

IncMR: Incremental Data Processing based on MapReduce

Cairong Yan^{*,†}

^{*}Department of Computer Science and Technology
Donghua University, Shanghai, China
cryan@dhu.edu.cn

Xin Yang[†], Ze Yu[†], Min Li[†], Xiaolin Li[†]

[†]Scalable Software Systems Laboratory
University of Florida, FL, USA
{xinyang, zeyu, minli}@ufl.edu, andyli@ece.ufl.edu

Abstract—MapReduce programming model is widely used for large scale and one-time data-intensive distributed computing, but lacks flexibility and efficiency of processing small incremental data. IncMR framework is proposed in this paper for incrementally processing new data of a large data set, which takes state as implicit input and combines it with new data. Map tasks are created according to new splits instead of entire splits while reduce tasks fetch their inputs including the state and the intermediate results of new map tasks from designate nodes or local nodes. Data locality is considered as one of the main optimization means for job scheduling. It is implemented based on Hadoop, compatible with the original MapReduce interfaces and transparent to users. Experiments show that non-iterative algorithms running in MapReduce framework can be migrated to IncMR directly to get efficient incremental and continuous processing without any modification. IncMR is competitive and in all studied cases runs faster than that processing the entire data set.

Keywords: MapReduce, Incremental data processing, State, Data locality, Compatible

I. INTRODUCTION

Applications with incremental or continuous input widely exist. Online aggregation and incremental cube building can provide rapid and efficient analysis for OLAP [1]. Incremental machine learning techniques are usually used to solve large-scale knowledge discovery problems. Incremental web pages crawling, indexing, and ranking are critical to search engine [2] [3]. Typically, incremental analysis can achieve more efficient data processing than batch processing [4].

MapReduce is emerging as an important programming model for data-intensive applications. The model proposed by Google is very attractive for ad-hoc parallel processing of arbitrary data. It shows good performance for batch parallel data processing [5]. MapReduce enables easy development of scalable parallel applications to process vast amount of data on large clusters of commodity machines. Its popular open-source implementation, Hadoop [6], developed primarily by Yahoo and Apache, runs jobs on hundreds of terabytes of data [2]. Hadoop is also used at Facebook, Amazon, and Last.fm. Because of its high efficiency, high scalability, and high reliability, MapReduce framework is used in many fields, such as life science computing [7], text processing [8], web searching [9],

graph processing [10], relational data processing [11], data mining, machine learning [12] [4], and video analysis [13]. Additionally, the framework is not only used in traditional clusters, multi-core and multi-processor systems [14] [15], and heterogeneous environments [16], but also used in systems with GPU and FPGA, and mobile systems [17] [18].

Such advantages attract researchers to apply it on incremental data processing. Most studies tend to modify its framework as little as possible to achieve their requirements. Both CBP of Yahoo and Percolator of Google support incremental data processing framework based on MapReduce model [19] [20]. Incoop is a generic MapReduce framework, which allows existing MapReduce programs, not designed for incremental processing, to execute transparently in an incremental manner. Almost all these frameworks support incremental data processing by combining the new data with the data derived from previous batches or iteratively calculating the new data. State is a fundamental requirement for tracking prior jobs. Generally, both the intermediate outputs and the final results of the previous jobs can be taken as state. In MapReduce framework, a job usually splits the input data sets into independent chunks which are processed by map tasks in a completely parallel manner. Then the outputs of maps are sorted and forwarded to reduce tasks. There are two kinds of computing outputs separately from map tasks and reduce tasks. The former is the intermediate result and saved in local nodes. The latter is the final result and saved in the distributed file system. If we do not modify the application and directly do incremental computation, it is not feasible to only keep the previous reduce outputs because different applications running in the framework will have different data flows and computing processes. In order to retain the original algorithms and program flows, the intermediate outputs are necessary to be saved. In this paper, we take the intermediate results as a part of the state.

When we design and implement an incremental computation framework, many factors including algorithm accuracy, run time, and space overhead need to be taken into consideration. This paper puts focus on the transplantation of parallel algorithms based on MapReduce model and compatibility of non-incremental and incremental processing. A parallel programming framework is presented, which aims to be compatible with the original MapReduce APIs so that programmers do

The work presented in this paper is supported in part by National Science Foundation (grants CCF-1128805, OCI-0904938, and CNS-0709329).

not need to rewrite the algorithms. It makes the following contributions:

Incremental MapReduce framework. It supports incremental data input, incremental data processing, intermediate state preservation, incremental map and reduce functions. The input manager can dynamically discover new inputs and then submit jobs to the master node.

Dynamic resource allocation based on the state. The state provides an important reference for resource request and allocation of the next execution. State information includes prior processing results, intermediate results, execution time, and the number of reduce tasks. Input data, acting as the observation, will change the current state into a new state after the job finishes.

Friendly APIs for applications. Method `submitJob()` in Class `JobClient` is overloaded. Users only need to follow the method parameters to submit their jobs without modifying their algorithms or application programs. Furthermore, for continuous inputs, users can get updated outputs in time.

The rest of the paper is organized as follows. Section II presents the problem description and incremental computation framework. Section III presents the main techniques in IncMR framework. A case study and related state management algorithms are presented in Section IV. The system performance evaluation metrics and preliminary evaluation results are given in Section V. Section VI presents the related work. Section VII concludes the paper.

II. INCMR DATA PROCESSING FRAMEWORK

The data explosion is creating a surge in demand of storage and computing, e.g., web log analysis and web site statistics. Incremental data processing approaches can help them lighten the computation burden and save computation resources. This section first introduces several observations from the MapReduce implementation, then proposes an incremental computation framework.

A. Limitations of MapReduce

MapReduce programming model is popular due to its simplicity and high efficiency. In MapReduce framework, when a job is submitted, the related input is divided into fixed-size pieces called blocks which are located in different data nodes. A job creates multiple splits according to the number of blocks and each split is processed independently as the input of a separate map task. A map task in a worker node runs the user-defined map function for each record in the split and writes its output to the local disk. When there are multiple reducers, the map tasks partition their outputs and each partition is for one reduce task. Reduce tasks will write their outputs to the distributed file system. Despite its powerful automatic parallelization with strong fault-tolerance, the original MapReduce exhibits the following limitations.

Stateless. When map and reduce tasks finish, they write their outputs to a local disk or a distributed file system, and then inform the scheduler. When a job finishes, related intermediate outputs will be deleted by a cleanup mechanism.

When new data or input arrives, a new job needs to be created to process it. This is just like HTTP, a stateless protocol, which provides no means of storing a user's data between requests. To some extent, we can also say that MapReduce model is stateless.

Stage independent. A MapReduce job can be divided into two stages: map stage and reduce stage. Each stage will not interrupt the other's execution. In the map stage, each computing thread executes the map method according to the input split allocated to it, and writes the output to the local node. In the reduce stage, each reduce thread fetches input from designate nodes, executes the reduce method, and writes the output to the specified file system. All map tasks and reduce tasks execute their codes without disturbing each other.

Singe-step. Map tasks and reduce tasks will execute only once orderly for a job. Map tasks may finish at different times, and reduce tasks start copying their outputs as soon as all map tasks complete successfully.

B. Extension of MapReduce

Because of the limitations mentioned above, MapReduce model has to be extended for incremental computation. Table I shows some typical solutions for incremental or continuous computation related to MapReduce programming model and other similar models. They feature different input patterns, separate or coupled control approaches between state and dataflow, compatible or new added interfaces. Batch parallel processing refers to those that provide high efficient large-scale parallel computation with one batch input and produce one output such as Google MapReduce [5], Hadoop [6] and Dyrad/DryadLINQ [21] [22]. Incremental algorithms, more complicated than the original algorithms, can improve the runtime by modifying algorithms. Its input includes newly added data and the latest running result [4]. Continuous bulk processing is another kind of solution to support incremental processing by providing new primitives for developers to design delicate dataflow oriented applications. It takes the intermediate results of the prior executions as a part of explicit input. CBP of Yahoo and Percolator of Google both provide such incremental computation frameworks [19] [20]. Incremental computation based on MapReduce, making full use of MapReduce programming model, supports incremental processing by modifying the kernel implementation of map and reduce stages. Because HDFS does not support appends currently, some approaches also modify the distributed file system to support incremental data discovery and intermediate result storage [23].

IncMR is an improved framework for large-scale incremental data processing. The framework, as shown in Figure 1, inherits the simplicity of the original MapReduce model. It does not modify HDFS and still uses the same APIs of MapReduce. All algorithms or programs can complete incremental data processing without any modification. Specifically, it translates meets the following objectives.

Compatibility. Original MapReduce interfaces are retained to avoid incompatibility with the existing implementation

TABLE I
TYPICAL INCREMENTAL DATA PROCESSING APPROACHES

Approaches	Input	State&dataflow	Advantages	Disadvantages
batch parallel processing	one batch	stateless	simple implementation; one input, one output	waste of time for re-running the entire data set
incremental algorithm	incremental	stateful&uncoupled	unchanged parallel framework	complicated algorithm design; manual user interaction
continuous bulk processing (CBP)	incremental	coupled	new model and primitives	delicate data flows built for different applications
incremental computation based on MR	incremental	coupled	same algorithm; same MapReduce APIs	modified HDFS; modified programming model
IncMR	incremental	coupled	same algorithm; same MapReduce APIs; same HDFS without modification	repartition of state data

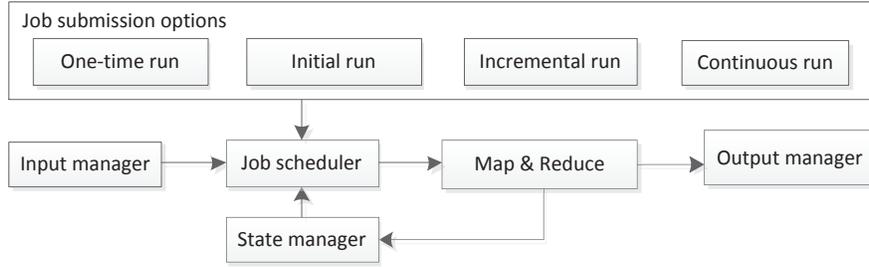


Fig. 1. Framework for incremental data processing based on MapReduce

of applications. New interfaces are added by overloading methods. HDFS is still used without modification.

Transparency. Users do not need to know how their incremental data are processed. All state data including their storage paths are transparent to users.

Reduced resource usage. Computing resource request and allocation are determined by the historical state information and current added data size. Dynamic scheduling decision is useful for reducing overhead.

Several important modules are added in IncMR framework. The input manager is used to detect the newly added data automatically. According to the execution delay configuration or input size threshold, it determines if a new data processing will be started. Job scheduler, different from traditional job scheduler which determines the number of tasks to run only according to the configuration file defined in advance, takes the state into consideration when choosing nodes for reduce tasks. State manager stores all needed information for incremental jobs and provides decision support for job scheduler. Output manager is responsible for the storage and update of all outputs.

III. INCMR INTERNAL MECHANISM

In IncMR framework, input, state, data flow, control flow management, and resource allocation are key components.

A. Job submission options

The original input interface of Hadoop MapReduce requires users to specify the input files. In IncMR, the concept of job

is extended. There are four kinds of job submission, one-time job, initial job, incremental job, and continuous incremental job.

- **One-time run.** To be compatible with the conventional use case, IncMR provides almost the same client submission APIs. No state needs to be saved when the job is one-time run.
- **Initial run.** The job is executed just like one-time run. However, the execute state of the job needs to be saved. If users do not request to terminate the whole job, system will keep the state.
- **Incremental run.** Based on initial run, incremental run requires system to fetch the state of the prior runs and combine it with the new added data. At the same time, the state will be updated according to the current run.
- **Continuous run.** For some data analysis applications, new inputs are continuously added to the system, so the system needs to discover these new data automatically and submits jobs automatically.

B. Iterative computation

Incremental computation means processing new request based on the former results. MapReduce model can also support iterative processing [7].

Suppose input data is $D_i = \{d_0, d_1, \dots, d_i\}$, in which d_i is the i^{th} added input and D_i is the total input including d_0 to d_i . The outputs of map stage and reduce stage for D_i are M_i and R_i respectively.

TABLE II
STATE VARIABLES

$$M_i = \text{map}(D_i) = M_{i-1} \bigcup \text{map}(d_i) = \bigcup_{k=1}^i \text{map}(d_k) \quad (1)$$

$$R_i = \text{reduce}(M_i) = \text{reduce}\left(\bigcup_{k=1}^i \text{map}(d_k)\right) \quad (2)$$

The data size of $\text{map}(d_k)$ is important and saved persistently so as to support incremental computation.

Suppose Tm_i is the computation time of map stage for the total input including the i^{th} , Tr_i is the computation time of reduce stage, tm_i and tr_i are execution time of d_i at the i^{th} time respectively. Tm'_i and Tr'_i are execution time of map tasks and reduce tasks in IncMR.

We denote the execution time using the batch approach as T_i , and the execution time using the incremental approach as T'_i .

$$T_i = Tm_i + Tr_i \approx \sum_{k=1}^i tm_k + \sum_{k=1}^i tr_k \quad (3)$$

$$T'_i = Tm'_i + Tr'_i \approx tm_i + \sum_{k=1}^i tr_k \quad (4)$$

From the above equations we can see that incremental computation time is much less than batch parallel processing approach although their execution time of reduce stage is the same. So IncMR model shows advantage in data processing comparing with MapReduce model. We discuss the implementation details of IncMR framework in the next section.

C. State management

In the MapReduce model, execution of a map task is divided into two phases. The map phase reads split, parses it into records and applies map function to each record. The commit phase registers the final output with the TaskTracker and informs the JobTracker that the task has finished. The output of a map task is stored in its local node. Execution of a reduce task is divided into three phases. The shuffle phase fetches input data from every map task's output. The sort phase groups the input and the reduce phase applies the user-defined reduce function to each key and corresponding list of value [24]. At the end, the output of reduce tasks will be taken as the final output of the job written in distributed file system.

In our IncMR, new map tasks for incremental job process new input and they have no relation with the prior jobs. To new reduce tasks, all they need to process includes the outputs of both the prior map tasks and the current map tasks.

For supporting incremental computation, some prior information has to be saved as state. State is described by a set of variables. The global variable records the information of all incremental jobs and local variables record the details of each kind of incremental job and their execution results. Table II lists all the variables and figure 2 shows their relations.

One IncJobList instance includes several IncJob instances. Each IncJob instance includes one OutputList instance and

Scope	Variables	Description
Global	incJobList	to record all the jobs that need to be executed incrementally
Local	outputList	to specify the storage location of the intermediate results for each kind of incremental jobs.
Local	outputInfo	to record each map state of each run
Local	execList	to record execution state of each kind of incremental jobs
Local	execInfo	to record each reduce state of each run

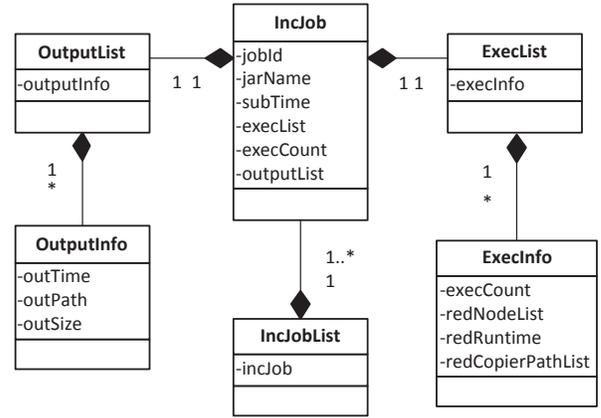


Fig. 2. The relations of IncMR state variables

one ExecList instance which are the records of one initial run, incremental run or continuous run.

D. IncMR data flow

For map tasks, the execution flow is the same as that in original MapReduce model. The job scheduler will create map tasks according to the size of incremental input. For reduce tasks, the execution of function $\text{reduce}()$ is almost the same as that in original MapReduce. However, its copy phase is different because the data resource includes not only the current map outputs but also the state. The IncMR data flow is shown as figure 3 where the internal data flows of both map task and reduce task have not been changed. The biggest difference between IncMR model and MapReduce model is the data copy phase of reduce tasks. Where to find the data and how to copy the data become the key items to be considered.

Storage means of historical computing data. Incremental process can not do without historical computing data or results. These data can be stored in a special storage node, master node, or task nodes. Considering the data size, network transferring bandwidth and data locality, we store these data in task nodes directly in IncMR model. The related paths are recorded as state variables by which incremental run can find the prior computing results so as to save computing time.

Repertition of prior map outputs. In MapReduce model, partitioner controls the partitioning of the keys of the inter-

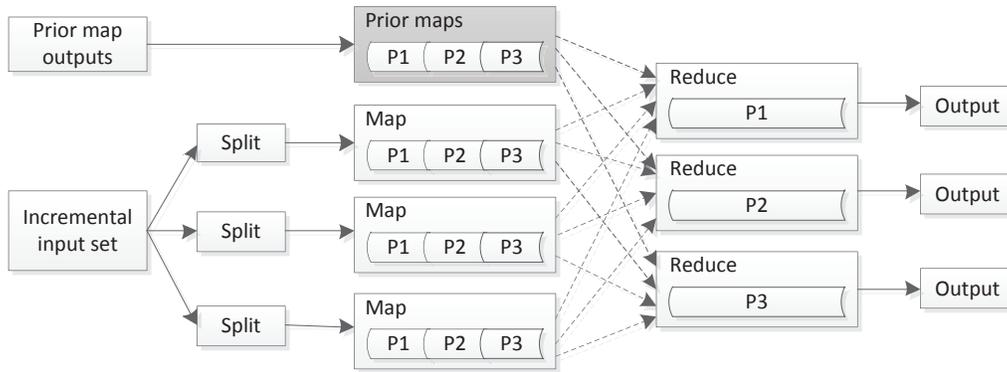


Fig. 3. IncMR data flow with three map tasks and three reduce tasks

mediate map outputs. The key is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job. Hence partitioner also controls which of the reduce tasks the intermediate key is sent for reduction. Basically, the partition function will not be changed during incremental execution. However, as the amount of data increases continuously, the number of reduce tasks need to be increased to satisfy the user's deadline requirements. In this case, the prior map outputs have to be repartitioned according to the new reduce number.

E. Locality control and optimization

Job scheduler is responsible for creating map tasks and reduce tasks. The main overhead of IncMR lies in the storage and transmission of many intermediate results. In the shuffle phase, the framework fetches the relevant partition of the output of all the mappers to each reduce task node via HTTP. The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged. Locality control is always used for optimization of job and task scheduling [25] [16].

Naive locality control. When there is only one reduce task and it is always located in the same node, the reduce task can fetch cached state from the local node directly without needing to fetch the prior map outputs from other nodes. Basically, it is the simplest to control the data flow when there is only one reduce task.

Complex locality control. If the number of reduce tasks is more than 1 and is not changed, job scheduler will try to allocate reduce tasks to the nodes that have performed reduce tasks recently because they have cached related state. What's more, the fetched prior results are also sorted. This locality control will save plenty of time for data transferring. If the number of reduce tasks or the position of reduce task is changed, state fetching and data repartition operations can not be avoided.

IV. CASE ANALYSIS AND ALGORITHM

Let us imagine the incremental Web log analysis application running in IncMR framework. When it is first submitted, the

input data are stored as blocks in HDFS. Job scheduler partitions the input into splits according to the related configuration and creates one map task for each split. When all the map tasks finish, reduce tasks begin to retrieve their inputs from each map task node. Note that the output paths of map tasks should be saved as the state data. Then the reduce tasks will produce final outputs which are stored in HDFS. When new log files arrive and an incremental job is submitted, the new log files are still stored in HDFS. The creation and execution of map tasks are almost the same as the initial one. However, when reduce tasks begin to retrieve their inputs, there are two kinds of data to be copied. One is the outputs of map tasks and the other is the state data.

Two scenarios need to be considered. First, there is only one reduce task for the job. When the reduce task begins to retrieve the input, it only needs to copy the state data directly and combines them with the current map outputs. Second, there are two or more reduce tasks. In this case, the state data may need to be repartitioned and forwarded to different reduce tasks.

Algorithm 1 describes the process of state storage which is managed by the job scheduler. After the end of an initial job or incremental job execution, the paths and contents of its map outputs will be recorded. In IncMR framework, the paths are stored in master node while the contents are stored in local nodes which run map tasks. In order to support scheduling based on locality, the information of reduce nodes is also recorded in the master node.

Algorithm 2 describes how the state data is repartitioned because of the changed reduce number. The process of repartition is similar to the partition process in map stage. Actually, in most cases the number of reduce will not be changed, so the state data does not need to be repartitioned. One special case is that the number of reduce is 1. In such case, the state data does not need to be repartitioned and will be forwarded to the only one reduce task. If the reduce number is unfortunately changed, the state data undoubtedly needs to be repartitioned. The repartition process can be presented as follows. First KV pairs, the intermediate results of the prior map tasks, are read from state data files and mapped to different partitions whose

Algorithm 1 State Storage

Input: *incJobList*: job list of initial run and incremental run
oldRNum: previous number of reduce tasks
newRNum: current number of reduce tasks
Output: *stateList*: state data for different reduce tasks

- 1: *stateList* $\leftarrow \emptyset$
- 2: *File aFile* $\leftarrow \emptyset$
- 3: **for all** *IncJob incJob* in *incJobList* **do**
- 4: **for all** *Output output* in *incJob.outputList* **do**
- 5: **if** *newRNum* > 1 **and** *oldRNum* \neq *newRNum* **then**
- 6: *aFile* \leftarrow *rePart(output.outPath)*
- 7: **else**
- 8: *aFile* \leftarrow *output.outPath*
- 9: **end if**
- 10: *stateList.add(aFile)*
- 11: **end for**
- 12: **end for**
- 13: **return** *stateList*

Algorithm 2 State Repartition

Input: *newRNum*: current number of reduce tasks
oldRNum: previous number of reduce tasks
oldStateFile: output of previous map tasks
Output: *newStateFile*: new file being repartitioned

- 1: **if** *newRNum* = 1 **or** *newRNum* = *oldRNum* **then**
- 2: *newStateFile* \leftarrow *oldStateFile*
- 3: **else**
- 4: *PKVPAIR pkv* $\leftarrow \emptyset$
- 5: **for all** *KVPAIR kv* in *oldStateFile* **do**
- 6: *pkv* \leftarrow *partition(kv.key, kv.value, newRNum)*
- 7: *collect(pkv, newStateFile)*
- 8: *sort(newStateFile)*
- 9: **end for**
- 10: *mergeParts(newStateFile)*
- 11: **end if**
- 12: **return** *newFile*

values are determined by reduce number. Then they are sorted and wrote to the local disk when the buffer is not enough. At last, the sorted spills are merged to one file which is repartitioned.

The copy of state can be shown as Algorithm 3. In order to improve the copy performance, if the state is stored in the local nodes of reduce tasks, it can be retrieved directly, otherwise, it needs to be copied from remote nodes by HTTP protocol.

V. EXPERIMENTAL EVALUATION

The current version of Hadoop from svn is not stable, so IncMR runtime is developed based on Hadoop-0.20.203.0. We built the experiment on hpc.ufl.edu which is the high performance center of University of Florida. Two classical parallel programs based on MapReduce model, WordCount and Grep, are adopted to test IncMR runtime. WordCount is a popular data-intensive application to determine the frequency

Algorithm 3 State Copy

Input: *stateFiles*: state files for different reduce tasks
Output: *stateLocal*: state file list in local reduce node

- 1: *stateLocal* $\leftarrow \emptyset$
- 2: **for all** *File aFile* in *stateFiles* **do**
- 3: **if** *isNotLocal(aFile)* **then**
- 4: *aFile* \leftarrow *copyToLocal(aFile)*
- 5: **end if**
- 6: *stateLocal.add(aFile.localPath())*
- 7: **end for**
- 8: **return** *stateLocal*

of words in a document while Grep extracts matching strings from text files and counts how many times they occur. The related experimental parameters are shown as Table III.

TABLE III
RELATED PARAMETERS CONFIGURED IN THE EXPERIMENT.

Parameters	Values
Number of nodes	one main node and four slave nodes
DFS block size	64M
Number of reduce tasks	2
Initial file size	WordCount: 256M Grep: 256M
Incremental ratio	50%

A. Measurements

Run-time is the key factor to assess the effectiveness of incremental processing. Here run-time refers to the execute time of a job and run-time speedup refers to the ratio of the run-time difference between IncMR and MapReduce to the execution time of MapReduce. Greater value of speedup shows higher efficiency of incremental computation.

Figure 4 and figure 5 show the results of WordCount and Grep running separately on Hadoop and IncMR runtime. From them, we can find that with the increase in the amount of new data, the acceleration becomes more apparent and the incremental processing model shows an obvious advantage.

B. Optimizations

To analyze the data locality and system optimization, the state processing overhead is evaluated in the experiment. Because the storage and access of the state data in the network take a direct impact on the execution speed, here state processing overhead is mainly reflected in fetching state, repartitioning prior map outputs, sorting and merging input in reduce nodes. If the number of reduce tasks is not changed, the state can be obtained from the local reduce node instead of all map nodes. In this case, state fetching and repartition can be ignored. However, the sort and merge phases still need some time.

In order to present the efficiency of optimization, we set only one reduce task. When the reduce task needs to retrieve

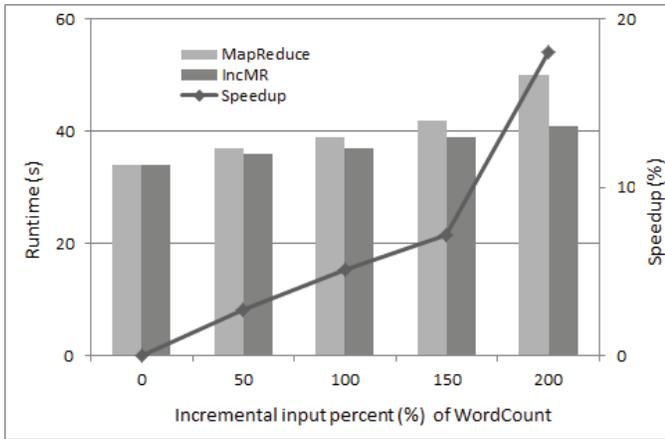


Fig. 4. Run-time speedup versus input size for WordCount application

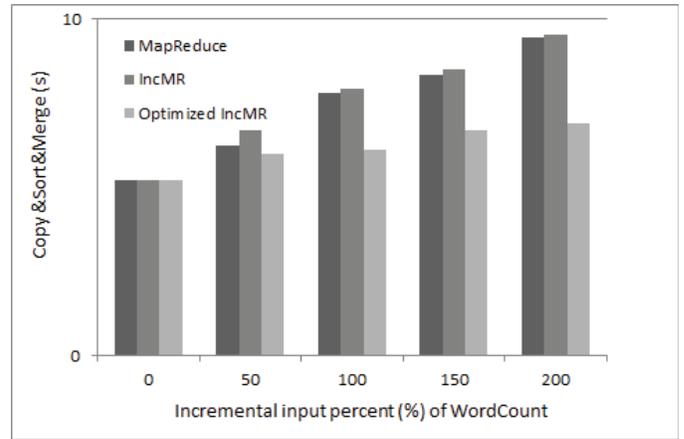


Fig. 6. State processing overhead versus different input size

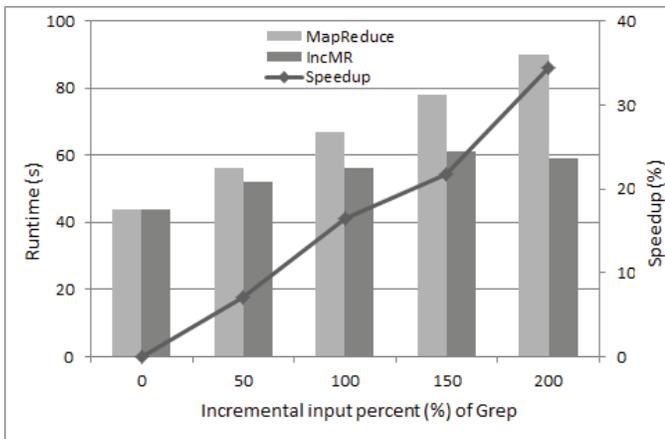


Fig. 5. Run-time speedup versus input size for Grep application

state, it can be found in the local node. So it is not necessary to copy from remote nodes.

Figure 6 shows the copy, sort and merge time of reduce task for WordCount application running on Hadoop, IncMR and optimized IncMR. From it, we can find that IncMR needs more time on copy, sort and merge phases because of the state processing while optimized IncMR shows better performance because only new map outputs need to be copied and the state can be obtained from the local nodes of reduce tasks. The incremental processing brings faster response for users and saves more computation resources for cloud providers.

VI. RELATED WORK

Iterative MapReduce. Iterative computing widely exists in data mining, text processing, graph processing, and other data-intensive and computing-intensive applications. MapReduce programming paradigm is designed initially for single step execution. Its high efficiency and simplicity attract a lot of enthusiasm of applying it in iterative environment and algorithms. HaLoop, a runtime based on Hadoop, supports iterative data analysis applications especially for large-scale data. By caching the related invariant data and reducers' local

outputs for one job, it can execute recursively [26]. Twister is a lightweight MapReduce runtime. It uses publish/subscribe messaging infrastructure for communication and supports iterative task execution [7]. In order to provide solutions for applications such as data mining or social network analysis with relational data, iMapReduce is designed to support automatically processing iterative tasks by reusing the prior task processors and eliminating the shuffle load of the static data [27]. Additionally, iterative computing is an indispensable part in incremental processing. We also apply related iterative computing methods in our IncMR framework.

Continuous MapReduce. Ad-hoc data processing is a critical paradigm for wide-scale data processing especially for unstructured data. Reference [1] implements an ad-hoc data processing abstraction in a distributed stream processor based on MapReduce programming model to support continuous inputs. MapReduce Online adopts pipelining technique within a job and between jobs, supports single-job and multi-job online aggregation, and also provides database continuous queries over data streams [28] [24]. Reference [29] combines MapReduce programming model with the continuous query model characterized by Cut-Rewind to process dynamic stream data chunk. CMR, continuous MapReduce, is an architecture for continuous and large-scale data analysis [30]. Continuous processing is a special case of incremental processing. Although the input is continuous, the processing is discrete. Time interval or delay is an important factor to be considered. IncMR presented in this paper supports continuous processing.

Incremental parallel data processing. Reference [20] presents a generalized architecture for continuous incremental bulk processing. It takes the prior state as an explicit input combined with the new input. A set of dataflow primitives is also provided to perform web analytics and mine large-scale and evolving graphs. Percolator is a system for incrementally data processing by using distributed transactions and notifications. It is mainly used to create Google Web search index [19]. Incoop is an extended MapReduce framework for incremental computations. It updates HDFS to incremental

HDFS, improves the scheduling performance, and provides a transparent solution for users [23]. These are general models for incremental applications. PIESVM is a special parallel incremental extreme SVM classifier. It is designed based on MapReduce model and can save training time for SVM algorithm [4]. General models are more common and suitable for incremental applications. Similarly IncMR takes the advantage of general models, continues to use HDFS, and adopts the state management approach to deal with the incremental problem.

Applications based on MapReduce. When programming in MapReduce framework, all we should do is to prepare the input data, implement the mapper, the reducer, and optionally, the combiner and the partitioner. The execution is handled transparently by the framework in clusters ranging from a single node to thousands of nodes with the data-set ranging from gigabytes to petabytes and different data structures including relational database [11], text [8], graph [10], video and audio [13]. Through a simple interface with two functions, map and reduce, MapReduce model facilitates parallel implementation of many real-world tasks such as data processing for search engines [9] and machine learning [12] [4]. MapReduce is now popularly used in scientific computing fields too. When the new input arrives, it is a challenge for applications to continuously deal with it based on the original MapReduce. This paper addresses the problem in general.

VII. CONCLUSION

This paper presents an incremental data processing model which is compatible with the MapReduce model and its runtime. It supports MapReduce-based applications without any modification. Future research will be focused on how to optimize the state management because the size and location of the state data have significant impact on the efficiency and network bandwidth. Our future research includes state acquisition, preservation, transmission, and updating. Another point is how to utilize the latest state to optimize the sort and merge process in the reduce phase.

REFERENCES

- [1] D. Logothetis and K. Yocum, "Ad-hoc data processing in the cloud," in *34th International Conference on Very Large Data Bases (VLDB)*, 2008.
- [2] G. DeMorales, A. Gionis, and M. Sozio, "Social content matching in mapreduce," in *37th International Conference on Very Large Data Bases (VLDB)*, 2011.
- [3] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *26th International Conference on Data Engineering (ICDE)*, 2010.
- [4] Q. He, C. Du, Q. Wang, F. Zhuang, and Z. Shi, "A parallel incremental extreme svm classifier," *Neurocomputing*, vol. 74, no. 16, pp. 2532–2540, 2011.
- [5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [6] "MapReduce Tutorial," url: <http://hadoop.apache.org/mapreduce/docs/r0.21.0/mapred-tutorial.html>.
- [7] J. Ekanakake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.
- [8] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.
- [9] M. D. E. T. W. L. Lin, J., "Of ivory and smurfs: Loxodontan mapreduce experiments for web search," in *18th Text Retrieval Conference (TREC)*, 2009.
- [10] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and C. G., "Pregel: a system for large-scale graph processing," in *International Conference on Management of data (SIGMOD)*, 2010.
- [11] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: Simplified relational data processing on large clusters," in *International Conference on Management of data (SIGMOD)*, 2007.
- [12] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-reduce for machine learning in multicore," in *21st Annual Conference on Neural Information Processing Systems (NIPS)*, 2007.
- [13] R. Pereira, M. Azambuja, K. Breitman, and M. Endler, "An architecture for distributed high performance video processing in the cloud," in *3rd International Conference on Cloud computing (Cloud)*, 2010.
- [14] R. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.
- [15] J. Talbot, R. Yoo, and C. Kozyrakis, "Phoenix++: Modular mapreduce for shared-memory systems," in *2nd International Workshop on MapReduce and its Applications (MapReduce)*, 2011.
- [16] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *8th USENIX Conference on Operating systems Design and Implementation (OSDI)*, 2008.
- [17] K. Tsoi and W. Luk, "Axel: A heterogeneous cluster with fpgas and gpus," in *18th International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010.
- [18] A. Dou and V. Kalogeraki, "Misco: A mapreduce framework for mobile system," in *International Conference on Pervasive Technologies Related to Assistive Environments (PETRA)*, 2010.
- [19] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [20] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *ACM Symposium on Cloud Computing (SOCC)*, 2010.
- [21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *2nd ACM European Conference on Computer Systems (EuroSys)*, 2007.
- [22] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
- [23] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquini, "Incoop: Mapreduce for incremental computations," in *ACM Symposium on Cloud Computing (SOCC)*, 2011.
- [24] T. Condie, N. Conway, P. Alvaro, J. Heelerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *7th USENIX conference on Networked systems design and implementation (NSDI)*, 2010.
- [25] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and S. I., "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *5th ACM European Conference on Computer Systems (EuroSys)*, 2010.
- [26] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "Haloop: efficient iterative data processing on large clusters," in *34th International Conference on Very Large Data Bases (VLDB)*, 2010.
- [27] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," in *1st International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2011.
- [28] T. Condie, "Online aggregation and continuous query support in mapreduce," in *7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.
- [29] Q. Chen and M. Hsu, "Continuous mapreduce for in-db stream analytics," in *International Conference on On the Move to Meaningful Internet Systems (OTM)*, 2010.
- [30] C. Trezzo, "Continuous mapreduce: An architecture for large-scale in-situ data processing," Master's thesis, University of California, San Diego of Computer Science, 2010.