and software services to third parties. These third parties manage the hardware and/or software that the client wishes to use.

Cloud services lower the barrier to entry for leveraging large amounts of compute power. It is no longer necessary to acquire and facilitate one's own personal cluster which would require the management of hardware, networks, power, failures, and possibly even heating, ventilation, and air conditioning services. Instead, through these cloud services, a client can rent access to the infrastructure or software platforms they need for the period of time that they desire. Infrastructure-as-a-Service (IaaS) providers give clients the ability to host their own virtualized computers that are entirely within their control. A client could then pay IaaS providers to launch compute hosts to facilitate their data processing needs. Platform-as-a-Service (PaaS) providers abstract away the management of infrastructure from the user while allowing software solutions (including data processing frameworks) to be deployed across resources hidden from the end-user. In this way, a client will simply pay to use the service or frameworks that the PaaS provider offers.

Batch processing and stream processing frameworks, such as C-MR, may be hosted by a user explicitly through IaaS offerings or accessed indirectly through PaaS offerings. Examples of cloud computing service providers (including IaaS and PaaS) are Amazon EC2[7], Windows Azure[8], Rackspace[9], and Linode[10].

The increasingly prevalent cloud-computing ecosystem encourages the use of distributed applications which can scale with ease. In addition to being readily scalable, these ecosystems also encourage elasticity. Elasticity, in this context, is the on-the-fly addition or removal of hosts from the working set of computing resources that are facilitating the application. Cloud computing clients need only to pay for the resources they use, therefore it is advantageous to decrease the resources used when in a state of underutilization and increase the pool of resources that are used when there is an increase in computational demand.

Stream applications are often volatile in their processing requirements as fluctuations in stream volume can occur over time. Therefore, they are an ideal candidate for cloud computing services which allow elastic support. As we designed and extended C-MR for distributed use, we kept scalability and elasticity in mind as a necessary features to support.

Scalability

Data volumes continue to increase at an incredible rate. Data commonly processed by stream applications include event logs, click streams, image/video feeds, network traffic, and various other data feeds. The increase of network-attached devices, data collection, and the pervasiveness of social networking into our lives has provided a goldmine of data that companies are rushing to analyze. Therefore, processing frameworks facilitating the analysis of such huge volumes of data must keep up by distributing and managing these computations across a similarly growing set of computing resources. As 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), it is important that we use distributed resources in a manner that efficiently meets the objectives of the stream applications.

Elasticity

Data streams are often characterized by bursts and volatility – the amounts and types of data a stream may contain can change over time. For example, an application analyzing network traffic may observe higher loads during business hours while experiencing much smaller load between 2-5am. We therefore find ourselves in the situation of having to meet the performance requirements of stream processing applications while being faced with regularly changing processing requirements.

The computing resources that processing frameworks use are certainly not free. Many applications compete to use these resources and therefore it is important to not acquire more computing nodes than is needed to meet the performance objectives of the application at hand. There is also a large financial cost to hosting and facilitating one's own large cluster of computing nodes which has opened up a large space in the market for cloud computing in which processing jobs can be shipped off to virtualized sources. Using such cloud computing infrastructures results in monetary fees for each virtualized computing node used for the time that they are acquired (whether in use or not). This results in the goal, yet again, to only use the computing power that an application will require to meet its performance objectives or money will have been wasted.

With various computing resources available for use, whether local or in the cloud, it is important for modern stream processing frameworks to support the real-time addition or removal of these resources on-the-fly to ensure that application objectives are met while minimizing the cost of using them.

4.1.2 Extensions to C-MR

As presented in the previous chapter, the single-host C-MR architecture supported multiple processing elements (CPUs/GPUs) while facilitating all of the intermediate workflow data within shared memory buffers accessible by each of the processors. The move to a distributed architecture necessitates a variety of changes to C-MR which are outlined below.

Multi-host communication

The first, and most obvious, difference between C-MR and a distributed version of C-MR is that multiple hosts are being leveraged in the distributed version. Therefore, there must be a communication mechanism established so that hosts can transmits control information as well as data between each other. Further, it is necessary to determine what kind of communication model should be established.

As our goal is to focus on user-provided performance goals, such as latency, a large part of our work deals with predicting performance of stream applications being executed. To accomplish this, it is important to coordinate information and statistics between all of the participating hosts so that we can provide a global analysis of the processing environment. To this end, we use a client-server model such that client hosts produce statistics to a server (a "master" host) which then performs load balancing and directs the client in regard to the specific workloads they will be responsible for. These control data (statistics and routing policies) are communicated via this client-server model.

Application data, on the other hand, may be communicated from any one host to any other host participating in the application. This is the nature of Reduce-like data – Reduce keys are not always predictable, and therefore it is possible to generate data from a Map operation that may have *a*ny possible Reduce key. This permits the possibility of having all-to-all communication within the Reduce step in which the data are not routed through the master.

Distributing workflow buffers

Stream processing framework are meant to process unbounded streams with high throughput and low latency. Therefore, it has always been a design decision for stream processing engines to keep data in main memory whenever possible – accessing disk drives to fetch memory results in extraordinarily long latencies and is avoided at all costs. Thus, when handling larger stream volumes, it may be necessary to distribute workflow data to multiple hosts such that a single host is not overloaded and forced to push data into swap space. Likewise, there may be an incentive to distribute data for parallel processing in order to reduce end-to-end latency.

To distribute the shared memory workflow buffers in C-MR, it is important to remember their purpose. Both data and punctuations are stored in these buffers. Map data were instantly placed in operator queues, ready to be processed by the local processors, and Reduce data were batched into a staging area and waited for window-bounding punctuations. Once enough window-bounding punctuations had arrived (one for each of the local processors), then a Reduce window could be materialized and, like the Map data, placed in an operator queue to be processed.

With this in mind, distributing workflow buffers still requires the local processors within a C-MR host to replicate and propagate punctuations and it is now also necessary for hosts to do the same for the benefit of each other. An inter-host window-bounding punctuation scheme will ensure the arrival of data from all hosts before finalizing and materializing windows.

Assigning data to hosts

Communicating data between hosts was not a challenge faced within the previous implementation of C-MR. The single-host implementation kept data in the shared memory buffers until a processor requested data (via the scheduler) to consume. A processor would then take and process that data and then deposit it back into the workflow buffers. In this sense, the single-host implementation maintained pull-based acquisition of data. In shared memory, this strategy was convenient, fast, and effective at serving heterogeneous or non-dedicated processors at differing rates.

Employing a similar strategy of pull-based data acquisition at the distributed level between C-MR hosts, however, would result in a large increase in latency. This latency would occur during each request for data by one of the host's local processors. Such a protocol for requesting and transferring data for each of the local processors is not scalable and introduces too much wait time over the expensive network medium. Instead, data should use a push-based data distribution strategy which optimistically communicates data to hosts relative to their expected processing ability. If an error is made by under- or over-serving a host data, then data can be retro-actively migrated between hosts or expectations can be altered for the future push-based distribution of data.

It is also quite important to be aware of the types of processing that will occur on the data that are assigned to the hosts. Data meant to be processed by a Map operation may, literally, be sent to any host to be processed as each data belonging to a similar Map operation can be processed independently of the others. With data destined for a Reduce operator, on the other hand, the destination is quite important as the data must be grouped together by similar keys and then processed in batches defined by the temporal windows. Therefore, all participating hosts must send each data that is intended for a Reduce operator to a known and pre-determined location. This is even the case when two hosts produce data with the same and never-before-seen keys – they must be able to simultaneously, and without coordination, determine the location to deliver data for that specific key.

While the single-host strategy simply collected Reduce data of a similar key together within the intermediate buffers, it is now necessary to apply another higher-level mapping which associates such Reduce keys with a specific C-MR host. Once the data arrives at the appropriate host, it can be inserted into the local workflow buffer of that host and managed as it was in the single-host implementation.

4.2 Challenges

While the general types of challenges faced in distributing C-MR have been faced by other distributed data processing frameworks, C-MR imposes constraints which make many of those previous strategies ineligible for implementation.

Given n computing nodes to participate in workflow processing, we'd ideally like them all to constantly do valuable work when able; here, we define valuable work as being directed by the user-defined scheduler to accomplish workflow-wide goals. Inadequate load balancing and restrictive operator scheduling policies both result in underutilization of computing nodes. By not properly utilizing computing resources, stream applications can under-perform.

Our single-host implementation of C-MR leveraged centralized workflow buffers that lived in shared memory. The pull-based acquisition of this data, by computing nodes, resulted in the fine-grained balancing of the data of the entire workflow across the C-MR nodes. In a distributed environment, however, where the workflow buffers must be distributed and are therefore decentralized, an alternative method must be used to allocate workloads to hosts in a fine-grained manner for appropriate utilization.

4.2.1 Estimating latency

The principles used behind designing C-MR, from the beginning, have been to encourage computing nodes to perform work that is beneficial to improving the bottom-line of the stream application. As most stream applications are time-sensitive, this bottom-line most often refers to maintaining low end-to-end latencies and is therefore is our focus with regard to stream application performance. The generic-node processing model allows nodes to freely contribute to operators situated across the workflow as directed by the scheduler to assist in meeting application performance requirements.

In a single-host setting, the task at hand was simply to perform good operator scheduling. Load balancing was taken out of the equation as all processors were able to simply consume data from the same shared-memory buffers. In a distributed setting, however, we now have the added challenge of getting data to the distributed hosts. Once the data arrive at the distributed hosts, we must still employ the operator scheduling strategies that we used before. However, in order to effectively use the processors at the individual hosts we must have an understanding of their capacity to do work in order to meet the application's performance requirements. If there is a large load imbalance between hosts then we may find the case in which the processors on one host may spin idly while the processors on another hosts are overburdened. We must perform due diligence in attempting to balance the load such that all processors can simultaneously do good work.

Now, to ensure we can achieve load balancing configurations that support our applications performance requirements, we must be able to predict the benefits of possible load allocation configurations. In other words, if we have a large unallocated chunk of work to assign to the available hosts, we must try to determine how to divide that workload amongst those hosts such that the total workload can be processed the quickest. Applications contain many different types of workloads (represented by the operators within the workflow) and we must therefore develop a method for understanding the complex end-to-end relationship of load balancing across the workflow with respect to how such changes ultimately impact our bottom-line – average latency. Any transfer in load throughout the workflow of operators can have an impact in this regard and we must determine how to use the information available to estimate the result of potential workload balancing configurations which include the speculative addition and removal of hosts.

4.2.2 Enacting redistributions of load

The single-host implementation of C-MR did not have any form of explicit load balancing policy in place. The fact that the C-MR computing nodes could access the data directly from shared memory buffers enabled them to consume data in a pull-based manner without the need to pre-allocate data to the nodes. In this way, no single node was ever more burdened than any other. Instead, the nodes collectively processed whatever load existed.

With the transition to a distributed implementation of C-MR a pull-based data acquisition strategy is ineligible due to latency constraints, as was discussed earlier. Instead it is necessary to implement a push-based data distribution strategy which will naturally require a load-balancing strategy to determine the amount of data to push to each host. Such a load-balancing strategy would not only determine the amount of data each host would consume (as would be the case for Map operations) but must also determine which types of data a host will consume as well (in the case of Reduce operations aggregating values belonging to a specific key).

Employing even a standard load-balancing strategy is not entirely straight-forward within the C-MR framework. We must consider that hosts can be added to or removed from the working set of hosts in real time and that changes directed by the load balancer must be enacted synchronously by all hosts. This raises the challenge of implementing a mechanism to control the data routing policies related specifically to Reduce operations. For Reduce operations, we must ensure that, as the working set changes or as routing policies change, all Reduce data that are both of a like-key and share a window will be collected at the same host.

Therefore, the distributed implementation of C-MR must have synchronization primitives in place to ensure that Reduce data are routed to the correct locations from the participating hosts in the event of host addition/removal and load balancing.

4.2.3 Conserving network bandwidth

While Google's MapReduce[20] and other implementations, such as Hadoop[2], do not themselves support workflows of MapReduce jobs, it is possible to string together a sequence of jobs to support workflow execution. There are frameworks designed to do specifically this including: Pig[34], Nova[33], Hive[41], and Cascading[11]. The results of MapReduce jobs, and even their intermediate results, are written to disk. Therefore between the Map and Reduce phase, and between invocations of MapReduce jobs within a workflow, there is not a large degree of communication of data. Instead, MapReduce mappers and reducers are more commonly executed on the hosts which already contain the data to preserve data locality.

Stream processing applications attempt to perform finer-grained load balancing, and may therefore find it necessary to communicate inter-job data (for instance from a Reduce operation to a downstream Map operation) between hosts on-the-fly. As load balancing policies are enacted, this can impact where a host will send such data. However, unlike with data consumed by Reduce operations, we have some leniency with data destined for Map operations as such data can be processed anywhere – even locally. It should therefore be a design goal to minimize the amount of network transmissions that occur, with regard to data destined for map operations, while maintaining the workload proportions defined by the load balancer.

4.3 Design and implementation

To meet the requirements of a distributed stream processing framework, to support real-time stream applications and address the requirements and challenges listed above, we outline the following design decisions and modifications to the C-MR architecture.

4.3.1 Hierarchical application of C-MR processing model

The C-MR processing model used for the single-host implementation need not change to support distributed execution, but changes to the implementation to support that model *a*re required. We first simply re-define the definition of a computing node. In the previous single-host model, the computing nodes were represented by both CPUs and GPUs, whereas in the distributed processing model they can be represented by hosts (the computers themselves).

In this regard, the processing model does not change at all. The nodes still consume data from intermediate stream buffers and then produce their results to output intermediate stream buffers. In this way, the data works its way from the inputs to the outputs. The differences are now that: 1) the intermediate workflow buffers are now distributed across multiple hosts; 2) data are now propagated in a push-based manner; and 3) the computing nodes (now hosts) process data differently once they are received. The result is a hierarchical application of our processing model as seen in Figure 4.1. Once data are received by a host, they enter an inner C-MR instance which processes data in a manner that is quite similar to the single-host C-MR design.

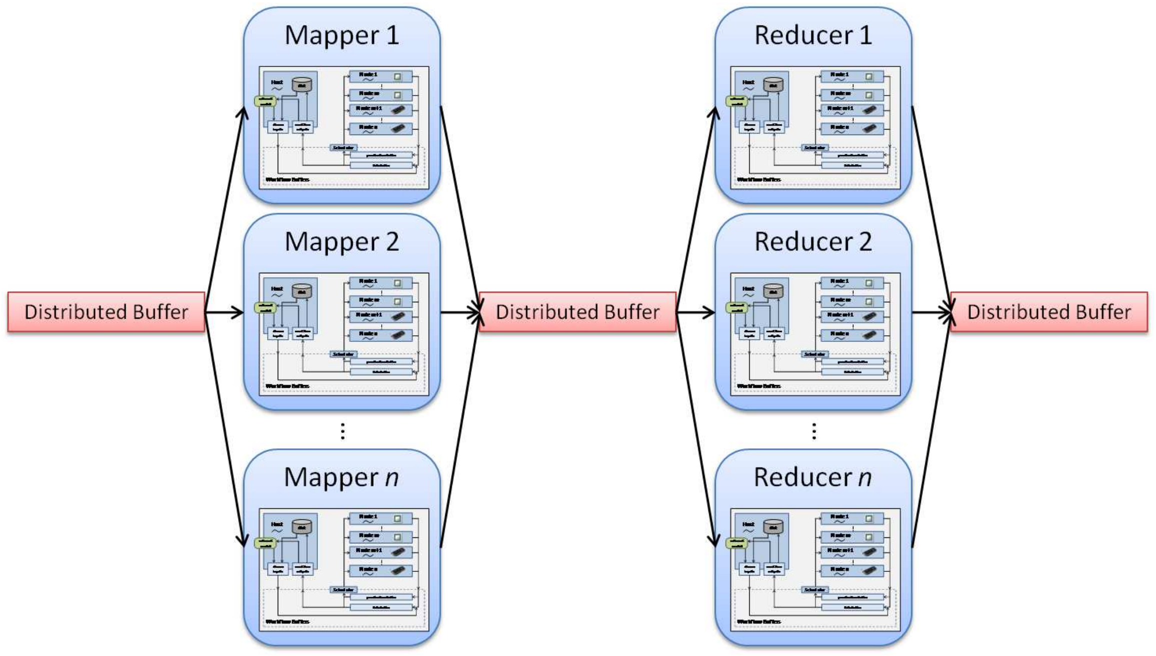


Figure 4.1: Hierarchical application of processing model

It is not difficult to see that this abstraction can be taken even further with another step up the hierarchy whereas clusters of hosts may be considered computing nodes. This could be done to maintain a degree of data locality if one were to increase the network of hosts that would facilitate the stream computations.

4.3.2 Hierarchical punctuation handling

By nesting a modified C-MR instance within each of the participating hosts, it is now necessary to have two layers of punctuation management to preserve stream order and bound windows. In fact, it is necessary to ensure stream order at any time in which a stream is partitioned. By using this hierarchical model, we effectively partition streams at two levels due to the fact that data parallelism is actually occurring at two levels – across hosts and then within hosts and across processors.

The stream ordering strategy defined in section 2.2 can be similarly applied, hierarchically, to this scenario. In fact, all intermediate workflow streams within the distributed variant of C-MR are always partitioned. Beyond the application inputs and outputs, there is no point, as there was with the single-host implementation, where the stream is entirely merged and re-sorted. Instead, it is necessary to maintain within each parallel stream the semantics of a window-bounding point.

Take for example a host waiting for the entirety of window x to arrive at a Reduce operation. This host can begin to materialize window x as soon as it knows that all other hosts have finished sending all material relevant to window x. Therefore, if those hosts each maintain their own individual window-bounding punctuations relative to window x, then our host will know that the relevant data for window x has fully arrived when it has received punctuations from each of those hosts denoting the end of window x. That is to say, that as long as each host's parallel stream remains sorted then it can produce a window-bounding punctuation downstream to notify other hosts that it has seen the end of its respective window.

The distributed punctuation passing differs from the single-host variant in that this scenario includes all-to-all communication between the hosts whereas the single-host variant entailed all hosts reading from the same, single input buffer and merging results to the same output queue.

Figure 4.2 depicts an example in which host i is waiting to bound a specific window. Given the existence of n different hosts, we can expect each of those n host to deliver window-bounding punctuations to the input of host i to denote that they have each seen the end of the window (Fig 4.2a). Once the host has received all n of those window bounding punctuations, then the host can certify that the window has fully arrived and may be materialized (Fig 4.2b). This materialization will trigger the release of lists of data for each relevant key in this window and may even result in the propagation of additional punctuations for subsequent downstream windows that must be materialized. If this is the case, then the stream would eventually be sorted yet again downstream requiring the introduction of new punctuations at this point to bound that downstream window. In this way, the inner-instance of C-MR would produce a new punctuation for each of m local processors (Fig 4.2c). After the processors have produced these punctuations into the merged output stream, we can be sure that the host's stream has again been bounded for this particular window (Fig 4.2d). At this point, the host must inform all of the other hosts (which it has been producing data to all along) that all of the relevant data for the window has been sent. It does this by sending to each host a punctuation denoting that host i has seen the end of the given window (Fig 4.2e). Those punctuations will travel down the stream (Fig 4.2f) and can later be collected as in Figure 4.2a.

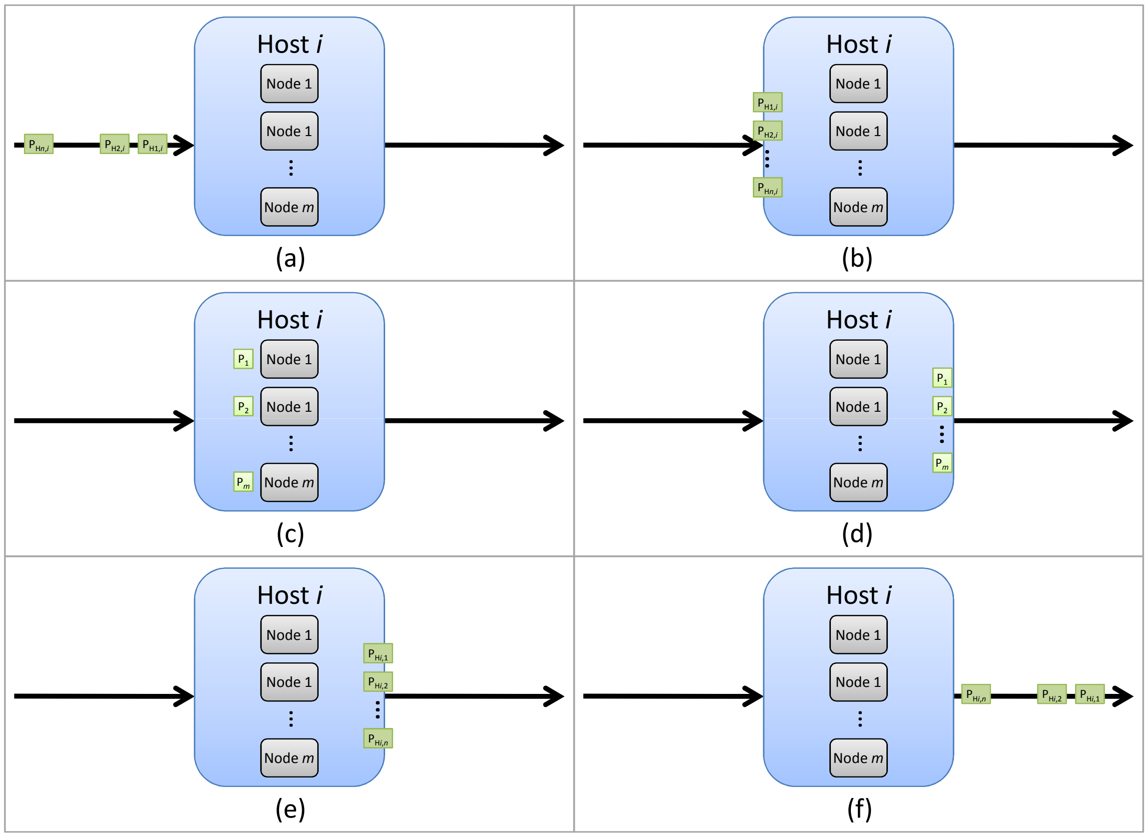


Figure 4.2: Inter-host and intra-host punctuation replication and collection

4.3.3 Distributed C-MR architecture

The architecture of the distributed implementation of C-MR differs from the single-host variant in quite a few significant ways. The single-host variant included the following three significant components:

- A **host process** which launches a node process for each of the CPUs/GPUs used and facilitates the connections from the application's inputs and outputs to the workflow.
- The **node processes** which repeatedly consume data from the workflow buffers via the scheduling policy while producing results back into the workflow buffers.
- The **workflow buffers** which buffer Reduce data and punctuations to materialize windows (potentially producing additional punctuations) and populate operator input queues with data (both for Map and Reduce) and punctuations that are ready for consumption by nodes.

The distributed implementation of C-MR makes a few significant changes to the previous architecture and adds another layer of complexity on top. It is now required to facilitate both client instances of C-MR (which will bear similarities to the single-host variant with new modules added to facilitate data/control message communication) and a "master" instance which has the following additional responsibilities:

- **Managing application inputs/outputs**: While the stream processing that takes place may be distributed, the inputs and outputs to the application may not be distributed and may originate at a single and specific point. Similarly, a programmer using this data processing framework will expect a single interface for stream application deployment. The master will receive the application inputs from a user and deliver the results back to that user. Therefore, an additional responsibility of the master will be to perform the initial partitioning of the input streams.
- **Load Balancing**: Any form of run-time optimizations or adaptive query processing ought to be motivated by improving application performance. Therefore, the distributed implementation of C-MR utilizes a centralized load balancer which attempts to predict the impact of the load balancing policies it enacts. This is done through analysis of the stream workload and client-reported indicators to a centralized point (the master) to enable estimations of the latency benefits of potential load balancing configurations.
- **Statistics collection**: To perform load balancing with any knowledge of how re-allocations of load will affect the application's bottom line requires the collection of statistics to estimate performance. Therefore hosts will regularly report statistics to the master so that this information can be used to keep the load balancer up-to-date with information concerning the stream workload and the participating computing resources.

The module supporting load balancing duties will only be invoked on the host that is responsible for consuming the application input streams and producing results to the end-user – the host providing the interface to the user. All other hosts will be running an instance of the distributed C-MR client to communicate with each other as well as the master. The newly modified architecture that will be run by all of the non-master clients can be seen in Figure 4.3.

As you can see, these clients now support the ability to consume their streams from an external source other than the application inputs. Data received are accessed via the Input Manager which listens for data from other clients (including the master). Data received from the Input Manager is placed into the Workflow Buffers from which Nodes can fetch data to process. As Nodes finish processing data they produce their results to a Routing Table facilitated by the Output Manager. If

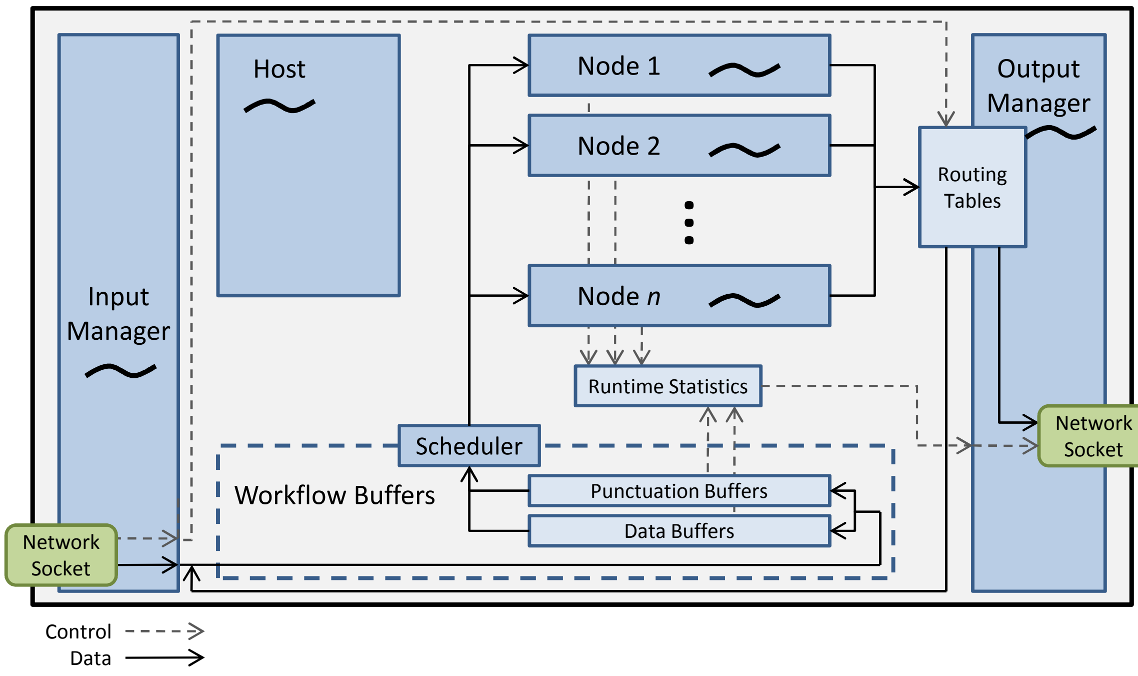


Figure 4.3: The architecture for the C-MR client that is designed for use on distributed hosts.

the results are to be propagated to another host then the Output Manager will transmit that data over the network. If the results can be kept locally for continued processing down the workflow then the data can be re-inserted back into the Workflow Buffers.

As the application runs, the Nodes (processors) report observed statistics regarding data processing times and the wait times exhibited before processing. The Output Manager periodically reports these statistics to the master. There they provide the necessary information to update routing tables in order to balance load across the available hosts.

The Host thread for clients does very little. It only serves to initialize the Node threads and the workflow buffers. The master, on the other hand, makes much more use of the Host thread. The master host uses the host thread, as in the single-host architecture, to read in the application's input streams and to write to the application's output streams. Another extra responsibility is the role of performing load balancing. Figure 4.4 shows the architecture for a master host with the additional responsibilities of the Host thread as well as the load balancing modules.

As the master reads from the application's input streams via the Host thread, the results are sent directly to the routing table of the Output Manager. This is where the initial partitioning of the input streams takes place. Data can either be directed to other client instances of C-MR on the available hosts or can be consumed locally at the master by being inserted directly into the local

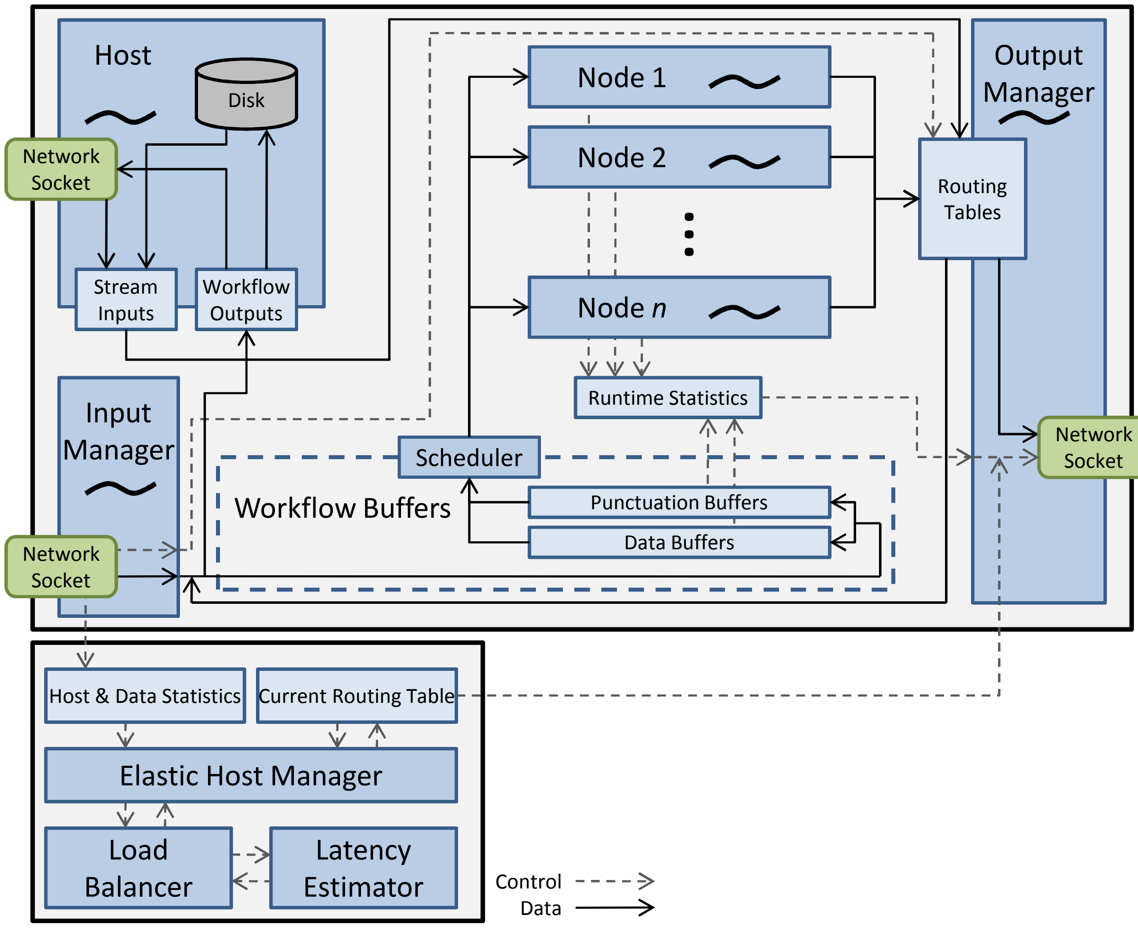


Figure 4.4: The architecture for the C-MR master.

Workflow Buffer. As the master is also the terminal point for all workflow results, data may be received from the Input Manager (from any of the clients) and directed to the Host thread to be produced as the application's results.

The master also includes an additional load balancing module as it is responsible for performing the centralized updating and reporting of the routing tables. With statistics collected from the available clients, the master is able to predict the performance of varying load configurations (including configurations employing additional or fewer hosts as directed by the application programmer). For varying configurations of hosts elected for use, the master will produce a candidate load balancing strategy and estimate the resulting latency. Upon finding a load balancing strategy that meets the applications requirements, the master's routing table can be updated accordingly and the routing table can then be forwarded to all of the client hosts to take effect.

4.4 Workload allocations

As with any distributed system that performs data parallel processing, it is necessary to define how a workload should be parallelized. This section discusses how partitions are formed, allocated, and describes how data are associated with partitions in the case of both Map and Reduce operations.

4.4.1 Partitioning the hash space

The Google[20] and Hadoop[2] MapReduce implementations have all of their input data already pre-partitioned. That is, the data is read from a distributed file system (the Google File System (GFS)[21] and the Hadoop Distributed File System (HDFS)[1], respectively) such that the input data is already distributed onto a number of hosts. Hosts that posses this data can launch a MapReduce process to begin consuming that data. In this sense, effort is taken to try to launch MapReduce jobs where the data "lives" to preserve locality.

The data is not, however, pre-partitioned for the Reduce phase – instead each partition is likely scattered over many hosts. As we know, Reduce operations aggregate data that share a key. Therefore, all data with a like key must be grouped together for this aggregation to take place. When hosts facilitating Map operations produce key/value pairs for consumption by a Reduce operation, those hosts must ensure that all data with identical keys are deposited at the same host for aggregation. MapReduce, however, deals with "black-box" style operations where the code of Map and Reduce operations are not known to the underlying framework. Therefore, it is impossible to generally predict with any certainty what kinds of keys (and therefore partitions) will be produced by a Map operation. MapReduce resolves this issue by taking the hash of the key to create a unique identifier that lives in the hash-space. It is this hash space that can be partitioned amongst participating hosts to divide the workload of a Reduce operation.

C-MR uses this same strategy – hashing keys into identifiers, and then partitioning the hash-space amongst participating distributed hosts. However, C-MR must also employ a strategy to partition and distribute data destined for Map operations. At some level, we follow a very similar strategy but one that does not involve hashing. For Map data, we again partition a "space". For the purpose of partitioning it does not really matter what space, so we simply partition the same range used for Reduce data. That space is divided up into partitions which are assigned to the participating distributed hosts. In the Reduce case, a partition corresponds to a very specific set of data (the data whose keys hash into that partition's region of the hash-space). In the Map case, however, the partition simply represents the proportion (from the whole hash-space) of data which a host should process. Therefore, if a host has ultimately been allocated 15% of the hash-space, then the host would be responsible for processing 15% of that data.

4.4.2 Partition granularity

Before partitions can be allocated to the distributed hosts which facilitate the data-parallel computations, it is necessary to first determine how to partition the hash-space. There are multiple relevant strategies that we explored, each with their own strengths and weaknesses:

- **Elastic Partitions**: One eligible strategy focuses on what we've termed "elastic partitions" which may grow and shrink in size. Each host would only be allocated a single elastic partition but that partition would be resizable in order to accommodate the proportion of the workload that a host is capable of processing. Therefore, high performance hosts may have larger elastic partitions than the low performing hosts.

  Implementing such a partitioning strategy would involve the placement of a single index into the hash space for each host. If we think of the hash-space as laid out linearly, then the elastic partitions for each host would be represented by the space between their index into the hash-space and the next index to the left (or the end of the hash space if no others are found to the left). In this way, the index of the final host would be the right boundary of the linear hash space. An example of this can be seen in Figure 4.5.

  Elastic partitions provide ultimately fine-grained partitioning such that each host's allocated workload can match, as close as possible, its processing capacity. To transfer load between elastic partitions we can simply decrease the width of a donor partition, increase the width of the receiver partition, and correspondingly offset the host indexes in-between those partitions.

- **Many Partitions**: An alternative would be to use a larger number of fixed sized partitions. The goal would then be to distribute those partitions to the hosts in amounts that are proportional to their respective processing capacities. The resulting implementation would require a larger routing table for hosts as there are more possible partitions for a key to hash into when determining its destination host. While it is possible for the many partition strategy to support perfectly fine-grained load balancing (as the elastic partition strategy can) doing so would be unwieldy as this would result in a partition for every single possible value in the hash-space. By supporting partitions of any fixed size, this partitioning strategy supports variable granularity. It is common, however, to simply allow the number of available partitions to be defined as a multiple of the total number of participating hosts.

  Transferring load from one host to another, using this strategy, would simply involve the transfer in ownership of one or many partitions – this corresponds to updating the corresponding host ID for the transfered partition in the hash space. Therefore, the hash-space acts as a routing table which provides an entry per partition and identifies the host which will be allocated that partition. An example of this partitioning strategy is depicted in Figure 4.6.

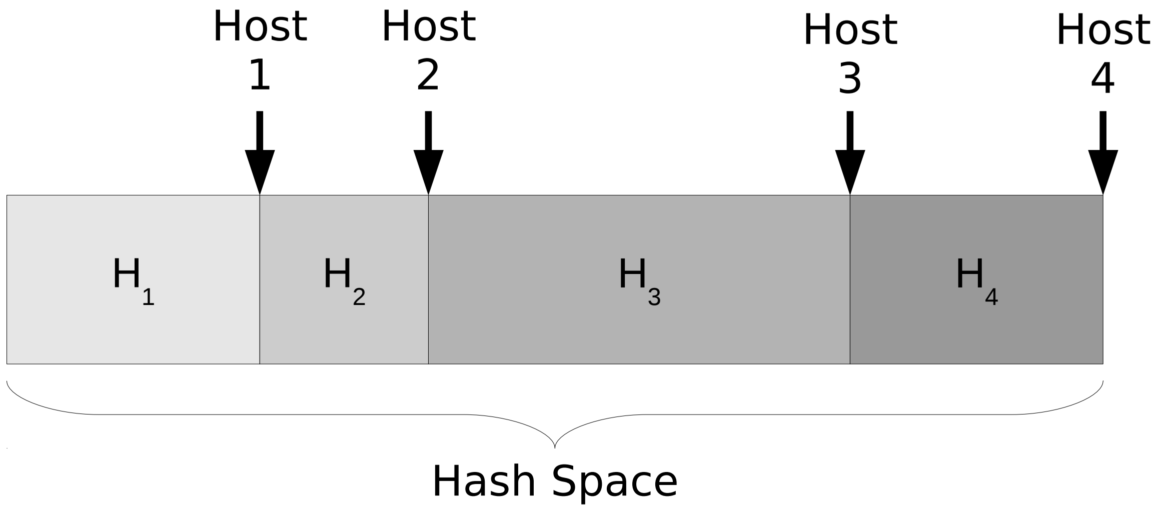


Figure 4.5: Elastic partitioning provides a single adjustable partition per host.

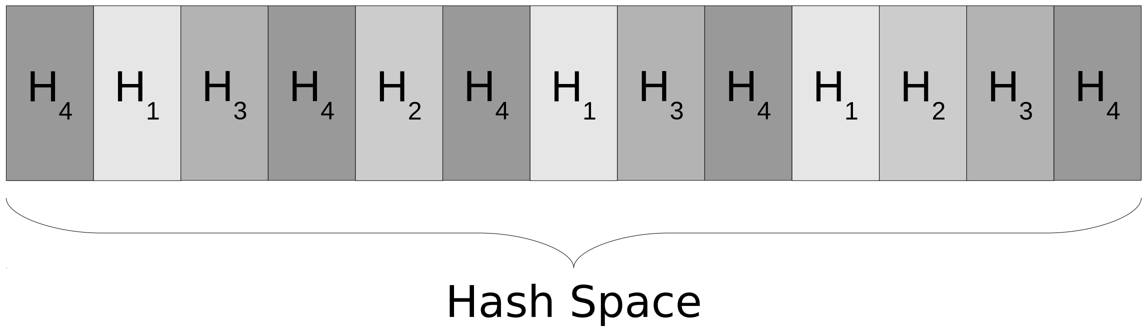

H_4 H_1 H_3 H_4 H_2 H_4 H_1 H_3 H_4 H_1 H_2 H_3 H_4
Hash Space

Figure 4.6: Many partitions enable the transfer of fixed-sized partitions between hosts.

When comparing these two strategies, we can find benefits to both. The elastic strategy is elegant and supports perfectly fine-grained workload partitioning. On the other hand, using many fixed-sized partitions is intuitive, trivially supports workload transactions, and provides fine-grained partitioning (but can not reasonably support perfectly fine-grained partitioning). Picking a strategy to use becomes much more straightforward when we consider the requirements of moving state between hosts. When Reduce operators consume temporal windows which are not tumbling – meaning that they have overlap – then a transfer of workloads would be required between the hosts.

Given such workload transition there are two possibilities to enable this state migration:

1. The state at the donor host can be instantly and entirely moved to the receiver host such that the receiver host can compute the very next window. The cost of this is the movement of all non-expired state and is initiated immediately once the change in ownerships is confirmed.

2. The upstream hosts, who deliver future data for this partition, would send the data to both hosts (donor and receiver) allowing the donor to continue to process the subsequent windows until the receiver's state has caught up fully such that it can commence processing the subsequent windows. The cost here is the replication of data over the network and the penalty that the load migration does not actually occur until the receiver host has fully populated its window for the relevant partition.

In both cases the communication cost is the same – the same amount of data is being transfered (either upfront or staggered). It is easy to see, then, that as a goal it is important to minimize state transfers while doing load balancing. The elastic partitioning strategy, unfortunately, requires quite a bit of unnecessary state migration when transferring load between two hosts. In fact, if one host increases the size of its elastic partition while another host decreases the size of its partition, then all hosts with partitions in between those two will be required to offset their index into the hash-space which will result in those portions of partitions being transfered to adjacent hosts. In other words, hosts which may keep their partition size perfectly static may have to participate in expensive state migrations simply due to the fact that their partitions moved along the hash-space.

Another reason to shy away from the use of the elastic partitioning strategy is because it is most effective when there is no data skew. If it is deemed that a host should receive a slightly larger partition size, then that host may increase its elastic partition to cover a portion of the hash-space that is extremely populated with data (more-so than other portions of the hash-space). Therefore, it is not safe to reason that the hash-space has data uniformly distributed across it and the elastic partition model is unable to account for such skew.

Using many fixed-sized partitions simplifies all of these problems. Regarding state migration, a migration in a workload is simply performed as the change in ownership of a partition. Therefore, only the parties taking place in the transfer will have a modified workload as a result of a partition exchange. Also, statistics can be observed regarding the data density of each of the partitions. By keeping track of the number of data observed, over time, within each of the partitions, then it is possible to estimate the likelihood of future data densities for those partitions. Therefore we can be aware of the data density of partitions we choose to move between hosts and even choose partitions of densities more appropriate for the desired adjustment in processing obligations for the hosts.

Being aware of such data densities allows us to be slightly more fine-grained in how we distribute work between hosts.

For these reasons, we implemented the many partitions strategy in the distributed implementation of C-MR. In its current implementation we simply assign the starting number of partitions to be a multiple of the number of hosts used upon startup of the stream application. Future work should analyze how this partitioning granularity should change with respect to the elastic addition or removal of distributed hosts. It is possible that changes in data skew within the stream can also cause for need to modify partition granularity to provide a finer-grained distribution of data density. However, changing partition granularity on-the-fly can possibly result in many immediate state migrations if the subsequent partition sizes are not a common factor of the previous partition sizes.

4.5 Distributed punctuation implementation

Section 4.3.2 described how the distributed processing model can maintain stream order, amidst the hierarchical application of the C-MR model, by hierarchically applying a punctuation management strategy. This section explains the implementation details that accomplish this and how it fits within the distributed C-MR architecture.

4.5.1 Inter-host punctuation management

Data-parallel stream processing will always pose a challenge to maintaining stream order as the data produced to a merged output stream may not maintain the order of the input stream. Hosts can process data at varying rates and therefore a data may be accelerated or slowed down such that it advances beyond or falls behind the rest of the data it is meant to be processed within. While not all window-based operations require that order to be maintained within windows, it is still necessary to ensure that all data relevant to a window has arrived before the window can be materialized and then processed.

The distributed C-MR processing model performs exactly this type of stream partitioning. Enacting data parallelism results in a series of parallel streams, each being consumed by a differing host, that each maintain their sorted order (prior to parallel execution within the hosts). By replicating window bounding punctuations from the sorted stream to the parallel streams, each of these parallel streams will maintain the window-bounding property of the punctuation. By the time all of the replicated punctuations have arrived at the merged output stream, the same window-bounding property will be satisfied on that stream. The challenge faced by the distributed implementation of

C-MR, however, is that there is no merged output stream until the results are produced to the user – the stream remains partitioned at all intermediate points within the workflow. This is addressed by replicating punctuations which maintain the window-bounding property to all of the parallel output streams as is depicted in Figure 4.2 from section 4.3.2.

Replication

Confirming that the window-bounding property indicated by replicated punctuations has been satisfied will occur in one of two ways based on if: 1) an operator's input stream is consumed directly from the application input via the master; or 2) if the operator's input stream is consumed by merging parallel output streams from separate hosts from an upstream operator. We discuss both of these cases below:

1. As is the case in the single-host implementation of C-MR, we assume that the application inputs are themselves sorted. As the master consume these application input streams it will insert the initial punctuations (relative to the downstream Reduce operator's slide intervals) to denote that the window-bounding property is satisfied. At this point, the stream is sorted and properly punctuated and must be parallelized to the distributed hosts that will process the data. The master sends the input data on to the hosts via the Output Manager and instead of simply sending on a punctuation via the Output Manager, the Master instead sends each host n replicated punctuations (one replica for each active host) for a total of nxn punctuations. In this way, the master mimics the behavior of the collection of hosts which would have otherwise done the same had this workflow stream not been derived directly from the application input. Ultimately, each host will receive from the master n punctuations – the receipt of which will satisfy the window-bounding property.

2. In the case where punctuations are not being received from the master via the application's inputs, punctuations are received (and hence replicated) from the collection of hosts themselves. Each of these hosts will manage one of the parallel streams that a "whole" stream is divided into. Therefore, these hosts will consume data, process data, and produce data from their Output Managers which may be directed to one of any of the active hosts. Likewise, these hosts will have window-bounding punctuations reach their Output Managers to be forwarded on to other hosts. At this point (when a window-bounding punctuation reaches a host's Output Manager) the punctuation will be replicated n times with one replica being sent to each host. Once a host receives a punctuation for the downstream operator from all n hosts, then the relevant window can be materialized.

Collection

A receiving host will know that its merged (yet still parallel) stream will satisfy the window-bounding property when one replica punctuation from every active node has been received at its input. The Input Manager receives these replicated punctuations and buffers them within the Workflow Buffer. Once the Input Manager has received and placed a replica from each host at the Workflow Buffer – signifying that all hosts have produced the contents of the relevant window – the corresponding data within the Workflow Buffer can be materialized and made available to be processed by the host's interior processors (CPUs/GPUs). At this point the host's interior stream is properly sorted so that the host may produce punctuations for additional downstream Reduce operators if applicable.

Ultimately, the replicas are generated at a host's Output Manager when the host's local/interior stream has been sorted (by intra-host punctuation management, as will be discussed in section 4.5.2) and punctuations are collected by the Input Manager within the Workflow Buffers where they can be materialized upon receipt of all punctuations.

4.5.2 Intra-host punctuation management

The previous section just discussed how a host can confirm that its merged input streams can satisfy window-bounding properties through punctuation management. Immediately after this property has been satisfied, the stream may be partitioned locally, yet again, for parallel consumption by multiple local processors (CPUs/GPUs). To ensure the window-bounding properties of the merged output of these local processors, it is necessary to yet again punctuate the stream – this time within the host.

Replication

Punctuation replication within the host will occur immediately before the stream is partitioned for parallel consumption. Therefore, right after the inter-host punctuation collection has occurred to sort the stream and materialize punctuations, we will immediately apply intra-host punctuation replication upon observing a punctuation from the workflow buffers. Similarly to in the single-host implementation (depicted in Figure 2.3 from Section 2.2.2), when a processor consumes a punctuation from the workflow buffer, it will replicate and push a copy to each of the other processors. Since the processors have serialized access to any operator's input buffer, we can guarantee that the replicas will properly bound each of the processor's parallel streams with respect to the given window.

Collection

As the local processors produce their results to the Output Manager, they also produce any replicated punctuations that have been issued to the Output Manager. It is at the Output Manager that these replicas are collected. Once all replicas have arrived at the output manager, the window-bounding property has been satisfied. At this point, the inter-host punctuation management policy will kick in and produce a corresponding punctuation for each of the other hosts to transition from intra-host to inter-host punctuation management.

4.6 Network and communication management

This section discusses the networking considerations that were made for the distributed implementation of C-MR to provide many-to-many communication, facilitate efficient use of network bandwidth, and how specific networking modules were implemented within the distributed C-MR architecture.

4.6.1 Networking model

They very nature of MapReduce requires many-to-many communication. Mappers are initially launched on hosts which contain relevant data situated within GFS or HDFS, process that work, and then they forward their results to potentially all of many Reducers. MapReduce uses a strict two-phase process in which the Mappers finish processing their data fully before the Reducers are launched. As Mappers process their data, the results are spilled back to disk. Once the Mappers have finished, an optional Combine step can be executed to further consolidate results. Then a Shuffle step takes place in which the Mapper's output data is sorted by key and then distributed to the Reducers – one after the other.

In this sense, implementations of MapReduce enable a Mapper to send a large batch of data to each Reducer in a single shot. The Mapper can establish a TCP connection, send the data, close the TCP connection, and then move on to the next Reducer. C-MR requires a very different networking model.

In the case of the distributed implementation of C-MR, there are Map operations and Reduce operations across a workflow of MapReduce jobs that can all be running simultaneously. At any given moment, a host can process data belonging to a Map operation which will produce a key. That key will be hashed into the hash-space to determine which partition it will belong to and the routing table in the Output Manager will determine which host has ownership of that partition. The result will then be sent on to the appropriate host. Given that Reduce keys produced by Map operations are not predictable, and that it is vitally important to maintain low latencies, we do not have a means to know in advance where data are headed to establish a connection to a TCP socket. Because a host may, at any time, be required to communicate data to any other host, it is necessary to have extremely quick access to an established TCP connection to the destination host. This is clearly obvious in the case of inter-host punctuation replication in which replicated punctuations are immediately fired off to every active host.

Therefore, we employ a many-to-many networking model with TCP connections established between hosts at startup and maintained throughout the application's lifetime. While this many-to-many socket relationship has the potential to limit the scalability of data processing platforms, stream processing applications generally require far fewer hosts than large-scale batch processing applications employ.

4.6.2 Host communication modules

The following sections describe the communication modules implemented in the distributed C-MR framework and detail the types of data and control messages communicated and the manner in which connections are maintained.

Input Manager

The Input Manager actually operates as a pair of threads which are launched by the host thread at startup. The first thread is responsible for establishing a TCP socket to listen for and receive data, punctuations, and any control messages that may arrive from hosts or the master. Data and punctuations are placed into an input buffer. The second thread consumes from this input buffer and inserts the contents into the Workflow Buffers. Both threads are expanded upon below:

1. **TCP Listening Thread**: The listening thread establishes and listens on a single TCP socket that all other hosts (including the master) will write to. This socket receives data, punctuations from the hosts, and routing table updates from the master. The data and punctuations are placed into an input queue which are then later consumed by the input queue thread. This allows the TCP Listening Thread to be free to handle incoming data from other hosts while leaving the efforts of integrating data into the Workflow Buffers to the other thread.

   In addition to receiving data and punctuations, the listening thread also receives control messages which may either be statistics updates (received only on the host running the master) or routing table updates. If statistics updates are received by the master, then the statistics are inserted into the load balancing module. Otherwise, if routing tables are received, they are appended to the routing table used by the Output Manager for future use.

2. **Input Queue Thread**: A secondary thread is used to take data received from the TCP listener and insert it into the workflow because it is possible that inserting punctuations into the workflow may trigger window materialization which need not detract from the duties of the TCP listening thread. Materializing windows involves moving data into the queues that processors consume from and also may require the creation of additional punctuations if there are subsequent downstream Reduce operations. In addition to this, the normal work of hashing key values and inserting data into appropriate buffers need not hinder the progress of the TCP listening thread and those other hosts waiting on its immediate responsiveness.

Output Manager

The Output Manager is facilitated by its own thread which is launched at startup by the host thread. It then establishes TCP socket connections between the TCP listeners facilitated by the Input Managers on all of the other hosts.

The Output Manager is then free to handle data produced by the host's local processors. As the processors consume data, they produce the results to a buffer that is read by the Output Manager. The Output Manager reads from this buffer, determines how to route the data (by consulting the routing table), and then sends the data to the appropriate host. That data may be sent via TCP connection to a remote host or, if the data is determined to stay locally, the data will be re-inserted into the workflow manager.

In addition to receiving data produced by the processors, the Output Manager will also receive punctuations from the processors via the same buffer. The Output Manager is responsible for counting the received punctuations and expects to receive replicas from all processors for each window that those punctuations bound. Therefore, it is the Output Manager which, on receiving all local punctuation replicas, produces new replica punctuations for inter-host punctuation management to send to the other hosts. In this sense, it performs the collection phase for intra-host punctuation management and the replication phase for inter-host punctuation management.

Beyond these responsibilities, the Output Manager also maintains an interface for communicating statistics to the master and for sending new routing tables from the master to all hosts.

4.6.3 Network conservation

C-MR takes steps to minimize the amount of data communicated over the network. The steps taken in the subsequent sections serve to reduce latencies by avoiding unnecessary network transfers or by batching communication together to initiate fewer transfers.

Routing Map data

We've previously established that data destined for Reduce tasks must be collected at a very specific location so that it can be aggregated with all other data of a similar key while data destined for Map operations can be processed literally anywhere. Map data have this flexibility because the straightforward transformations and filtering performed by Map operations can occur independently on each data item in the stream.

When it comes to produce data that is destined for a Map operation, we must determine if or where it should be sent. Suppose that a processor has finished aggregating data for a Reduce operation, and its results are destined for a downstream Map operator. The processor will take the resulting data and put it into the output buffer to be read by the Output Manager. The Output Manager then takes this data and consults the routing table – not to determine which partition the data hashes to, but to determine which proportion of the overall map workload the local host is required to process.

By observing statistics regarding the quantity of data produced by an operator and by also being aware of how much data a host is meant to process, regarding the downstream operator, we can develop an understanding of: 1) how much of the resulting data to process locally; 2) how much of that data we should forward to other hosts who will not produce enough of their own data to process; and, 3) how much data a host can expect to receive from other hosts if the local host will not produce enough of the data to fulfill its processing obligations.

In other words, if through the act of load balancing and the exchange of partitions we find that a host is meant to process 30% of a Map operator's workload and we find that this host will generate 40% of the Map operator's workload from the upstream operator, then we can retain a large portion of the workload produced by this host (75%) and transmit the remaining portion (25%) to those that are expected to under-produce from their upstream operators. By observing the statistics regarding the amount of data produced by operators and being aware of each host's relative share of that processing work, we can estimate the amount of data each host will produce. The routing tables tell us the proportion of data that each host will consume for the immediately downstream operation. We then note the hosts which will under-produce data for their downstream workloads. For those hosts that over-produce data for their downstream workloads, they can take the excess of their workloads and divide it proportionally amongst the hosts that are known to under-produce relative to the amounts by which those hosts have each under-produced.

In the distributed C-MR implementation, we interleave the local processing and shipment of excess data to under-producing hosts in such a way that does not result in only sending excess data once all of the local data has been processed. This allows processing to occur during the data communication (both sending and receiving) on behalf of both the over-producing and under-producing hosts.

Data batching

Another strategy that can be used to decrease the number of network transfers is data batching. For instance, the Output Manager can send a number of data items to a remote host in one transfer instead of sending each data item individually.

There is a significant trade-off to be aware of when batching data for network communication. When only sending a single data at a time, we are assuring that the first data item sent will have a low latency. However, subsequent data items may have increasingly larger latencies (providing that they were all ready for transfer within a relatively small amount of time). Batching data together for transfer decreases the total number of transfers and therefore decreases the total amount of network traffic. Depending on the inter-arrival rate of the data, a batching strategy may reduce the average latency of transferring a large amount of data between two points. However, this is very much dependent on the characteristics of the application and data arrival rates. If data are significantly spaced apart, then batching will only introduce delays.

C-MR, therefore, supports the use of batching for both network transfers as well as for batch processing at the host's local processors (in an attempt to reduce overhead while receiving cache coherency benefits) but does not try to actively learn whether batching is suitable for the application at hand or what amount of batching is most appropriate. This would be an excellent avenue for future work that would have wide ranging implications for a number of stream and batch processing frameworks.

4.7 Load balancing

The distributed implementation of C-MR does not share the luxury of the single-host implementation which allows the local processors (CPUs/GPUs) to consume data from the shared workflow buffers in a pull-based manner. In this way, load was not pre-partitioned to each of the processors. Instead load was perpetually shared between all processors. Load balancing was implicit; a processor would never be idle if there was data available to able to be consume and no single processor was ever more burdened than any other.

Performing such pull-based data acquisition from a shared workflow buffer is not a feasible option for a distributed system. The single-host implementation could get away with this because the communication cost of a processor fetching data from the shared workflow buffer was essentially free – the host would just acquire a pointer to the data's location in the shared memory. In a distributed scenario, such data acquisition would result in a costly network transmission each time a host requests data. While it's true that data must be communicated to these hosts anyway, as data must be distributed in order to apply parallel processing, the additional cost comes from a host engaging in the protocol to request data, wait for a response, and wait for the data to arrive before processing could begin.

In an ideal scenario, we would already know exactly how much data each host should consume to adequately balance an application's workload. Then, it would be possible to just transmit this portion of data to a host without it ever having to ask. This would allow the host's local processors to spend their time doing useful processing work without large delays between data. Therefore C-MR aims to facilitate this case – predicting the appropriate portion of workload to allocate each host. Thus, instead of pull-based data acquisition, a pre-determined data allocation strategy is employed.

4.7.1 Balancing objective

In most stream and batch processing systems, the act of load balancing is performed on homogeneous computers. These computers are usually presumed to be equal and what data is available to process is evenly distributed between them. In this sense the "load" or amount of work to be done is what is being balanced between the hosts. When using heterogeneous hosts, however, we do not have the assurance that each host will be able to handle equal load as gracefully as the others.

The general goal of batch processing frameworks is to finish processing a large amount of data as quickly as possible. In this sense, the goal is to minimize the time between starting the batch load and finishing the batch load. A "fast" computer might be allocated a larger workload than a "slow" computer as it will be able spend less computing time on that workload than the slow computer. The use of heterogeneous computers, therefore, requires not strictly balancing data, but allocating data to hosts in order to minimize the start-to-finish processing time.

The situation is quite similar in stream processing frameworks when using heterogeneous computers. The goal in stream processing frameworks is often to minimize the average latency of results. Supposing we must allocate a new piece of data to one of any available hosts, the question that we should ask is: "given the workloads already present at each computer, which host can be allocated this data such that the average data latency across all hosts is increased the least?" The objective is really quite similar to that in the batch processing scenario. Given a large workload distributed onto heterogeneous hosts, each host will actively produce data at a specific rate relative to the volume of data it has to process and its processing potential. Increasing or decreasing the workload of each host may increase or decrease the latency of the results it produces. Therefore, when possible, our goal is to transfer load from the poorer performing computers (those with higher tuple latencies) to those that are currently performing better (with lower tuple latencies) such that the resulting transfer brings down the overall average latency across the pair of hosts.

4.7.2 Latency prediction

The workload balancing goal we focus on with C-MR is to reduce the end-to-end latency of stream processing applications. Knowing how to move load between computers requires being aware of their individual host performances with respect to the partitions they are processing. As C-MR uses a centralized load balancing strategy, it is necessary to collect statistics such as these at a centralized location in order to make judgements regarding the exchange of load between hosts.

Further, we use these latency statistics to build a latency prediction model to estimate the potential latency outcomes of candidate load balancing strategies.

4.7.3 Latency prediction models

When determining how to estimate latency, we investigated three possible strategies.

1. **Curve exploration**: This method requires finding high-dimensional curves which denote operator latencies given workload allocations for each of the available hosts. For each workload configuration employed, actual latency observations would be noted as a sampled point within this curve. As workload balancing changes are enacted, additional configurations will be achieved and therefore new sampled points will be identified in the high-dimensional space. These sampled points will be used to estimate the slope of the curve at various points in space in order to attempt to generate new sample points in locations where we hope to find a minima.

2. **Simulation-based latency estimation**: End-to-end workflow latencies can also be derived through discrete event simulation. After workload allocation strategies have been defined, a simulation could mimic input stream rates to produce simulated data to simulated hosts. The simulator would implement the very same operator scheduler used on the hosts and simulate the processing of its data in the corresponding order. To do this, the simulator would need to maintain statistics on the stream arrival rates, processing speeds, and partition characteristics.

3. **Queue Length estimation**: Another strategy we investigated was to determine the average latency of data through each workflow operator and then calculate an end-to-end latency through the composition of the latencies of each of the workflow operators. To estimate the average latency of data through an operator we identify the components that contribute to

latency. Those are: 1) network transmission time; 2) time spent waiting in a host's queue to be processed; and 3) time spent being processed. We therefore determine how these latency components are effected by increasing or decreasing the load of a workflow operator while also considering the impact on queuing wait-times perceived due to other operator workloads at each host. With a new configuration of workloads for each of the operators, this strategy aims to estimate new queueing times in order to predict average operator latencies.

After careful consideration of each of these three strategies, we quickly realized that the curve exploration method had many significant hurdles. The first problem with such a strategy is that exploring and determining the features of such a high-dimensional curve would be very difficult to do without a large number of sampling points. Investigating new sampling points would, in fact, take quite a bit of time since it is required that the workload configuration corresponding to each sampled point be executed in the actual processing environment to acquire the latency observation. Secondly, the shape of the curve is not actually static and as the "shape" can change due to any of the following events: 1) modifications to scheduling policies; 2) changes to host processing potentials due to external workloads; and 3) changes in the skew/volume of any of the stream partitions. Such a strategy is simply not tractable and not suitable for a dynamic stream processing environment.

The simulation-based approach obviously requires a bit of computational effort to perform discrete event simulation for a fully filled pipeline of streaming data through potentially large workflows of operators. In fact, a discrete event simulator incurs all of the busy-work and bookkeeping work that the real-time system incurs (including punctuation replication and collection). Further, the simulation would only be executed once all workload allocation strategies have been assigned through the load balancer. Therefore, the simulator would not provide end-to-end latency insight regarding the distribution of individual workload partitions (without resulting in the performance penalty of a simulation invocation for each individual transfer). Instead, the simulator is only intended only for evaluating workload partitioning candidates.

The queueing-time estimation strategy, on the other hand, can provide insight regarding the redistribution of even a single partition. This strategy creates a model of the average latency associated with each host for each operator and can easily reflect changes as a host's workload increases or decreases. The model is not expensive to maintain – only requiring simple extrapolation of queueing times given workload re-allocations. As the computational complexity of this model is the lesser of the available options and the required statistical information to drive the model is easily produced by the hosts, we decided to use the strategy as we moved forward in creating latency predictions.

4.7.4 Building a wait-queue model

In this section we further expand on the implementation of the queue-length estimation model. To do so, we must first discuss what components contribute the latency of a data item through an operator as is facilitated by a host. The following steps occur:

1. A data item is transmitted to a host
2. The data item waits to be processed
3. The data is processed by a computing node at the host

The transmission time of data to a host can involve the transfer of data over the network or simply the re-insertion of data into a host's own workflow buffer. In either case the latency penalty for each type of transmission is easily determined and rather constant. Likewise, the processing time for data is also rather constant. Hosts can report variations in processing times (if hosts have background processes stealing CPU cycles or if there is skew in data complexity) but there result of workload balancing will not specifically impact the time it takes to process a data.

What *is* impacted by workload balancing and the redistribution of partitions is the wait time perceived by the data waiting to be processed. Also, these wait times can be changed instantaneously through the sudden switch of scheduling policies or they can be changed gradually through the use of a progressive scheduling policies. In the long run, the expectation of host transmission times and host processing times are relatively fixed whereas there can be great variability in the amount of time data wait to be processed.

Factors behind wait-time variability

Due to the use of the generic-node processing model, enabling computing resources on a host to be used to process data from any workflow operator which has data available to be processed, a data item may perceive wait times from data of any other type as they are processed before it. More generally, as a data item waits to be processed, a number of other data items corresponding to different (or even the same) operators may be processed first. In the case where a host has a single processor or core, the summation of these times will correspond to the time that a data item waited before being processed. This is the notion that we use to build our model.

We capture the average amount of time that a data item of each operator type will spend waiting on data from other operators before being processed. For example, we might observe that, given operators of type A, B, and C, that data from operator A would, on average, wait for 3 data items of type A, 1 data item of type B, and 0.5 items of type C before having its turn to be processed.

Since the processing times of each type of data is being observed at each host, it is straightforward to multiple the processing time of each operator type by the corresponding number of data waited on by for each type to achieve the average time spent waiting.

In C-MR we implement an internal counter system to detect the average number of data types that each data item wait on before being processed. Keeping these wait-time averages, for each operator type, allows us to maintain a model the predicts the latency of data item of any operator type through a host.

Let's consider how this model represents variability in the processing environment. If a host's workload for operator B increased, then we would see no change in the communication or processing times corresponding to operators A or C on that host. However, we would see the potential for increases in the amount of time data for operators A and C waited to be processed. Lets drill down on data for operator A. Let the total wait time perceived by operator A be expressed as W_A. That wait time is composed from waits perceived by data being processed that belongs to various other operators. Therefore, we can express W_A as:

$$W_A = W_{A,A} + W_{A,B} + W_{A,C}$$

such that $W_{A,i}$ represents the average amount of time that a data item destined for operator A will wait on the processing of data destined for operator i. So, if this particular host's workload for operator B increased, then we would only see a potential increase in the wait time data of type A perceive from data of type B: $(W_{A,B})$. This is because $W_{A,B}$ is specifically related to the processing effort required of the host for operator B. Therefore, as the workload of B increases, so does the potential for $W_{i,B}$ to increase for any operator i. We can be sure, however, that no other wait times of the type $(W_{i,j} | j \neq B)$ will increase as their waits are derived specifically from the processing of workload types not affiliated with operator B and are therefore not subject to change given the increase to only the workload of operator B.

Likewise, we can similarly expect the decrease of a workload of operator B to have a corresponding impact such that only wait times of type $W_{i,B}$ are affected. Given a change to the workload of an operator at a specific host, we perform a linear extrapolation of the wait times of type $W_{i,B}$ such that the wait time is scaled proportionally to the change in workload. Thus, as the workload of operator i changes for a given host, we perform a scaling of the wait times of types $(\forall j, W_{j,i})$ relative to the change in the workload allocation. So, if the workload of operator B decreased by 20%, then we would modify the model in the following way:

$$\forall i, W'_{i,B} = W_{i,B} \cdot \tfrac{1.0-0.2}{1.0}$$

The linear extrapolation we apply to wait times, given changing workloads, is used with the assumption that the arrival distribution of the newly allocated/de-allocated workload follows an arrival distribution identical to that of the prior workload. In reality, it is possible that a large amount of newly allocated data for operator B could possibly arrive specifically during a time when there is no other data to be processed (some regular period of down-time). In such a scenario, there would be no increase to the wait times of data from other operators as there would be no additional contention. Likewise, it is possible to see the removal of a workload that never contended with any other and therefore never contributed to wait times of other operators. In both of these cases, a linear extrapolation of wait times which is proportional to the change in workload would not be appropriate. However, predicting the arrival distribution of workload partitions and, likewise, detecting the arrival skew of certain types of data within the stream is a challenging task which is not addressed in this thesis. Our assumption, that arrival distributions are similar prior to and after a change in workload is based on the following notion. At any time a data item of interest X may be processed, and before it is processed other data of various types will be processed. In other words, in hindsight (after a data is processed), we can assume the average data item X to initially be situated at the end of a queue of other data that were processed first:

```
X A B B C A C C A A B B A A B A C
```

That is to say, the average data item X from operator i might be proceeded by 7 data of type A, 5 data of type B, and 4 data of type C. Here, we hold the assumption that if the workload of operator $j \in A, B, C$ changes, then the proportion of data of type j proceeding X in the queue will increase or decrease proportionally. To make any other type of consideration would assume a knowledge of the data arrival distribution of all intermediate streams and the skew of arrivals between partitions of (like-key) data within the stream. Without this information, we can make no other assumption than to assume consistency with recent observations.

Our observations do capture some of this data distribution and skew in arrival patterns, but we do not use this information to predict future variations. Our assumption simply expects arrival distributions to be similar to previous observations.

4.7.5 Statistics collection

For the master host to predict the average latency of data through operators for each host and to predict the average end-to-end latency of data through the workflow, it is necessary to maintain information about the performance of the hosts, the distribution of data in the workload partitions, and the scheduling habits of the hosts. This information must be observed at each host, reported to the master, and then collected and maintained by the master for use during load balancing.

Statistics observed

We now further discuss the types of statistics that *each* host observes and reports for *each* operator in the workflow.

1. **Average processing times**: The time it takes a host to process a data item of a specific workflow operator is a vital component to estimating the latency of data through that operator. Likewise, processing time is a vital in calculating latency. Therefore, each host observes and reports this statistic for each workflow operator so that the master can build and maintain a model to determine a host's processing performance for each of the workflow operators. This information will further help the master when determining how to move workload partitions between hosts.

   To calculate this statistic, a host simply observes the time it takes to process data and computes a moving average of these times to report. A moving average is used so that the average can reflect changes in the processing environment. If a host becomes burdened with an external workload then the observed processing times will increase. Likewise, if an external process stops stealing CPU cycles then the processing time may decrease. Observing and reporting a moving average allows the master to be aware of the real-time performance changes of each of the hosts to better distribute workloads across those hosts.

2. **Partition size**: As the fundamental unit of workload transfered between hosts is a partition, the load balancing process must be aware of the processing requirements of each of those partitions. When balancing Map workloads, the sum of all data observed in all partitions gives us a notion of total workload size for the operator. From this the load balancer can determine what portion of the total workload should be consumed by each host. However, for Reduce workloads, hosts must aggregate data belonging to the partitions they have been assigned. As hashing schemes are used to map such data to partitions, it will be the case that partitions will have differing numbers of data in them. Likewise, the nature of dynamic streams will also result in data skew over time such that each partitions size may change with the subsequent windows that will be processed.

   Hosts, therefore, report the number of data belonging to each partition they receive (or, in the case of Map operations, report the total amount of data received) so that the load balancing process will be aware of the data complexity contained within each partition as it balances load.

3. **Average wait time per operator**: For the purpose of allowing the master to produce a wait-queue model for each of the hosts, all hosts will report the average amount of time that data from the identified operator will wait on data from all operator types before being processed. The wait times for each individual data are collected and summed over the period of statistics collection. This results in the total amount of time that each data, for a particular operator, waits due to processing from data of all other types (include its own type) and for *each* of the local processors. Once the statistics collection time period ends, we divide this sum by the number of data observed for the relevant operator and by the number of local processors. Then, we use the sum of the wait times perceived from each of the operators to represent the average amount of time a data spends in the queue waiting to be processed.

   We modify this running sum which represents wait time when the following events occur:

   - **When a processor finishes processing a data**. We then take the time it took for the local processor to process the data, multiply it by the volume of the items waiting in the queue, and add it to the wait-time for that queue. The intuition here is that each of those data were required to wait while the local processor was doing this processing. There are, however, two edge cases which we address in the subsequent bulleted items.
   - **When a data is removed from the queue to be executed**. The above case doesn't consider data that wait while one processor processes but are then selected to be processed by another processor. This prevents the first, running, processor from applying its wait time for the newly scheduled data. To correct this, when a data is removed from the queue, we take the current timer values from each of the currently processing processors and add those values to the wait-times corresponding to each queue holding data. Thus, we add partial processing times to include as the wait time for a data.
   - **When a data is added to the queue**. The first bulled point notes that processing times are multiplied by the number of data in a queue to add to the wait time. This doesn't correctly reflect the wait time for data that were added to the queue after those processors began processing. To remedy this, when a data is added to a queue we **subtract** the timer values from each of the running processors from the appropriate wait sums. Thus, if a processor is $\frac{3}{4}$ finished processing its data, we subtract that amount of the processing time from the queue. Once that processor finishes, it will add a full portion of the processing time for the given data. Therefore, there will be a $\frac{1}{4}$ net addition of processing time to the wait-sum for the newly added data. If the recently added data is scheduled by another processor in the meantime, then we will add to the wait times as indicated in the second

bullet point and the resulting net addition will still equate to the amount of time the data spent waiting in the queue.

Instead of using timers, one can simply calculate the number of data processed while another data waits and multiply those times by the respective processing times to estimate wait time. Likewise, sampling can be used to detect these processing times. However, this strategy does not account for partial wait times (as outlined in the bullet points above) and is therefore less accurate.

Reporting statistics

Currently, the distributed implementation of C-MR employs punctuation-driven statistics reporting to communicate the previously mentioned statistics to the master host for load balancing.

Reduce operations are very much punctuation-oriented. Punctuations define when the data they are batching may be materialized into windows and processed, and they also denote delineations within the stream that offer opportunities to modify routing tables. That is, punctuations provide a synchronization mechanism that allows hosts to alter reduce-key routing tables for windows bounded by future punctuations. Likewise, they offer an opportunity to synchronize the reporting of statistics to the master host.

At present, the interval denoting the period between the start of statistics collection and the time of statistics reporting is defined as a common multiple of all workflow window sizes. During this interval operators will collect statistics. Once the interval has elapsed (as confirmed by a punctuation which is a multiple of the interval being received at a host) then the host will report the statistics of its operators to the master.

Upon the master's receipt of all operator statistics, from each host, load balancing can begin.

4.7.6 Workload re-distribution

To do load balancing on a set of hosts, the master first needs updated statistics reported from all operating hosts; these hosts must report information relating to each of the workflow operators. This information provides the master host with knowledge of the performance each host is currently able to deliver with respect to each workflow operator.

The inputs to this load balancing process are: 1) the reported statistics; 2) the working set of hosts that will facilitate workflow processing; and 3) the previously used routing tables which define the current workload allocations. With this information, the master host then begins balancing load on each operator in the workflow individually.

Balancing operator load

The first step in balancing the load of an operator is taking account of the working set of hosts to be used. As mentioned previously, the set of hosts to be used is passed to the load balancer as input. It is possible for that set of hosts to be different than the set of hosts used previously. The new working set may include the addition of a new host or might not include a host that was previously present.

For those hosts that are new to the working set of hosts, we will not have up-to-date statistics as they have not been processing data. When faced with the addition of a host, we take the average of statistics from other hosts that did report statistics. This allows the estimation of processing abilities of the new host to be similar to those hosts that are already active; for a homogeneous set of hosts, this is to be expected. Given a set of heterogeneous hosts of specific types (such as those types that can be utilized on Amazon EC2[7]) we instead take the averages of statistics reported by hosts of the same type which are active in the system. For hosts of an unknown or never-before-seen type, we can simply assume the average of all host types for an initial estimation before statistics are later reported which will more accurately describe that new host type.

It is also possible that a host will be removed from the working set of hosts. In such a case, we would find that a host is defined in the previously used routing table and is no longer found in the set of hosts to be used. Upon identifying such hosts and their orphaned workloads, we mark those workload partitions as unallocated. Before the load balancing process officially begins, for each operator, we quickly and simply distribute those unallocated partitions to the available set of hosts in a round-robin manner.

With statistics in place for the active set of hosts and no unallocated workload partitions left to distribute, we then determine the average latency of data through each host with regard to the operator at hand. First, for each host, we iterate through each of the partitions it is currently allocated and take the sum of the counts of data within those partitions. The result is a total number of data processed by each contributing host for each workflow operator. We then compute the expected average latency of data through the operation, using operator workload quantities as well as wait time and processing time statistics, as depicted in equation 4.1.

$$\text{averageLatency}(h, i) = P_{h,i} + \sum_{j} (W_{h,i,j} \cdot D_{h,j}) \tag{4.1}$$

This function takes arguments h and i which describe the host ID and the operator ID, respectively. The master host uses this information in combination with gathered statistics (processing time P and wait time W) and routing table information (data quantity allocated $D_{h,i}$ to host h for operator i) to produce latency estimations to compare host workload allocations. The value $P_{h,i}$ denotes the observed amount of time that host h spends, on average, processing a single data item for operator i. The value $W_{h,i,j}$ denotes the expected amount of wait time that data from operator i will perceive per data of operator j on host h. Thus, once data from operator i are eligible to be processed, their average latency through host h, given workloads specified by D, will include the processing time of that data as well as the time spent waiting to be processed. The first term in the equation is straightforward and represents the processing time. The second term in the equation represents the summation of the wait times perceived at the hands of all other operator data.

Having applied this function for all hosts for a particular operator, the result is an average latency for each participating host. This is used as the basis of determining which hosts are better performing and which hosts are poorer performing with regard to the current workloads. Hosts are sorted with regard to this latency metric. The intuition we use for load balancing is that if there is a single data to be allocated onto the set of hosts, then it would most likely be processed the quickest by issuing it to the host with the lowest average data latency for that operator. Likewise, we assume that if we were to free up a data item to be allocated to the fastest host, it is intuitive that it would be most beneficial to remove that data item from the host reporting the highest latency. Therefore, it follows that we can create donor-receiver pairs of hosts (pairing high-performing hosts with low-performing hosts) to transfer and balance workload processing responsibilities.

We follow a strategy used in [14, 37] which pairs the best performing host with the worst performing host, the second best performing host with the second worst performing host, and so on. We then iterate through these pairs and consider transferring workload partition ownership from the donors (poorer performers) to the receivers (higher performers). Given the available partitions the donor has ownership over, we first consider sending the larger partitions – those the reported statistics have shown to include the largest amount of data. Given a candidate partition to transfer from the donor and receiver, we estimate the average latencies of the hosts that would result from the transfer using a modified form of equation 4.1. The modified form scales the value $D_{h,j}$ where $j = i$ to reflect an increased (in the case of the receiver) or decreased (in the case of the donor) workload on operator i. The modified equation is expressed in equation 4.2 in which Iverson Notation[23] is used to represent the conditional summation.

$$\text{expectedLatency}(h, i, f) = P_{h,i} + \sum_{j} (W_{h,i,j} \cdot D_{h,j})[j \neq i] + (W_{h,i,i} \cdot D_{h,i} \cdot f) \tag{4.2}$$

The expected latency function is similar to the previous function but allows for a third parameter

f to represent the degree by which the original workload in scaled. Thus, if the original workload for host h given operator i was a volume of $D_{h,i} = 90$ data items over the observed statistic period but the proposed increase of $d = 20$ would result in 110 data items, then the scaling factor would be the following:

$$f = \frac{D_{h,i} + d}{D_{h,i}} = \frac{110}{90} = 1.2\overline{2} \tag{4.3}$$

Likewise, if the host's operator workload decreased by $d = 20$ then the scale factor value would be:

$$f = \frac{D_{h,i} - d}{D_{h,i}} = \frac{70}{90} = 0.7\overline{7} \tag{4.4}$$

After determining the expected changes in the average latencies of the donor/receiver pair, we only allow the partition to change hands if the average data latency between the two hosts has decreased. That is, given a data volume, d, to transfer between a donor host and a receiving host, we check to see if the following statement is true:

$$\begin{aligned}
&\text{Let donor}L = \text{averageLatency}(\text{donor}, i)\\
&\text{Let recvr}L = \text{averageLatency}(\text{recvr}, i)\\
&\text{Let donor}L' = \text{expectedLatency}(\text{donor}, i, \tfrac{D_{\text{donor},i} - d}{D_{\text{donor},i}})\\
&\text{Let recvr}L' = \text{expectedLatency}(\text{recvr}, i, \tfrac{D_{\text{recvr},i} - d}{D_{\text{recvr},i}})
\end{aligned}$$

$$\text{donor}L \cdot D_{\text{donor},i} + \text{recvr}L \cdot D_{\text{recvr},i} > \text{donor}L' \cdot (D_{\text{donor},i} - d) + \text{recvr}L' \cdot (D_{\text{recvr},i} + d) \tag{4.5}$$

If the resulting expected latency is lower, then we commence a transfer in ownership of the partition. After transferring ownership, it is necessary to reflect the changes in the routing table which will also update D to modify the data counts of the hosts in preparation for future data migrations and latency estimations. The process can then be repeated for subsequent pairings of hosts.

Once all pairs of hosts finish swapping data, as they are able, we repeat the process for the remaining operators in the workflow. It is important to note that changes made by balancing one operator will be propagated through the statistics for the balancing of subsequent operators. That is to say, as workloads are reallocated for operators, the wait-queue model will be updated accordingly. Therefore, if a host h receives an additional volume of data to process for operator i, then the wait-queue model will be updated to reflect an increase in latency at operator j due to the increased wait time perceived via $W_{h,i,j}$.

4.7.7 Enacting new routing policies

Once the workloads of all operators have been balanced, the master can report the new set of routing tables back to the hosts.

Routing tables for Map operators can be implemented immediately once received by a host. In contrast to Reduce operations, the partitions identified in the routing table for a Map operator do no correspond to a specific set of data. Instead, we use them to represent a portion of the whole operator workload. Therefore, when a new routing table is received, a host will aim to consume a portion of the entire operator workload that corresponds to the portion of partitions it has been allocated. This change in workload can be implemented immediately, upon arrival of a new routing table, as Map data are processed independently of each other and therefore there is no need to synchronize routing policy changes.

Routing tables for Reduce operators, on the other hand, must have their changes imposed synchronously across all participating hosts. For each window of data in the stream, it must be the case that all data with a similar key must arrive at the same host. Therefore, when a change in the routing table occurs, it must happen between windows. In this way, the transition from one window to the next may also include a transition between routing policies. As transitions between windows are already synchronized by control punctuations, we use this same mechanism to implement a change to the routing table. Therefore, the master host will identify a window-boundary timestamp (being a multiple of all reduce window slide values) in which the changes should be implemented.

When a host receives a new routing table, it stores that routing table (and the timestamp at which it will become activated) alongside of the routing table that is currently in use. As data are produced for downstream Reduce operations, the timestamp on the data will be checked to see which routing policy will be used. The appropriate routing table will be identified and the data will be forwarded on appropriately.

4.7.8 Balancing operator load

The distributed implementation of C-MR uses an observation-oriented method of characterizing the potential of hosts because we consider them, and the operators they execute, to be black boxes. Hosts with a similar number of cores and clock speeds may process similar data items in different times due to the underlying differences in their micro-architectures (e.g., cache sizes). Our goal is to identify these differences at runtime and to balance workloads in a way that reduces latency.

As an example of this, we present the results of an experiment in which we use C-MR to execute a streaming image processing application. The application receives a stream of images of size

1920x1080, performs a grayscale conversion of the pixels within the image, computes the average pixel value, and performs a transformation on pixels beneath a threshold which was derived from the average pixel value.

We deployed this stream application onto four quad-core hosts. Each host had a different underlying architecture and processor speed. These processors included a 3.6 GHz AMD Phenom II X4 975, a 3.4 GHz AMD Phenom II X4 965, a 2.6 GHz AMD Phenom 9950, and a 2.4 GHz Intel Core2 Quad CPU. Upon starting the application, the load balancer uniformly distributed the load between the hosts. Only after observing reported statistics did the load balancer step in to take corrective measures.

Figure 4.7 shows the results of load balancing iterations over time. As each set of snapshots were sent back from the hosts to the master regarding average processing times and average wait times, the master used this information and predicted whether latency will be decreased through workload migrations. The figure shows not only the actual performance observed due to the actions of the load balancer but also the load balancer's prediction of its performance.

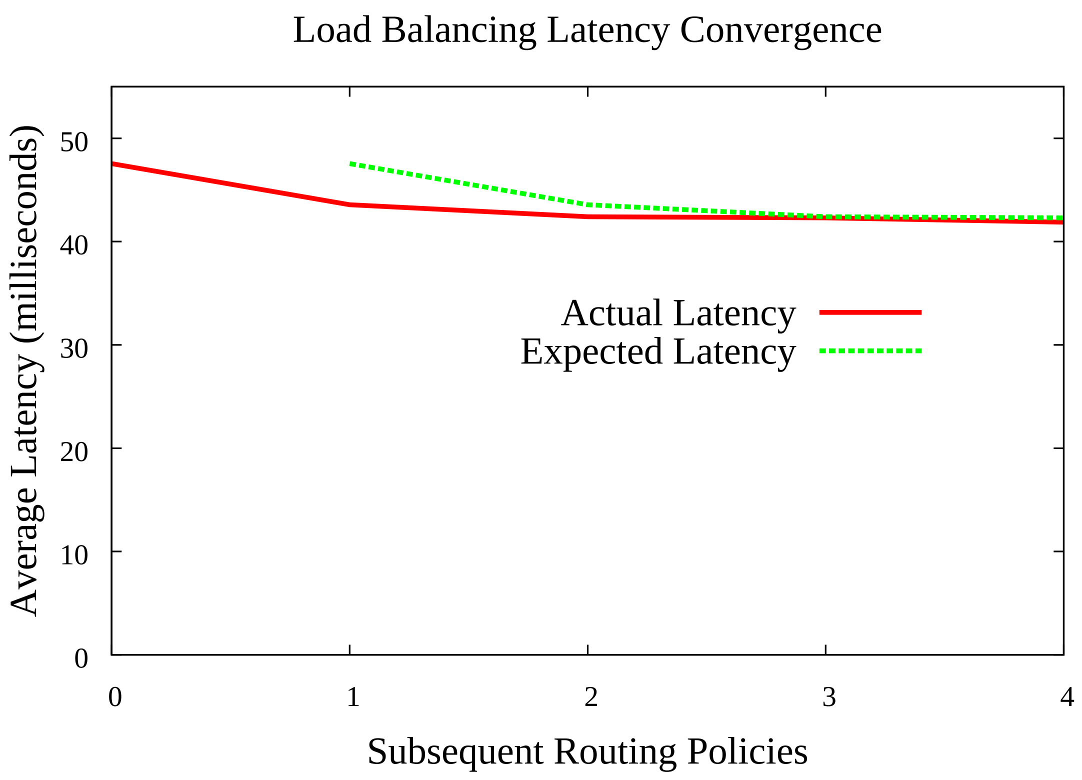


Figure 4.7: Latency convergence through load balancing on heterogeneous hosts

Initial reports showed that the time data spent on the hosts waiting to be processed was small – no host was overloaded by consuming 25% of the total workload. However, some hosts reported significantly lower processing times than the other hosts. This observation enabled the master to estimate that the average latency of processing data could be reduced by migrating workload partitions to the hosts with the lower processing times. As seen in the figure, this pattern of decreasing latency converges somewhat quickly after the fourth round of load balancing and ultimately finds an equilibrium at around about 40.25 milliseconds. At this point, moving additional partitions to the host with the lower processing time resulted in a slight overload which was observed as a sudden increase in the wait time of data on that host $W_{h,i,i}$. After observing this overload, workload partitions were migrated back before additional load balancing took place.

In contrast to other frameworks which literally try to balance load, C-MR attempts to reduce the average latency of results. In the previous experiment, the application started with a perfectly divided workload with each host doing the same amount of work. However, the load balancing algorithm was able to improve latency performance by skewing the distribution of the workload across the hosts such that the faster hosts would receive the larger portions of the workload. In this experiment, we found that skewing the workload distribution towards the faster hosts resulted in a 15% decrease in the average latency of results.

Chapter 5

Related Work

As mentioned in the introduction, a large body of work exists with regard to workflow processing for both stream and batch applications.

Fault-tolerant and partitioned parallel processing for stream processing engines was presented in Flux[37, 38]. Flux operators provided the encapsulation of content-sensitive partitioned parallelism with fine-grained load balancing for an individual parallel operator in a workflow. If multiple parallel operators existed, then a different Flux operator would be invoked for each of them. Flux therefore provided only localized load-balancing which resulted in an inability to perform inter-operator optimizations such as migrating computing potential from one operator to another. Our worked stemmed from the desire to provide a similar form of fine-grained load balancing but across all operators in a workflow in a way that would allow the computing resources to collectively be instrumented by a centralized scheduler.

The Hadoop Online Prototype (HOP)[18] (later MapReduce Online[19]) was developed concurrently to our own work and addressed online aggregation for Hadoop's MapReduce framework[2]. HOP enables pipeline parallelism between Map and Reduce tasks and supports incremental processing of Reduce tasks through periodic snapshot evaluation of partial intermediate results produced by Map tasks. By supporting pipelined parallelism to attach continuously running mappers and continuously running reducers, HOP was able to provide elementary stream processing support. However, it's reliance on Hadoop's MapReduce implementation as well as disk buffers and HDFS[1], to provide fault tolerance, resulted in a significantly constrained and inflexible infrastructure to support general stream processing. HOP suffers from pinned allocations of computing nodes to operators (via the Hadoop job and task trackers) which requires that an invocation of a continuous task corresponds to a task thread and occupied task slot on a computing node. This prevents elasticity and the ability to do any form of dynamic load balancing between the nodes producing output from one MapReduce job to the nodes consuming the input for another MapReduce job. Our solution provides not only fine-grained intra-job load balancing but also inter-job balancing while supporting workflow-wide operator scheduling to coordinate the impact of computing resources across a stream application.

In-situ MapReduce (iMR) [31, 42] 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. iMR computing nodes are fixed to specific Map or Reduce jobs and are therefore unable to benefit from any form of load balancing and cannot adapt to workload or resource volatility.

IBM's DEDUCE[28] 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.

Chapter 6

Conclusions

6.1 Generic node processing model

The processing model used by C-MR presents a number of measurable benefits as shown in this dissertation. While previous stream processing engines have traditionally pinned the contributions of a computing node to a specific subset of an application workflow, we have shown that a processing model which allows for computing nodes to contribute to multiple data-parallel operations simultaneously allows for: 1) finer-grained application load balancing; 2) diversification of processing responsibilities to withstand stream and resource volatility; and 3) and novel scheduling opportunities.

Our processing model orchestrates the forms of parallelism enacted by participating computing nodes on-the-fly relative to the performance objectives of the stream application. Static allocations of computing nodes to operators result in the rigid application of data and task parallelism. Our dynamic workload-wide scheduling strategy supports the real-time application of parallelism strategies, employing those that are found to be beneficial for the current processing environment relative to application objectives.

6.2 Single-host C-MR implementation

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 hybrid and probabilistic, 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 that is used to execute batch workloads on MapReduce workflows. To support stream applications, it is necessary to schedule an entire workflow of stream operators simultaneously 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.

6.3 Distributed C-MR implementation

We presented a distributed implementation of C-MR which supports execution on multiple hosts. We identify and address the major challenges in distributing the C-MR processing model, namely: 1) employing the hierarchical application of C-MR and its punctuation management scheme; 2) achieving latency-oriented load-balancing given black-box computing hosts and black-box workflow operators; and 3) synchronizing routing table changes that occur due to load balancing across participating hosts for window-based aggregation operators.

We also present preliminary results from the distributed implementation using multiple hosts to show the benefit the C-MR processing model brings to distributed workflow processing architectures.

6.4 Open Challenges

The C-MR processing model deviates significantly from prior stream processing models. Therefore, a large part of this thesis involved defining how to perform stream processing management tasks in this processing model (such as load balancing and operator scheduling). There are additional areas we have not explored such as a fault tolerance model for the use with the generic-node processing model. Additionally, we predict that there would be benefits from the application of the processing model to executing batch workflows.

6.4.1 A generic node fault tolerance model

MapReduce implementations, and frameworks which provide support for the continuous execution of MapReduce jobs, leverage distributed file systems to replicate and replay data in the occurrence of faults.

In a streaming context, however, regularly buffering data to disk is generally unacceptable due to the high overhead and incurred latencies. Therefore to protect from data loss in the case of faults, it is necessary to replicate and replay data from someplace faster than disks. That is, the data must be replicated and replayed from the main memory of other hosts in the set of distributed computers. One strategy is to simply clone the full set of distributed computers and processing workflow so that if one host goes off-line, one from the other set can pick up immediately where the failed host left off. However, this doubles processing and infrastructures costs while wasting computing resources that could otherwise be improving application performance. The generic node processing model allows for a novel extension to upstream backup [25] while providing true K-safety such that upstream data is preserved at k different locations. This differs from the standard K-safety approach to upstream backup which requires replaying data through the previous k upstream operators. Instead, it is possible to use the characteristics of the distributed workflow buffers and routing tables to replicate data across multiple hosts in a manner that does not require playback through k upstream operators..

Generally, when a host produces data for a downstream operator it may have to send portions of that data to many (potentially all) other hosts. This is plain to see with data destined for Reduce operations. Those data have their descriptors hashed into the partitioned space and will be shipped to the appropriate host while a portion of the data remains at the local host to be processed since the local host will also be the destination for a fraction of the data. If a fault occurs, and local copies of the data are not retained at the sending hosts, then we would suffer permanent data loss. To remedy this problem we can to do two things:

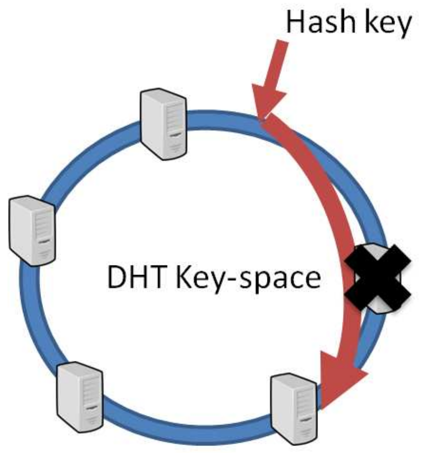


Figure 6.1: Backup data aggregation sites

1. **Retain a copy of the data at the sending host** until the receiving host has finished

processing the data and has transmitted the results of the work. If a host fails, all other hosts can resend replicas of the data lost on the failed host to the host with the next higher host ID number (or the lowest ID if the host with the highest ID failed).

2. **Send a copy of the data that isn't required to change hands** to another host for replication. If a host fails, then the upstream data it received from itself will be lost unless we can replicate it elsewhere. This data can be replicated to the host with the next highest ID. If a failure occurs, all other hosts will have also send their replicas to this host which will temporarily take over the responsibilities of the failed host.

Both of these steps ensure that another available host will be the recipient of backup data in the case of a fault, as can be seen in Figure6.1 which shows the ordering of hosts relative to ID in a ring structure. In order for this arrangement to support additional replication for K-safety, it will be necessary to ship replicated data to even more hosts. This would require individual hosts to not only retain their copy of data destined for another host but to also ship copies to the k hosts immediately following the target host. Likewise, the local data that a host will expect to process itself must also be sent to the next k hosts.

The hosts can release their replicated data once they all receive a punctuation downstream in the logical workflow that denotes that the relevant data has been successfully processed and transmitted. This removes the traditional requirements of having an additional form of control messages to manage the garbage collection of replications beyond the punctuations that are already in place using the generic node processing model.

6.4.2 Generic node processing for batch workflows

Additionally, we believe that the application of our model to batch processing frameworks will enable cross-job scheduling opportunities within a workflow. Batch processing frameworks have the general goal of trying to process as much as data as possible as quickly as possible thereby aiming for high throughputs.

Given large datasets that must be buffered to disk and a goal of optimizing for throughput, it is disk reads and writes that dominate per-tuple latency costs causing low throughputs. By applying our processing model to batch processing frameworks, it will be possible to utilize progressive scheduling techniques to decrease these costs.

While the goal of throughput-oriented scheduling policies has generally been to decrease context switches and increase cache coherency by minimizing pipelined and task parallelism in favor of data parallelism, it may instead be beneficial to maximize the amount of processing that can be done before writing to disk. This can be in the form of occasionally scheduling highly selective operators to free up shared-memory buffers or even by transferring data to computing nodes that have substantial availability in their shared memory buffers. Exploiting shared memory buffers before disk buffers will allow us to resort to disk access only when absolutely necessary and only when the cost does not outweigh that of potential data transfers to other computing nodes with available RAM.

The order of executing operators across a workflow also results in interesting opportunities for scheduling strategies in order to minimize the amount of data that is spilled to disk. For instance, if a substantial portion (or all) of the intermediate data produced by an operation is able to fit into shared memory without spilling to disk, then it can be advantageous to execute the immediate downstream operation next. This pattern can be continued down the workflow (with intermediate data being maintained in shared memory and without reading from disk) until some type of blocking operation is met. While following that path, and as the size of the resident working set potentially decreases, we can also begin to pre-fetch blocks from disk for other operations that we will later begin to execute.

Appendix A

Sample C-MR application

The following code defines Map and Reduce operations to parse an input stream of stock trades to evaluates a 60-second moving average for each symbol at 15-second intervals. `ParseStock` and `StockAverage` were written such that `StockAverage` can be re-used as a Combine step.

```
// Intermediate data format
struct IData {
    float value;
    int count;
};
// Mapper operator class that parses stock
// symbols and values from raw input strings
class ParseStock : public Map {
  void map(void* key, uint32_t keySize,
           void* val, uint32_t valSize,
           DataIterator* di)
  {
    string emitKey;   IData emitVal;

    // Parse "STOCK_SYMBOL STOCK_PRICE" string
    istringstream iss((char*)val);
    iss >> emitKey >> emitVal.value;
    emitVal.count = 1;

    emit((void*)emitKey.c_str(), emitKey.length()+1,
         (void*)&emitVal,        sizeof(emitVal),
         di);
  }
};

// Reduce operator class that computes the
// average stock price of a list of values
class StockAvg : public Reduce {
  void reduce(void* key, uint32_t keySize,
              DataIterator* di)
  {
    float sum = 0;   int count = 0;

    void* val;
    while ((val = di->getNextValue()) != NULL) {
      IData *data = (IData*)val;
      sum += data->value * data->count;
      count += data->count;
    }
    float avg = sum / (float)count;

    emit(key,         keySize,
         (void*)&avg, sizeof(avg), di);
  }
};

// Extracts data from a specified input stream
Data* readStream(FILE* inputStream) {
  Data* data = NULL;
  char  key[] = "stock_average";
  char* val = NULL;
  size_t bufSize = 0;

  if (getline(&val, &bufSize, inputStream) > 0) {
    data = new Data;
    data->setData( (void*)key, strlen(key)+1,
                   (void*)val, strlen(val)+1);
  }
  return data;
};

// Handles data received at the output
void outFunc(Data* data) {
  cout << data->timestamp.tv_sec << " "
       << (char*)data->key       << " "
       << *(float*)data->value   << endl;
}

int main(int argc, char** argv) {
  int winSize = 60; // 60 second window
  int winSlide= 15; // 15 second slide

  Query q;
  // Attach the input stream and stream
  // reading function to the workflow
  q.addInput(
      stdin,      // input stream
      readStream, // stream reading function
      1,          // # ops attached downstream
      1);         // downstream operator ID(s)
  // Insert a MapReduce job into the
  // workflow with a window specified
  q.addMapReduce(
      1,          // ID in workflow
      MapReduce(new ParseStock, new StockAvg),
      Window(winSize,winSlide),
      1,          // # ops attached downstream
      2);         // downstream operator ID(s)
  // Attach an output function to the workflow
  q.addOutput(
      2,        // ID in workflow
      outFunc); // output handler function

  // Instantiate cmrHost, load query, and run
  Host cmrHost;
  cmrHost.addQuery(q);
  cmrHost.run();
  return 0;
}
```

Bibliography

[1] Hadoop Distributed File System, http://hadoop.apache.org/hdfs/, The Apache Software Foundation.

[2] Hadoop MapReduce, http://hadoop.apache.org/mapreduce/, The Apache Software Foundation.

[3] Apache Oozie^{TM} Workflow Scheduler for Hadoop, http://incubator.apache.org/oozie/, The Apache Software Foundation.

[4] NVIDIA CUDA, http://developer.nvidia.com/category/zone/cuda-zone, Nvidia Corporation.

[5] OpenCL, http://www.khronos.org/opencl/, Khronos Group.

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

[7] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/, Amazon.com, Inc.

[8] Windows Azure Platform, http://www.windowsazure.com/, Microsoft Corporation.

[9] Rackspace Cloud Hosting, http://www.rackspace.com/cloud/, Rackspace US, Inc.

[10] Linode – Xen VPS Hosting, http://www.linode.com/, Linode, LLC.

[11] Cascading, http://www.cascading.org/, Concurrent Inc.

[12] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[13] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[14] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[15] Nathan Backman, Rodrigo Fonseca, and Uğur Çetintemel. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 1:1–1:5, New York, NY, USA, 2012. ACM.

[16] Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, and Uğur Çetintemel. C-mr: continuously executing mapreduce workflows on multi-core processors. In *Proceedings of third international workshop on MapReduce and its Applications*, MapReduce '12, pages 1–8, New York, NY, USA, 2012. ACM.

[17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[18] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[19] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, pages 313–328. USENIX Association, 2010.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[22] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.

[23] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[24] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[25] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 779–790, 2005.

[26] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21-23 2007. also as MSR-TR-2006-140.

[27] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1081–1092, New York, NY, USA, 2010. ACM.

[28] Vibhore Kumar, Henrique Andrade, Buğra Gedik, and Kun-Lung 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.

[29] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, March 2005.

[30] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1):274–288, August 2008.

[31] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth 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.

[32] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2. http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, September 2009.

[33] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1081–1090, New York, NY, USA, 2011. ACM.

[34] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[35] OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[36] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 49, Washington, DC, USA, 2006. IEEE Computer Society.

[37] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. Technical Report UCB/CSD-2-1205, U.C. Berkeley, 2002.

[38] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 827–838, New York, NY, USA, 2004. ACM.

[39] Justin Talbot, Richard M. Yoo, and Christos 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.

[40] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[41] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[42] Christopher J. Trezzo. Continuous mapreduce: An architecture for large-scale in-situ data processing. University of California, San Diego, Masters Thesis, 2010.

[43] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, 2003.

[44] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB '06: Proceedings of the 32nd international conference on Very Large Data Bases*, pages 775–786. VLDB Endowment, 2006.

[45] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 791–802. IEEE Computer Society, 2005.