A Lightweight Continuous Jobs Mechanism for MapReduce Frameworks

Trong-Tuan Vu, Fabrice Huet

A Lightweight Continuous Jobs Mechanism for MapReduce Frameworks

Trong-Tuan Vu
INRIA Lille Nord Europe, France
trong-tuan.vu@inria.fr

Fabrice Huet
INRIA-University of Nice, France
fabrice.huet@inria.fr

***Abstract*—MapReduce is a programming model which allows the processing of vast amounts of data in parallel, on a large number of machines. It is particularly well suited to static or slow changing set of data since the execution time of a job is usually high. However, in practice data-centers collect data at fast rates which makes it very difficult to maintain up-to-date results. To address this challenge, we propose in this paper a generic mechanism for dealing with dynamic data in MapReduce frameworks. Long-standing MapReduce jobs, called *continuous Jobs*, are *automatically* re-executed to process new incoming data at a minimum cost. We present a simple and clean API which integrates nicely with the standard MapReduce model. Furthermore, we describe cHadoop, an implementation of our approach based on Hadoop which does not require modifications to the source code of the original framework. Thus, cHadoop can quickly be ported to any new version of Hadoop. We evaluate our proposal with two standard MapReduce applications (WordCount and WordCount-N-Count), and one real world application (RDF Query) on real datasets. Our evaluations on clusters ranging from 5 to 40 nodes demonstrate the benefit of our approach in terms of execution time and ease of use.**

***Keywords*-publish/subscribe, continuous MapReduce.**

I. Introduction

A. Context and Motivation

Computing challenges from different scientific fields, e.g. climatic changes, bio-informatics, health sciences and engineering, require an enormous amount of resources to be solved in a reasonable time. Computing resources are more and more available in the form of clusters, grids and clouds. As a result, the trend of using these resources for large-scale computation is recently growing rapidly and it is expected to replace high-end servers because of its unlimited computation resources and cheap investment. This change focuses on the huge number of commodity machines working in parallel to perform intensive computations within a reasonable time. However, efficient use of these resources by providing the users with simple tools is still a challenging issue. New programming models should provide the simple tool through which the users can easily express complex distributed programs by hiding all underlying complexity. Among them, MapReduce [5] is one of the most popular programing framework in the distributed environments. The MapReduce framework is very attractive both commercially and academically due to its simplicity. Hadoop [9], an open source implementation of MapReduce has been widely used in many large companies and institutions, e.g. Yahoo!, Facebook, Amazon etc, for large-scale data analysis. The users can quickly develop a complicated distributed application using MapReduce without a comprehensive knowledge of parallel and distributed systems. Furthermore, many algorithms as well as applications naturally fall into the MapReduce model, such as word count, grep, equi-join queries etc. and other applications required large data analysis coming from other domains (biology, geography, physics, etc). Therefore, it is considered as an important platform for large-scale, massively parallel data computation.

The MapReduce framework is particularly well suited to static or slow changing sets because of the dependency betwen the map and the reduce phases. Some applications, however, work on dynamic data, either added to the initial input set or generated during the execution, and should generate updated results without human intervention. This decoupling of data production and consumption is widespread and gives birth to many different programming models. As an example, the Publish/Subscribe model allows asynchronous generation and processing of data and is often used in large dynamic systems.

There are two key problems while adapting the above iterative applications in standard MapReduce. The first problem is that MapReduce jobs implementing the iterative applications have to be *manually* resubmitted to the system at each iteration. In the current implementations of MapReduce, there is no mechanism allowing MapReduce jobs to be resubmitted *automatically* to the system whenever their input dataset increase. Lack of such mechanism makes maintenance of up-to-date results difficult, not to say impossible. The second problem is that even though most of the dataset are processed in the previous iterations, the data must be reloaded and re-processed at each iteration while only few new incoming data are added, inducing unnecessary overhead.

B. Contributions

There are several research works based on MapReduce frameworks,which try to improve the performance of the original MapReduce or extend its applicability to more types of applications. However to the best of our knowledge, there has not been any research work providing a low level API for writing continuous MapReduce jobs.

From the framework side, we propose a modified MapReduce architecture, called *continuous MapReduce*, which sup-

ports continuous jobs. In our framework, a continuous job, which is submitted to the system, runs implicitly over a long time span, and keeps notifying new results to the users as each increment of the data occurs. In other words, after being submitted to the system, the continuous job will be invoked by new incoming data and will process them. To validate our approach, we develop *cHadoop*, a continuous version of Hadoop which is designed to efficiency handle the above type of applications. Our implementation is developed on top of the original Hadoop implementation without any modification of the original source code so that it can be easily ported to new versions of Hadoop.

To summarize, this paper makes the following contributions:

- a continuous job framework adapted to the MapReduce programming model.
- an extension to standard MapReduce jobs to deal with the complexity of data management for continuous jobs.
- an implementation, called cHadoop, which supports standard and continuous jobs and requires no direct modification of Hadoop source code
- an evaluation of the performance with two standard MapReduce applications (WordCount, WordCount-N-Count) and a realistic application (RDF queries).

The remaining of the paper is organized as follows. Section II presents the background of MapReduce and some most related papers to our research work. Section III is dedicated to a high level presentation of our approach and its main components. Section IV describes the implementation of our approach on top of Hadoop. Section V presents some case studies of the framework in detail. In the next section, Section VI, the results of our experiments are reported with some analysis of the performance of our approach. Finally, we conclude the paper and discuss some open issues.

II. Background and Related works

A. Background

1) MapReduce: Processing large amount of data often requires a lot of computations which are distributed on a large set of machines.

Google has proposed the MapReduce programming model [5] that allows the users to develop complex distributed programs but conceals all the underlying sophistications. The most attractive feature of MapReduce is its simplicity: a MapReduce program is comprised of only two primitive functions, called *map* and *reduce*, which are written by users to process key/value pairs.

First, a user-defined map function reads a set of records from an input file, then performs any desired operation, producing a set of intermediate key-value pairs. All pairs with the same key are grouped together and passed to the same $reduce$ function. The $reduce$ function process all values associated to a particular key and produces key-value pairs. Generally, all map/reduce instances are distributed on different nodes of a cluster.

Algorithm 1 The Standard MapReduce WordCount Code

```
function MAP(key, value)
    EmitIntermediate (value, 1)
end function

function REDUCE(key, values[])
    for v in values do
        result ← result + v
    end for
    EmitOutput (key, result)
end function
```

For instance, as presented in Algorithm 1, we consider the *wordcount* example which is directly taken from the original MapReduce article [5]. Each map function takes names of input documents as keys and their contents as values. Then it emits intermediate key-value pairs for every word in the document. Each pair contains a word as key, and an associated count of its occurrences as value, i.e. 1. In the reduce function, all intermediate key-value pairs are grouped based on the intermediate key, thus in this example, every word is associated with its occurrence values in the document. Finally, each reduce function sums together all word counts emitted for a specific word and writes the final output as a new file to the distributed file system.

2) Hadoop: Hadoop [9] is the Apache Software Foundation open source and Java-based implementation of the MapReduce framework. It provides a tool for processing a vast amount of data using the MapReduce model. Data are processed in-parallel on large clusters in a reliable and fault-tolerant manner.

The Hadoop framework contains two main components: Hadoop Distributed FileSystem (HDFS) and Hadoop MapReduce. HDFS is an open source implementation of the Distributed Google File System (GFS) [10]. It provides a unified storage by aggregating disk space from different machines called *DataNodes*. It supports standard operations such as reading and writing, and provides redundancy through block replication. It does not however, support modification of existing files; appending data to files is still experimental and should not be used for production work. When a file is copied to HDFS, it is divided into 64MB blocks which are randomly distributed on the data nodes. Although HDFS is distributed, it still has a single point of access, called *NameNode*, which maintains meta-data for the whole filesystem.

The Hadoop MapReduce distribution comprises the infrastructure to execute MapReduce Jobs. A master node, called *JobTracker* controls the distribution and execution of

the jobs on the computation nodes (*TaskTrackers*) across the cluster.

B. Related Works

Hadoop Online Prototype (HOP) [4], is an approach for getting "early results" from a job while it is being executed by a flush API. The flush API forces the job's mappers to send their current results to the job's reducers, and so it can return a snapshot of a final output to the users by processing the intermediate results. HOP can be considered as a potential approach for running MapReduce jobs continuously, however the authors only consider it in terms of returning snapshots of the final output to the users. There is no guarantee that the snapshots represent a possibly correct output of the job. It is only adapted to jobs where the output converge toward the final one. Recently, several research works have been studied to implement iterative algorithms efficiently on MapReduce frameworks such as HaLoop [2], Twister [6] or PIC [8]. The common characteristic of iterative applications is that they operate on both the original input data set (often called static data) and a generated one (called dynamic or variable data). These data sets are processed iteratively until the computation satisfies a convergence or termination condition. The distinction between the two types of data can be used to avoid repeated loading of the static data and improve performance.

Therefore, [2], [6], [8] propose some techniques to improve the performance by avoiding the repeated loading of unchanged original input data at each iteration. In our opinion, the current techniques could not be applied to our problem due to following limitation. The original input of their iterative applications is always static and does not change during the computation. As a consequence, there is no trigger mechanism to automatically re-launch the computation as input data sets evolve.

A low latency continuous MapReduce model is described in [1]. It is designed to quickly and efficiently process incoming data but does not seem to be able to maintain states between successive executions. At a higher level, [12] describes a workflow manager which pushes new data to Pig programs running on top of Hadoop. In their framework, the *de-duping* operation allows a task to emit data to be used for subsequent executions. Finally, the work in [3] is probably the closest to our proposal. The authors propose to save intermediate states of complex queries (SQL) in buffers using user defined functions. However neither the semantic nor the integration of such functions in a MapReduce workflow is clearly described.

Although these frameworks bear some close similarities to our proposal, there are some notable differences:

- The existing iterative frameworks (e.g. [2], [6], [8]) assume that the only new data added to the system comes from previous iterations and not from outside source.
- Uninterrupted execution until the end of the computation is assumed. Thus, they lack mechanism to automatically re-launch jobs when input data sets evolve.
- The implementation of current approaches (e.g. [2], [4], [8]) extensively modify the original Hadoop which makes it difficult to keep-up with the latest official release.
- The higher level frameworks (e.g. [3]) are not directly applicable to standard MapReduce jobs since they usually rely on specific languages or external APIs.

To the best of our knowledge, our proposal is the only one to work at the level of a MapReduce job and offer a simple and elegant solution to maintain state between executions. Because it does not rely on extensive modification of Hadoop, it can also be easily ported on new versions of Hadoop.

III. Continuous MapReduce jobs

We will call *continuous jobs* as a set of jobs which have to be executed *automatically* when new input data are available in a specified location of the filesystem. A continuous job should have the following properties:

- *efficiency*: it should not process the whole dataset at each iteration.
- *correctness:* the merging of all results should be equivalent to those obtained if running on the whole dataset.

A naive approach to implement continuous jobs is to simply re-execute jobs on new data added after the last execution, using timestamps. However, the main issue with this algorithm is that it does not have the *correctness* property. To clearly understand the above properties, consider a continuous version of the *wordcount* application. Let's assume that at time t_0, it runs on the dataset which only contains the word *foo* once. At time t_1, some new data is added which also contains *foo*. If the job only runs on the new data, it will report *foo* as appearing once although there are two occurrences in the whole dataset.

In order to hold the above properties, continuous jobs need to have the following features. The first one is to keep the *efficiency* property by taking only new data into consideration for the new re-execution, instead of the whole dataset. The second one is to hold the *correctness* property by saving some of the current data or results for the next execution. It is worth noting that, depending on the application, only a subset of the results should be carried to the next execution.

From our analysis, data can be classified into three different categories.

- *New data* have been added to the system since the last execution of a continuous job.
- *Result data* have been produced by a continuous job.
- *Carried data* have to be saved for subsequent runs.

The notion of *carried data* is application dependent and should be decided by the programmer. In our previous

example, the carried data for *word-count* would be the result of the previous run, as we will explain in detail in section IV-D. A natural place to decide which data should be carried is the reduce phase because it is usually the place producing the final results of the job. Therefore, the output of the *carry* and *reduce* operations are expected to be similar: both should produce $(key, value)$ pairs. We propose a modification to the definition of the MapReduce job which adds a *carry* operation during the *reduce* phase, as shown in Figure 1. The data generated by this method will be re-injected as an output of the map phase of the next execution. The rational is that they were generated by the previous reduce phase from "mapped" data. Hence, having them go through a mapper again might just be redundant. In practice, the developer simply calls a *carry* function to output $(key, value)$ pairs from inside the code of the *reduce* function.

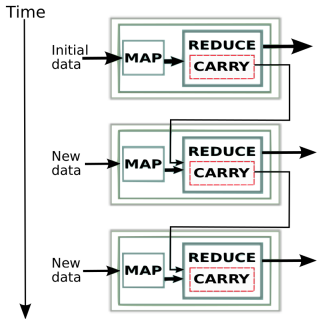

Figure 1: Execution Flow of Continuous Job

The new execution flow of the continuous job is described in Figure 1. During the first execution, it only scans existing data. If a *carry* method is defined, *carried data* will be produced and added to the input of reducers of the next execution. When new data are added, the job is executed against them and the carried one.

IV. Implementation

In this section, we discuss in detail the implementation of our continuous job mechanism in cHadoop, a continuous version of Hadoop. We have to address three different points. First, we need to store jobs submitted by a user so that they can be re-executed when necessary. Second, we need the filesystem to trigger the re-execution of a job whenever new data is added. Finally, we need to limit the continuous jobs to new data by using the timestamp mechanism. Therefore, we introduce two main components, a *Continuous Job Tracker*, a *Continuous NameNode*, which are central to managing and executing continuous jobs. To help users develop continuous jobs, we provide a *ContinuousJob* and the associated *ContinuousMapper* and *ContinuousReducer*. A global view of cHadoop is given in Figure 2.

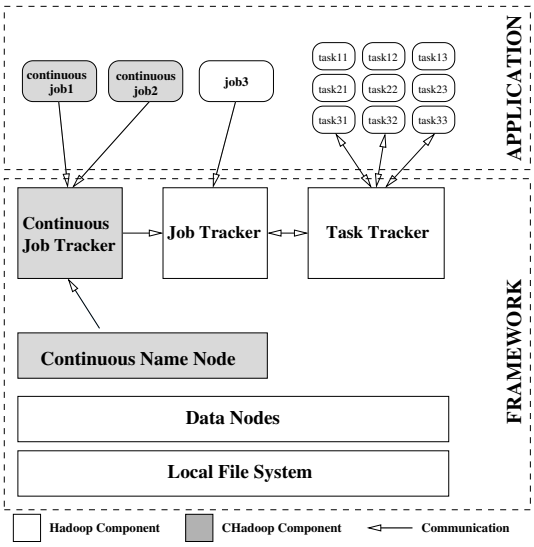

Figure 2: The cHadoop framework

A. Continuous Job Tracker

As explained in section II-A2, the *Job Tracker* is responsible for managing the users' submitted jobs and scheduling them on the cluster. Once they are finished, they are discarded and cannot be run again unless they are *manually* resubmitted. We introduce a new component, the *Continuous Job Tracker*, which acts as a proxy to the standard one. Therefore, instead of submitting continuous jobs to the *Job Tracker*, they are submitted to the *Continuous Job Tracker* which will resubmit them when necessary. The whole communication for submitting jobs to the *Continuous Job Tracker* or resubmitting them to the *Job Tracker* are transparently performed by the framework. They are implemented using public Hadoop APIs, making them easier to port to new versions.

As in the original Hadoop, non continuous jobs can be directly submitted to the *Job Tracker*. By keeping all the features provided in Hadoop, cHadoop not only brings new features for enabling continuous jobs but also supports standard jobs and tools such as the web frontend.

B. Continuous Name Node

In Hadoop, the *Name Node* is responsible for file system management and acts as a single entry point to the distributed filesystem. In the latest stable Hadoop release available at the time of writing [1] , there was no public API to listen to filesystem events such as file creation. Thus we could not avoid replacing the *NameNode* with

[1]Hadoop 1.0.4

our own version to have a *ContinuousNameNode* which triggers events for changes in the filesystem. Our version simply subclasses the *NameNode* class and overrides the required methods. When new data are added, an event will be sent to the *Continuous Job Tracker*. If there are any continuous jobs monitoring the path storing the new data, they will be resubmitted to the cluster. The communication is implemented using the RPC protocol provided by the Hadoop IPC/RPC API.

Furthermore, the *Continuous Name Node* also plays an important role in data management, which strongly impact the performance of our framework.

C. Continuous Data Management

In order to limit a continuous job to new data, we chose to timestamp the data when they are added to the system. In Hadoop or other MapReduce implementations, data are split in fixed size blocks which are distributed among data nodes. Appending data to existing files is at best experimental and no in-place modification are allowed. Thus the timestamping can be done at the block level. More precisely, when data are copied to the filesystem, all of their data blocks are time-stamped as follows: $< timestamp, block\_data_1, block\_data_2, ..., block\_data_n >$. Therefore, the overhead is expected to be negligible both in time and space.

Only blocks added to the filesystem after the last execution time of a continuous job are considered valid and will be selected as an input for the next execution. Other blocks are discarded as they do not contain any new data.

To implement such mechanism, we use the standard *InputFormat* which breaks the input of a job into *InputSplits*, i.e data blocks to be processed by mappers. We provide a *ContinuousInputFormat* which only passes valid splits to mappers using Algorithm 2.

Algorithm 2 Block Selection for Continuous Job j

$TS_j \leftarrow$ get timestamp of job j
$T_j \leftarrow$ get all data blocks of a given input path of job j
$V_j \leftarrow \emptyset$
for each block b in T_j **do**
  $time\_stamp \leftarrow$ get timestamp value of b
  if $time\_stamp \geq TS_j$ **then**
    $V_j \leftarrow V_j$ + b
  end if
end for
return V_j to job j

D. API example

The design choices made in the implementation make the code of a continuous job very similar to a standard one. To write a continuous job, we provide "continuous" versions of the required Hadoop classes (Job, Mapper, Reducer, ...) through sub-classing. In most cases, simply changing the name of the classes and the corresponding imports will be the only necessary step. In Hadoop, it is standard practice to not access the filesystem directly but rather rely on the *Context* object provided at runtime to write data at the end of the reduce phase. We provide a *ContinuousContext* which exposes a *carry* method to save data for subsequent executions. A slightly simplified Java version of the reducer of the continuous *Word-Count* is shown in Listing 1.

```
class WordCountReducer extends ContinuousReducer {
    ...
  void continuousReduce(..., ContinuousContext context){
    int sum = 0;
    for(IntWritable i : values) {
        sum = sum + i.get();
    }
    context.write(key, sum);
    context.carry(key, sum);
  }}
```

Listing 1: Java code for the reducer of continuous *Word-Count* in cHadoop

V. Case Studies

A. WordCount and WordCount-N-Count

The first application, WordCount is simply a continuous version of the famous WordCount job. WordCount-N-Count, on the other hand, reports words which appear at least N times in the input file. For WordCount, we simply need to carry the current result for the next execution. Hence the carried data are the same as those normally output by the reducer. For WordCount-N-Count, only words which appear less than N times need to be carried because new data might change their number of occurences (Algorithm 3).

Algorithm 3 Reduce function of WordCount-N-Count in cHadoop

function REDUCE(key,$values$[])
  for v **in** $values$ **do**
    result $\leftarrow$ **result + v**
  end for
  if result $\geq N$ **then**
    EmitOutput (key, result)
  else
    Carry (key, result)
  end if
end function

B. RDF Query

To further validate our proposal, we now consider the problem of distributed RDF queries [13] on large data

sets. RDF is a data format extensively used in semantic web technologies. Data are represented as triple in the form $subject, predicate, object$. A dedicated language, SPARQL, is used to perform complex queries over them. A query is usually made of a set of triple patterns and some filtering and returns only the triples matching all patterns and filters. We have implemented the query processing in Hadoop using the algorithm described in [11]. It is based on two phases: the selection phase and the join phase. The selection phase filters the RDF data that satisfy at least one triple pattern and the join phase runs iteratively based on the number of join variables in the Basic Graph Pattern of the query [11].

```
SELECT ?yr
WHERE {
?journal rdf:type bench:Journal.
?journal dc:title "Journal 1 (1940)"^^xsd:string.
?journal dcterms:issued ?yr
}
```

Listing 2: Query Q_1 of SP^2Bench

Figure 2 presents the query Q_1 of the benchmark SP^2Bench [14] which will be considered in our experiments. It consists in three sub-queries joined by a shared join variable $journal$. Basically this query returns the issued year ($dcterms : issued$) of a publication of type $bench : journal$ with title *Journal 1 (1940)*. In cHadoop, this query is implemented as two continuous MapReduce jobs. The first one, called *Selection Job*, takes RDF triples as input and outputs only those matching at least one of the subqueries. The second job performs a *Join* on the $journal$ variable to identify, among the candidate triples, those which match the whole query. The non matching triples of the join job are carried for subsequent executions. The overall execution flow of the query is presented in Figure 3.

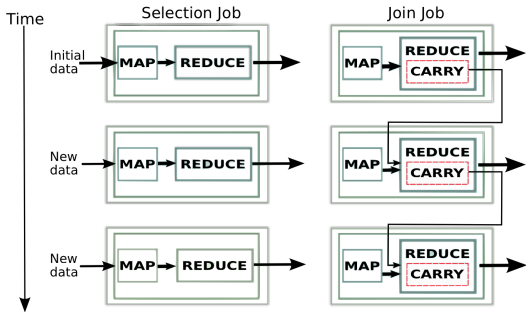

Figure 3: Execution flow of query Q_1 in cHadoop

VI. Experiment Results

In this section, we report the results of executing three applications using cHadoop on two clusters of different sizes. All results are compared to a similar application implemented using standard Hadoop.

A. Experimental Setup

We have run our experiments on two clusters, namely *small* and *large* to demonstrate the use of the cHadoop framework and its benefits at different scales. The small testbed is a 5 nodes cluster and the large one a 40 nodes cluster. Each node is equipped with two Intel 2.26Ghz processors (4cores/processor), 32GB of RAM and 510GB of storage space. The nodes are connected in a mesh topology with 10 gigabit Ethernet links. The framework was configured as follows: the masters (*ContinuousJobTracker*, *JobTracker* and *ContinuousNameNode*) were running on the same physical machine, and the clients, i.e *TaskTracker* and *DataNode*, were distributed among the 5 and 40 physical machines in the small and large testbed, respectively. From Hadoop point of view, the small (resp. large) cluster is configured to have a total 40 map and 20 reduce slots (resp. 320 map and 160 reduce slots). Finally, Hadoop 1.0.4 and Java 1.7 are used in our experiments.

The scenario for all experiments is as follows. An application, implemented as a continuous job, is submitted to the system which initially contains no data. More precisely, the application takes (subscribes to) an empty folder in the file system as its input. We then add new data to this folder which triggers the execution. This step is repeated for some iterations to increase the total amount of data in the filesystem. For the standard approach, after inserting a new dataset, the jobs need to be resubmitted *manually*.

For the applications WordCount and WordCount-N-Count, we use the benchmark [7]. At each iteration, 20GB of data, containing approximately 45 million words, are inserted. For the RDF query experiment, we use SP^{2}Bench [14] and add 3GB of data containing 20 million triples at each iteration. In all experiments, the number of reducers was set to the maximal configuration of the Hadoop cluster.

For all experiments we report the total execution time and the speedup achieved by cHadoop over standard Hadoop.

B. Experiment Results

Figure 4 shows the execution time and the speedup experienced on the small cluster for our three test applications. For the first iteration, there is no significant difference between standard Hadoop and cHadoop, showing that the overhead introduced by our framework is negligible. However, as the dataset keeps increasing, our approach starts to outperform the original one because each subsequent execution processes only a small fraction of the data (the new and carried one). The speedup of our approach increases with the number of iteration because its execution times only slowly increases with the amount of data.

Using the same dataset, we have ran the experiments on the large cluster (Figure 5). In this situation, the large

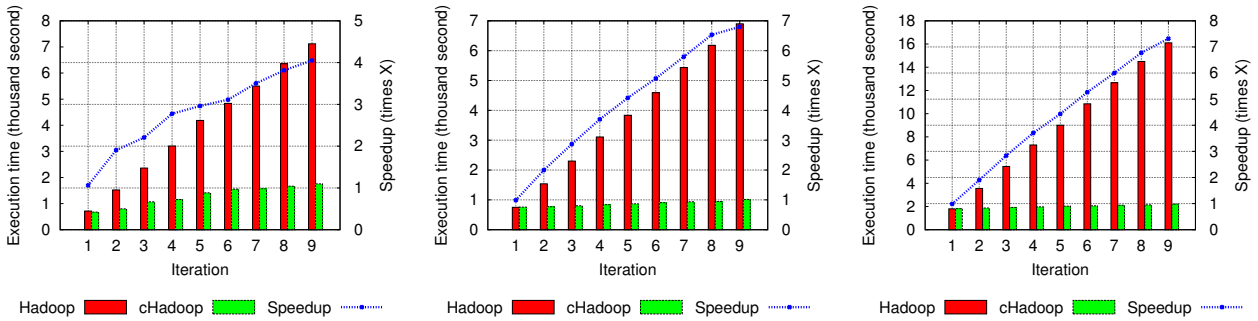

Figure 4: Experiments on small cluster. **Left**: WordCount. **Middle**: WordCount-N-Count, N = 20. **Right**: RDF Query.

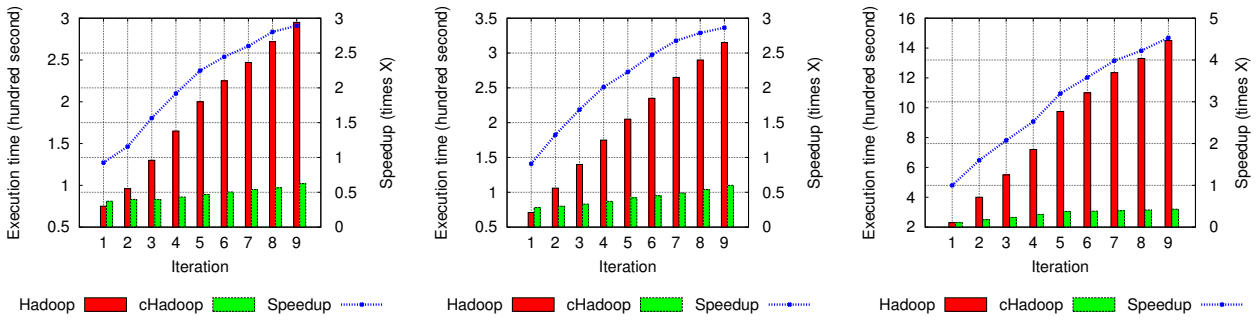

Figure 5: Experiments on large cluster **Left**: WordCount. **Middle**: WordCount-N-Count, N = 20. **Right**: RDF Query.

number of computing resources greatly reduces the amount of data to be processed on each node, hence lowering the impact of the growing dataset. Indeed for WordCount and WordCount-N-Count, we only achieve half the speedup of the small cluster. The RDF experiment maintains a better speedup because of the two phases: the join phase does not manipulate a large amount of data but still takes significant execution time because of overhead in Hadoop. This experiment shows that having continuous jobs on small dataset, compared to the cluster size, still provides significant performance improvements.

The obtained speedup can be explained by the much lower number of mappers started, as shown in Figure 6. In Hadoop, the number of mappers is dependent on the block size of the distributed filesystem and hence, the data input size. For the original job, the number of mappers increases with the size of the whole dataset since it simply takes the whole data set as its input. In contrast, the continuous job only needs to process new and carried data, greatly reducing the overhead in computation during its re-execution. More precisely, for example, in the WordCount application, carried data are increased by 0.6 GB on average by each iteration. However, since they have already been processed, they are in a much more compact form and requires less resources when re-executing. Similarly, in WordCount-N-Count, the size of carried data is about 1.7 GB after each iteration. This is an important benefit from our approach: the number of computing resources needed for executing continuous jobs is reduced compared to a standard job. This allows for better sharing of the Hadoop cluster among many users.

VII. Conclusion and Future Works

In this paper, we have presented a framework to support continuous MapReduce applications frameworks. Jobs registered by the user are automatically re-executed when new data is added to the system. Since running the job only on new data can lead to incorrect results so we have introduced the notion of carried data. These data are produced by a *carry* function in the reduce phase of continuous job and are automatically added as an input for the subsequent run. We have provided an implementation in Hadoop which does not require modifications to the original framework and is thus easily ported to new versions of the library. New data are identified using a timestamping mechanism, totally transparent to the user. The provided API is very close to the standard one, making it very easy to turn existing jobs into continuous one. Using two standard MapReduce applications and a non-trivial example of SPARQL queries, we have highlighted the good performance and the low overhead of our implementation. A limitation of the current implementation is the latency of the job re-execution. Since we strictly follow the standard Hadoop workflow, restarting a job on our cluster can take up to 20 seconds, which is very inefficient especially when the set of new and carried

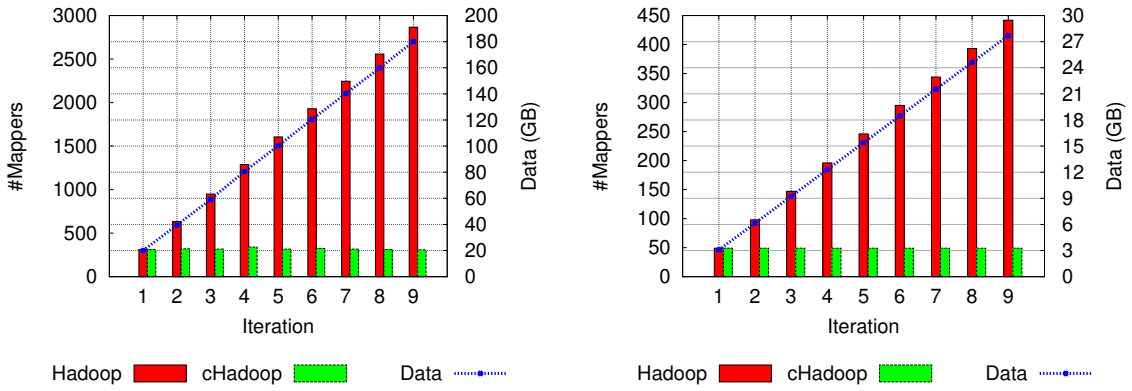

Figure 6: Impact of Continuous Data Management. **Left**: Number of mappers in both WordCount and WordCount-N-Count. **Right**: Number of mappers of the Selection Job in RDF Query.

data is small. We plan to investigate this in future work.

VIII. Acknowledgement

The authors would like to thank Clément Agarini, Anthony Damiano, Ludwig Poggi and Justine Rochas for their work on the implementation. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] N. Backman, K. Pattabiraman, and U. Cetintemel. C-mr: A continuous-mapreduce processing model for low-latency stream processing on multi-core architectures. Technical Report Technical Report CS-10-01, Department of Computer Science, Brown University, 2010.

[2] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, September 2010.

[3] Qiming Chen and Meichun Hsu. Continuous mapreduce for in-db stream analytics. In *Proceedings of the 2010 International Conference on On The Move to Meaningful Internet Systems*, OTM, pages 16–34. Springer-Verlag, 2010.

[4] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 21–21. USENIX Association, 2010.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, 2004.

[6] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC, pages 810–818. ACM, 2010.

[7] Mithuna Thottethodi Faraz Ahmad, Seyong Lee and T.N. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. Technical report, Purdue University, 2012.

[8] Reza Farivar, Anand Raghunathan, Srimat Chakradhar, Harshit Kharbanda, and Roy H. Campbell. Pic: Partitioned iterative convergence for clusters. In *Proceedings of the 2012 International Conference on Cluster Computing*, CLUSTER, pages 391–401. IEEE, 2012.

[9] Apache Software Foundation. Apache hadoop.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Operating System Review*, 37:29–43, 2003.

[11] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. Sparql basic graph pattern processing with iterative mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC, pages 6:1–6:6. ACM, 2010.

[12] 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 International Conference on Management of Data*, SIGMOD, pages 1081–1090. ACM, 2011.

[13] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf (working draft). Technical report, W3C, 2007.

[14] Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. Sp^2bench: A sparql performance benchmark. In *Semantic Web Information Management*, pages 371–393. 2009.