DEDUCE: At the Intersection of MapReduce and Stream Processing

Vibhore Kumar, Henrique Andrade, Buğra Gedik, Kun-Lung Wu
IBM Thomas J. Watson Research Center, Hawthorne, NY 10532
vibhorek@us.ibm.com, hcma@us.ibm.com, bgedik@us.ibm.com, klwu@us.ibm.com

ABSTRACT

MapReduce and stream processing are two emerging, but different, paradigms for analyzing, processing and making sense of large volumes of modern day data. While MapReduce offers the capability to analyze several terabytes of stored data, stream processing solutions offer the ability to process, possibly, a few million updates every second. However, there is an increasing number of data processing applications which need a solution that effectively and efficiently combines the benefits of MapReduce and stream processing to address their data processing needs. For example, in the automated stock trading domain, applications usually require periodic analysis of large amounts of stored data to generate a model using MapReduce, which is then used to process a stream of incident updates using a stream processing system. This paper presents DEDUCE, which extends IBM's System S stream processing middleware with support for MapReduce by providing (1) language and runtime support for easily specifying and embedding MapReduce jobs as elements of a larger data-flow, (2) capability to describe reusable modules that can be used as `map` and `reduce` tasks, and (3) configuration parameters that can be tweaked to control and manage the usage of shared resources by the MapReduce and stream processing components. We describe the motivation for DEDUCE and the design and implementation of the MapReduce extensions for System S, and then present experimental results.

1 Introduction

The last few years have witnessed a rapid evolution in the data profile associated with modern data processing applications. Not only has the volume of stored data increased manifold, but the rate at which streaming data or the updates arrive has also witnessed a rapid pace of growth - often beyond the realm of traditional data processing technologies. Even more importantly, the data profile, from being composed entirely of the data at rest (stored data) or the data in motion (streaming data) is now some combination of the two. Examples include, finance applications like automated stock trading [8], scientific applications like the square kilometer array [20], network security and monitoring applications like the ones used for intrusion detection [17], and several others. Such applications are now faced with an increasingly pressing need to not only scalably (and often at regular intervals) analyze exceedingly large volumes of stored historical data for drawing inferences or detecting patterns, but also provide the capability to process very fast moving streams of data using the inferences or patterns mined from the stored data.

While innovations like MapReduce [7] and stream processing systems [1, 3, 5, 9, 14] are targeted towards addressing the needs of such applications, these innovations, however, offer only a part of the solution, addressing only the data at rest, in case of MapReduce or only the data in motion, as is the case with stream processing systems. Hence, application developers are not only forced to adopt two different programming paradigms, but also maintain two separate infrastructures for handling the data at rest and the data in motion. This independent and decoupled implementation approach often makes composing and managing the application more complex, and also limits the opportunities for infrastructure sharing and end-to-end optimization.

This paper describes DEDUCE, a middleware that offers a unified abstraction and runtime for addressing the needs of modern data processing applications, like the ones outlined above. DEDUCE, to the best of our knowledge, is the first effort that attempts to combine real-time stream processing with the capabilities of a massive data analysis framework like MapReduce. Towards this end, DEDUCE builds on our previous work on System S, especially the stream processing core [2] and the SPADE declarative stream processing engine [9], and augments their capabilities with those of the MapReduce paradigm. In particular, DEDUCE makes the following important contributions

- *Language Constructs:* DEDUCE extends SPADE'S data-flow composition language to enable the specification and use of MapReduce jobs as data-flow elements.
- *Reusable Modules:* DEDUCE provides the capability to describe reusable modules for implementing offline MapReduce tasks aimed at calibrated analytic models.
- *Runtime Support:* DEDUCE augments the System S runtime infrastructure to support the execution and optimized deployment of `map` and reduce tasks.
- *Control Parameters:* DEDUCE provides configuration parameters (e.g., update frequency, resource utilization hints, etc.) associated with the MapReduce jobs that can be tweaked to perform end-to-end system optimizations and shared resource management.

The following section describes a simplified market data processing scenario to further motivate the need for a DEDUCE-like middleware that is capable of handling both the data at rest and the data in motion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

1.1 Example: Market Data Processing

Financial market customers are in an arms race to process increasingly large amounts of market data with shorter and shorter latencies, and ever more sophisticated analytics. The profits, for such customers, are driven not only by the capability to keep shortest and fastest possible processing path from market data ingestion to order execution, but also on the capability to continuously generate/update the underlying analytic models employed to formulate trading strategies. Most financial firms build proprietary solutions, often consisting of two independent components, one in which streams of information such as market data are processed against an analytic model to produce the required results and the other responsible for generating/updating the analytic model.

As we articulated before, DEDUCE provides a common and easy to use high-performance development platform for such applications. It leverages and extends the capability of the SPADE dataflow composition language and the System S runtime framework to (1) easily specify the data processing needs of the application in question, (2) scalably and efficiently process high-rate data streams using the pre-existing capabilities of the System S stream processing middleware, and (3) build and manage sophisticated analytic models from massive amounts of stored data using MapReduce extensions provided by the DEDUCE middleware.

2 Background

DEDUCE middleware extends the System S-based SPADE declarative stream processing engine with the capabilities of MapReduce framework. This section provides a brief overview of the System S, SPADE, and the MapReduce framework.

2.1 System S

System S is a large-scale, distributed data stream processing middleware under development at the IBM T. J. Watson Research Center. It supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes. The System S runtime can execute a large number of long-running applications that take the form of data-flow graphs. A data-flow graph consists of a set of Processing Elements (PEs) connected by streams, where each stream carries a series of data tuples. The PEs are basic execution containers that are distributed over compute nodes and a node generally hosts multiple PEs. The compute nodes are organized as a shared-nothing cluster of workstations (COW) or as a large supercomputer (e.g., Blue Gene). The PEs communicate with each other via input and output ports, which are connected by streams.

The DEDUCE middleware makes use of the PE containers to spawn the `map` and `reduce` tasks across the cluster, it also extends the System S job management module to schedule and monitor the spawned `map` and `reduce` tasks. We will describe such extensions in more detail in Section 4.2.

2.2 SPADE

SPADE [9] (Stream Processing Application Declarative Engine) is the declarative stream processing engine of System S. It is also the name of the declarative language used to program applications. The language is used to express parallel and distributed data-flow graphs containing the operators and resulting streams required to carry out the actual processing for an application. SPADE also offers toolkits of type-generic built-in stream processing operators and a broad range of adapters to ingest data from outside sources and publish data to outside destinations. It is also possible to augment or build a new toolkit with additional user-defined built-in operators (UBOPs) or create new adapters.

DEDUCE makes use of the extensible nature of the SPADE language. It uses UBOPs to build several operators for the DEDUCE toolkit. For instance, it uses the UBOPs to build the MapReduce operator (representing a MapReduce job in a data flow) and adapters that interface with the distributed file systems (like HDFS [12]) for accessing data. This is described in more detail in Section 4.

2.3 MapReduce

The MapReduce programming model provides a highly scalable approach to conducting data-intensive computations over large amounts of stored data. At a high-level, the model involves specification of an algorithm using two functions, `map` and `reduce`.

The `map` function reads the input data using an application specific data format, processes the data and generates a set of intermediate `<key, value>` pairs. The `reduce` function operates on a subset of generated intermediate `<key, value>` pairs, such that all the pairs with the same key belong to one subset, and outputs one or more output `<key, value>` pairs in sorted order.

A typical implementation of the MapReduce programming model (e.g. Hadoop [11]) consists of a distributed file system, a runtime that supports distributed execution of `map` and `reduce` tasks on the nodes hosting the distributed file system and default implementation of some of the programming model concepts like input formats, output formats, partition function, among others.

One of the main advantages of the MapReduce programming model is its simplicity. The user is only supposed to specify an algorithm as a pair of `map` and `reduce` tasks that conform to the requirements of the programming model. The details of parallelization, task setup, concurrent data access and fault tolerance are all hidden from the user. This level of abstraction allows the user to focus more on the algorithm and makes it a very convenient platform for developing massively scalable data-analysis algorithms. DEDUCE relies on the ease of use and massive scalability provided by the MapReduce programming model to analyze massive amounts of data and possibly generate the models that could be used by real-time stream processing applications.

3 DEDUCE: System Overview

This section provides a quick end-to-end overview of the DEDUCE middleware. The main challenge when designing the DEDUCE middleware was the need to maintain the same level of ease of use, expressibility and performance as provided by the System S stream processing middleware and the MapReduce programming framework. Towards this end, DEDUCE represents a MapReduce computation as a data-flow element or operator to enable easy composition of MapReduce jobs and the stream processing logic.

A MapReduce operator accepts several configuration parameters (described later in Section 4.1.1), which among other things also contain handles to the specification of the `map` and the `reduce` tasks. The input data set to the MapReduce operator can either be pre-specified at compilation time or could be provided at runtime as a punctuated[1] list of files or directories. Once the input is available the MapReduce operator spawns a MapReduce job and produces a list of punctuated list of files or directories, which point to the output data. A MapReduce operator can potentially spawn multiple MapReduce jobs over the application lifespan but such jobs are spawned only when the preceding job (if any) has completed its execution. In a typical application multiple jobs can be cascaded together to create a data-flow of MapReduce operators or the output from the MapReduce operators can be read to provide updates to the stream processing operators.

[1] A punctuation is a special marker in the SPADE language that delimits the input or the output stream

Table 1: WordCount application in DEDUCE

```
[Application]
WordCount

[Typedefs]
typespace WordCount

[Modules]
# map module
module WordCountMap{
  input       In : {key:Integer, value:String};
  output      Out: {key:String, value:Integer};
  topology    stream Out(key:String, value:Integer) :=
                Aggregate(In<punct(),pergroup>)[key]{Any(key),Cnt(value)}
}

# reduce module
module WordCountReduce{
  input       In : {key:String, value:Integer};
  output      Out: {key:String, value:Integer};
  topology    stream Out(key:String, value:Integer) :=
                Aggregate(In<punct(), pergroup>)[key]{Any(key), Sum(value)}
}

[Program]
# mapreduce operator - represents the mapreduce job
stream CountedWords(outputFiles:String) := MapReduce()[
                    name:"WordCount";
                    in:"/user/wc/input";
                    out:"/user/wc/output";
                    map:"WordCountMap";
                    reduce:"WordCountReduce";
                    reduceCount:4;
                    inputFormat:"TextInputFormat";
                    outputFormat:"TextOutputFormat";
                    configurationDirectory:"/user/wc/conf";
                    maxTasks:10]{}
```

Table 1 lists the implementation of the well-known `wordcount` application as a MapReduce job in DEDUCE (note that the modules shown above use the stream manipulation operators from the SPADE language). Upon compilation the `wordcount` application compiles into three independent artifacts, which include the main `wordcount` application that is submitted to the runtime for execution and two executables which correspond to the `map` and the `reduce` tasks (using module definitions). The code for the `map` and `reduce` tasks is generated at compile time and is compiled to native executables for high performance. Once submitted for execution, the MapReduce operator contained in the application assesses the input data set and infers the number and the location of `map` and `reduce` tasks, which are then scheduled for execution. The MapReduce operator during the execution of the tasks keeps track of the progress of the MapReduce job and this information can be easily retrieved for visualization and monitoring purposes.

4 DEDUCE: Detailed Design

In the following sections we provide details about the language extensions that were made to the SPADE'S programming language, the addition of appropriate support to the System S runtime, the control parameters associated with the MapReduce jobs, and the compilation, startup and management of DEDUCE applications.

4.1 Language Extensions

The main considerations for DEDUCE-specific language extensions to the SPADE language were (1) to be able to easily specify the MapReduce jobs, (2) to support MapReduce jobs as composable data-flow elements, and (3) to provide the capability for creating domain-specific collection of `map` and `reduce` modules. In terms of language extensions DEDUCE consists of two important components – the DEDUCE Operator Toolkit and the Module specification framework, which are described next

4.1.1 DEDUCE Operator Toolkit

As mentioned in Section 2.2, an important feature of the SPADE language is its capability to allow the specification and use of new user-defined built-in operators. We made use of this capability of the language to create the DEDUCE operator toolkit. In particular, this toolkit contains the following operators:

Table 2: MapReduce operator parameters

Parameter Name	Description
`name`	unique name for the job
`input`	optional, files or directories for input data
`output`	directory for output data
`mapModule`	name of the `map` module
`reduceModule`	optional, name of the `reduce` module
`mapCountHint`	optional, a hint for the number of `map` tasks to spawn
`reduceCount`	number of `reduce` tasks to be spawned
`inputFormat`	the format of input data
`outputFormat`	the format for output data
`confDir`	optional, location of configuration directory
`partitionFn`	optional, the partition function

(a) *MapReduce Operator*
We model the MapReduce job as a SPADE operator. This approach simplifies the design of applications that combine the data at rest with the data in motion. While the input data set for a MapReduce job can either be specified as a parameter to the operator or as a punctuated input stream containing the location of directories or files to be processed, the output of the MapReduce job is written to a pre-specified location on the distributed file system and the location of this output data is optionally available as a punctuated output stream from the MapReduce operator. A combination of mandatory and optional parameters (see Table 2) are used to configure the operator. The MapReduce operator also accepts a set of control parameters and these will be described in Section 4.3.

(b) *Data Input & Output Operators*
To make it easy for the users to specify the MapReduce jobs and to utilize the output from the MapReduce jobs, the DEDUCE middleware provides an implementation of operators that can read and write data to the underlying distributed file system, while conforming to a certain data format. These operators besides being used by the users are also used by the `map` and the `reduce` tasks to access data that is assumed to be formatted in conformance to the `inputFormat` and the `outputFormat` parameters that are specified as part of the `MapReduce` operator. These operators also hide the underlying distributed file system by using a common interface to access the file system. Currently, DEDUCE supports access to HDFS and work is in progress to support other distributed file systems like KFS [13]. At runtime the input an output operators choose the right implementation for the file system interface to be employed by making use of information provided in the MapReduce configuration file.

4.1.2 Module Specification Framework

The *module* specification framework is DEDUCE'S extension to the SPADE programming language. It allows the user to specify an operator graph along with the declaration of the data format that can be processed by the graph, and the format of data that is produced as a result of such processing. Modules provide the mechanism to express complex `map` and `reduce` tasks in terms of regular SPADE operators. The DEDUCE application listed in Table 1 shows an example of a module definition. Together with UBOPs, modules provide an easy way to implement domain specific toolkits for MapReduce, described next.

4.1.3 Domain-Specific MapReduce Toolkits

DEDUCE supports the creation of domain-specific MapReduce toolkits. A toolkit is a collection of domain-specific UBOPs (possibly implemented by a domain expert) and pre-written modules that may use the UBOPs specified in the toolkit. The UBOPs con-

tained in the toolkit typically implement functional units that, for example, perform *fast Fourier transforms* on streamed digital signals. In other words, operators are the building blocks employed for specifying a `map`/`reduce` module. Pre-written modules can be directly used by DEDUCE developers to rapidly prototype and deploy domain-specific MapReduce jobs.

4.2 Runtime Components

Additional runtime components (beyond what exists in System S) are needed to support the MapReduce programming model. These include a distributed file system to provide block-level access to data and a task management infrastructure that spawns and tracks the `map` and `reduce` task.

4.2.1 Distributed File System

Instead of developing our own distributed file system, we chose to write adapters that can interface with the existing distributed file systems. These adapters were implemented as a set of UBOPs. We already have an adapter for HDFS and, as previously mentioned, work is in progress to implement the same for KFS.

4.2.2 Scheduling `map` & `reduce` tasks

For the purpose of task management we extended the System S job management component. Termed *TaskManager*, the new component is capable of accepting requests for spawning and terminating `map` and `reduce` tasks. While, the termination requests are immediately executed, the spawn requests are added to a queue from which the next task to spawn is chosen based on its `uvalue` (a representation of its scheduling priority). The `uvalue` for a task can be based on a number of factors that include time of task arrival, task priority, progress percentage of the associated global job, etc. DEDUCE allows a default implementation which uses the time of task arrival parameter to implement a FIFO scheduling. However, users can override this default implementation to enforce custom scheduling policies. Once ready to be scheduled, every task is spawned as a new processing element.

4.3 Control Knobs

The MapReduce operator provides three optional control parameters that can be tweaked to adjust the physical layout and resource utilization level of a DEDUCE MapReduce job. These include:

- `updatePeriod` – this parameter refers to the expected time in which the operator can expect the arrival new input data, and therefore the operator should finish a particular job in no more than the specified `updatePeriod`. This has implications on the maximum number of concurrent `map` and `reduce` tasks that the operator can be expected to execute.
- `maxTasks` – this parameter refers to the maximum number of concurrent `map` and `reduce` tasks that can be spawned by the MapReduce operator. This affects the completion time of the job, and can assist in maintaining manageable loads across a system hosting multiple MapReduce jobs.
- `priority` – this parameter refers to the priority of the spawned MapReduce job. A higher priority task will, when possible, be preferentially scheduled by the `TaskManager`.

4.4 DEDUCE Application

A typical DEDUCE program represents a data-flow which is a combination of a regular stream processing application and one or more MapReduce operators[2]. The MapReduce operators analyze large volumes of stored and possibly transient data sets and, in several cases, build analytic models that are then forwarded to the stream processing component for use in real-time data analysis. A partial listing of an application (model-assisted bargain index computation) written in DEDUCE, and a schematic representation of the same is shown in Figure 1. Next, we provide a brief description of the application.

[2]There are no underlying assumptions that make the usage of DEDUCE as a regular MapReduce only platform inefficient or difficult.

4.4.1 Model Assisted Bargain Index Computation

This example application is designed to find when a stock is available at a *bargain* price in the stock market. The application does this by computing normalized bargain index, which is determined based on recent market value at which the stock has been selling, and inputs from automated analysis of recent news feeds, filings, etc. (which may run into gigabytes) about the institution that the stock represents. A good measure of recent market value of a stock is volume weighted average price or `VWAP`, which is computed over a sliding window as $\sum price \times volume / \sum volume$. The calculation of `VWAP` is done by the top most segment shown in Figure 1, which consists of the Functor that forwards the trades along with the $product$ of $price \times volume$ for each trade, followed by an Aggregator that computes for each window the sum of forwarded $product$ and the $volume$ for each stock symbol, and finally the Functor operator computes the `VWAP` for each stock symbol. To compute the simple bargain index, the quoted price is compared with the `VWAP` computed by the segment described above, this is shown as the Functor and the Join operator segment in the schematic. Finally, the bargain index is normalized using a model which is generated by the MapReduce job to take into account the effect of events outside the market. The MapReduce job periodically receives a list of files from external data source and it uses them to compute the desired model using the two `map` and `reduce` modules specified as part of the application.

5 Experimental Evaluation

In this section we report the results from our evaluation of DEDUCE as a MapReduce platform, and results from our study of interaction between the streaming component and MapReduce job(s) that belong to the same DEDUCE application. All the experiments detailed in the following sub-sections were performed on a set of 84 reserved nodes, using all or fewer nodes from the set to study various aspects of the middleware. The reserved set contained two types of node, the first one - having 14 nodes had 2 Intel Xeon 3.4 GHz CPUs, 6 GB RAM and 2 x 36.4 GB hard disks; the second set - having 70 nodes had 2 x dual core Intel 3.0 Ghz CPUs, 8 GB RAM and 2 x 73.4 GB hard disks. For our experiments we do not differentiate between these nodes.

5.1 Performance & Scalability

The first set of experiments was conducted to demonstrate the scalability of the DEDUCE middleware and to compare the performance of DEDUCE with that of Hadoop [11] (release 0.17.1). We conducted these experiments for both I/O (disk) intensive and compute intensive MapReduce jobs. For the following experiments, DEDUCE makes use of HDFS as the underlying distributed file system and accesses the HDFS using Java native interface (JNI).

5.1.1 I/O Intensive Workload

For measuring the performance of DEDUCE as compared to Hadoop with respect to I/O bound jobs we make use of string *grep and count* scenario. We used the network security logs from one of the IBM subsidiaries and ran the analysis. The job consisted of searching a particular set of events and counting the occurrence of each such type of event. We ran the experiments for varying sizes of the input

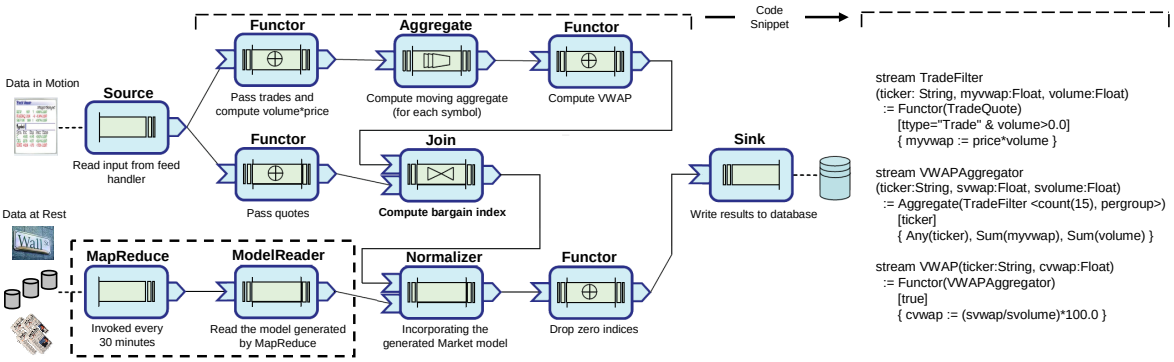

Figure 1: A schematic representation of the model assisted bargain index computation application

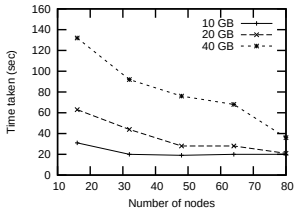

Figure 2: DEDUCE performance for I/O bound job

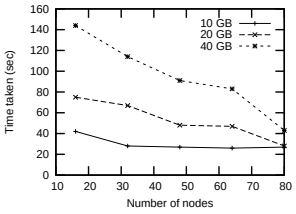

Figure 3: Hadoop performance for I/O bound job

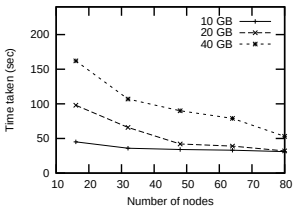

Figure 4: DEDUCE performance for compute bound job

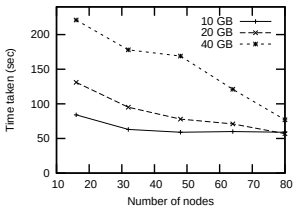

Figure 5: Hadoop performance for compute bound job

log file and the number of nodes used for data storage and computation. In the results shown in Figure 2 and Figure 3, we observe that DEDUCE performs significantly ($\approx 33\%$ reduction in job completion time) better than Hadoop for smaller sizes of the log file, while it does perform better even for larger file sizes the improvement is not as significant. This can be attributed to the overhead imposed by JNI for the DEDUCE implementation, which starts being the dominant factor for larger files. In terms of scalability, as expected, we observe that for smaller size of the log file the gain in performance stagnates and does not improve with addition of more nodes. However, for larger files we see a continuous increase in performance with the addition of nodes.

5.1.2 Compute Intensive Workload

The next experiment in the set was focused on determining the performance of DEDUCE in comparison to Hadoop with a compute intensive workload. We used the same MapReduce job as above except for a change in the `map` task, which was modified to include several floating point operations for every byte of input data. This workload corresponds to situations when MapReduce job is used for summarizing training data and creating analytic models. In this setup, DEDUCE outperforms Hadoop significantly. For instance, when using 48 nodes DEDUCE takes 90 seconds to process 40 GB of data, as opposed to 169 seconds taken by Hadoop. The results are shown in Figure 4 and Figure 5. DEDUCE makes use of the code generation and other high-performance features that are built into System S middleware for running the MapReduce jobs and this is reflected in the results.

5.2 Interaction between streaming application & MapReduce

This experiment was designed to demonstrate the importance of effective scheduling mechanisms for a middleware like DEDUCE. As mentioned in Section 4.2.2, DEDUCE allows the users to extend the scheduling policy by taking into consideration a number of factors that range from time of arrival of a job, job priority, job progress percentage, etc. We conducted this experiment on a set of 8 nodes, and deployed a DEDUCE application with a streaming component consisting of a source that continuously generated data by reading from a file, a time window based aggregator, a compute intensive operator, a filter operator and a sink in series – replicated twice, while the MapReduce component with 20 `map` and 4 `reduce` tasks was supposed to ingest 10 Gigabytes for a `wordcount` like application. We calculated the number of tuples processed by the streaming application and the percent of data file ingested by the MapReduce Job in a two minute window for different values of `maxTasks` parameter of the MapReduce job. Results shown in Table 3 show that there is be a strong level of interaction between the streaming application and the associated MapReduce job, and parameters like `maxTasks` can be used to trade delay in propagation of updates through the streaming application for faster generation of models and vice-versa.

Table 3: Interaction: streaming application & MapReduce jobs

`maxTasks`	Tuples processed	percentage of file processed
4	61389	59
8	57991	67
12	48567	78
16	42898	93

6 Related Work

This section provides a brief survey of the existing body of research in the related domains of stream processing systems, MapReduce and larger data analysis systems.

6.1 Stream Processing Systems

Recent years have witnessed a lot of activity in the domain of data stream processing both in the industry and the academia. While academic projects like Borealis [1], STREAMS [3], TelegraphCQ [5] and others [14] have focused on the more theoret-

ical aspects of the paradigm, industry activity [2] has primarily focused on commercialization and wide availability of this novel data processing paradigm. In addition, researchers have also focused on issues like load-distribution [2], load-balancing [21] and fault-tolerance [4] in stream processing systems. Stream processing systems typically provide system support for instantiating real-time data analysis applications that consume high-rate data streams. However, such systems are not meant to analyze large volumes of stored data and often rely on data warehouses and other ad hoc mechanisms for analyzing stored data, and such analysis is often the source of analytic model used by stream processing operators.

DEDUCE takes a more holistic approach to the problems associated with processing modern day data. It not only provides the stream processing capabilities, but also allows the users to specify and instantiate MapReduce jobs as an element of the data-flow.

6.2 MapReduce

The MapReduce programming model [7] has gained significant acclaim for its ease-of-use, scalability, and fault-tolerance. It offers a parallelization framework that depends on a runtime component for scheduling and managing parallel tasks [11] and a distributed file system that is responsible to providing parallel access to different blocks of the stored data [10, 12]. The success of the programming model has resulted in the development of an open source implementation of its runtime called Hadoop [11] and the Hadoop distributed file system [12]. A number of efforts are also focused on developing languages that help the specification of MapReduce jobs [18, 19]. Recent research efforts have also focused on adapting MapReduce for database like operations using a `map-reduce-merge` framework [22]. Another effort focuses on implementing well-known machine learning algorithms for the MapReduce programming model [15]. Although, MapReduce has also received a lot of criticism for ignoring a lot of research on task parallelization and data modeling conducted by database researchers [16], it continues to be the new preferred platform for developing large-scale analytics on static data. As compared to these infrastructure and language efforts DEDUCE is aimed at completing the real-time data analysis loop by supporting both the specification of streaming application and the associated analytics.

6.3 Data Analysis Systems

Data analysis systems have evolved from monolithic data warehouses [6] of the past to the modern day real-time data analysis systems [2]. The nature of the task has changed from offline to online and therefore new sets of challenges have been posed to the research community. Modern day data analysis systems should not only scale with the volume of stored data, but also scale with the rate at which live data arrives. In addition to that, modern frameworks must be able to provide processing capabilities to deal with both structured and unstructured data.

DEDUCE is an attempt to bring together the benefits of two emerging data analysis technologies in a way such that it addresses the needs of several modern day data processing applications. However, traditional technologies like warehouses and relational databases still continue to be very useful as far as consistency, persistence and transactions are concerned.

7 Conclusions & Future Work

This paper described DEDUCE, a middleware that aims to address the needs of the modern data processing applications that need to handle both data at rest and data in motion. In doing so, DEDUCE extends the SPADE declarative stream processing engine with the capabilities to specify and instantiate MapReduce jobs, thereby facilitating the development of real-time data analysis applications that must rely on model derived from historical or otherwise stored data. DEDUCE has been developed keeping in mind the ease of use and the need for high-performance and it seamlessly integrates a MapReduce job as an element in the larger data analysis flow. Additionally, DEDUCE also provides parameters associated with MapReduce jobs which can be used to tune the resource utilization or execution times of such jobs.

In terms of future work, although DEDUCE provides capabilities to the user to extend the scheduling mechanism for `map` and `reduce` tasks but there are a lot of opportunities to continue work in this direction. This becomes particularly important when the streaming application and MapReduce jobs share the same infrastructure. We would also like to extend this work to address the issue of relationship between the quality of generated models, their impact on system resources and the quality of generated real-time data. We are also looking into the issues related to development of MapReduce toolkits, and if such toolkits can be profiled in advance to assist in enforcement of service level agreements on the deployed MapReduce jobs.

Acknowledgments

We would like to thank Nagui Halim, the principal investigator of the System S project, for his continued support and invaluable guidance throughout the development of the DEDUCE middleware.

8 References

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] L. Amini et al. Spc: a distributed, scalable platform for data mining. In *DMSSP*, 2006.

[3] A. Arasu et al. Stream: the stanford stream data manager (demonstration description). In *SIGMOD*, 2003.

[4] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transansactions on Database Systems*, 33(1), 2008.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, 2003.

[6] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1), 1997.

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

[8] IBM - financial markets. `http://ibm.com/financialmarkets/`.

[9] B. Gedik et al. SPADE: the System S declarative stream processing engine. In *SIGMOD*, 2008.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5).

[11] Hadoop. `http://hadoop.apache.org/core`.

[12] The hadoop distributed file system: Architecture and design. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[13] High performance scalable storage. `http://kosmosfs.sourceforge.net`.

[14] V. Kumar et al. Implementing diverse messaging models with self-managing properties using IFLOW. In *ICAC*, 2006.

[15] Apache mahout. `http://lucene.apache.org/mahout`.

[16] Mapreduce: A major step backwards. `http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html`.

[17] S.-H. Oh, J.-S. Kang, Y.-C. Byun, G.-L. Park, and S.-Y. Byun. Intrusion detection based on clustering a data stream. *International Conference on Software Engineering Research, Management and Applications*, 0, 2005.

[18] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, New York, NY, USA, 2008.

[19] R. Pike et al. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal: Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 2005.

[20] Square kilometer array. `http://www.skatelescope.org`.

[21] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, 2005.

[22] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.