A New Efficient Resource Management Framework for Iterative MapReduce Processing in Large-Scale Data Analysis

Seungtae HONG\(^{a}\), Kyongseok PARK\(^{b}\), Chae-Deok LIM\(^{c}\), Nonmembers, and Jae-Woo CHANG\(^{d}\), Member

SUMMARY To analyze large-scale data efficiently, studies on Hadoop, one of the most popular MapReduce frameworks, have been actively done. Meanwhile, most of the large-scale data analysis applications, e.g., data clustering, are required to do the same map and reduce functions repeatedly. However, Hadoop cannot provide an optimal performance for iterative MapReduce jobs because it derives a result by doing one phase of map and reduce functions. To solve the problems, in this paper, we propose a new efficient resource management framework for iterative MapReduce processing in large-scale data analysis. For this, we first design an iterative job state-machine for managing the iterative MapReduce jobs. Secondly, we propose an invariant data caching mechanism for reducing the I/O costs of data accesses. Thirdly, we propose an iterative resource management technique for efficiently managing the resources of a Hadoop cluster. Fourthly, we devise a stop condition check mechanism for preventing unnecessary computation. Finally, we show the performance superiority of the proposed framework by comparing it with the existing frameworks.

key words: large-scale data analysis, iterative data processing framework, MapReduce, Hadoop

1. Introduction

Recently, researches on large-scale data have been actively done with the evolution of information technology. Large-scale data means a massive volume of complex data which cannot be dealt with the traditional data processing techniques\([1]\). In order to analyze large-scale data efficiently, MapReduce which was proposed in 2004 by Google\([2]\) has attracted much interest. MapReduce is a software framework that allows developers to write programs for processing big data in a distributed computing environment. MapReduce processes the data by utilizing map and reduce functions being commonly used in functional programming. Currently, Hadoop\([3]\), one of the most popular MapReduce frameworks, is widely utilized in the various fields of real world application\([4]\)–\([7]\).

Recently, there have been various iterative applications, such as scientific applications and geosocial data clustering\([8]\)–\([12]\). However, Hadoop has a critical problem that it has low performance on iterative processing which is essential for large-scale data analysis applications. To solve the problem, the Hadoop-based iterative data processing frameworks were proposed, such as HaLoop\([15]\),\([25]\), iMapReduce\([16]\), and iHadoop\([17]\). First, HaLoop dramatically improves the efficiency of iterative MapReduce job processing by using its loop-aware task scheduler and various caching mechanisms. Second, iMapReduce is a distributed computing framework for iterative computations, which allows users to specify the iterative computation with the separated map and reduce functions. Finally, iHadoop can process iterative applications by modifying the task scheduling of the traditional MapReduce model to efficiently support the iterative computations.

However, the existing iterative processing frameworks have the following drawbacks. First, because the existing frameworks are implemented based on the version v0.20 of Hadoop, they do not support the efficient resource management of cluster. Second, because the existing frameworks do not consider the characteristic of iterative applications, they do not provide an invariant data caching mechanism efficiently. Thirdly, since the existing frameworks do not consider the entire resources of cluster, the tasks for processing MapReduce job can be skewed towards particular nodes. Finally, they do not provide a stop condition check mechanism for preventing unnecessary computations.

To solve the problems, in this paper, we propose a new efficient resource management framework for iterative MapReduce processing in large-scale data analysis. First, we propose an iterative job state-machine for managing the iterative MapReduce jobs. The iterative job state-machine provides the stand-alone one which can minimize the internal changes due to the version update of Hadoop. Secondly, we propose an invariant data caching mechanism for reducing the I/O costs. The invariant data caching mechanism provides two caching mechanisms according to the types of applications, i.e., map input cache and reduce input cache, so as to reduce the I/O cost of loading and shuffling invariant data in the subsequent iterations. Thirdly, we devise an iterative resource management technique which can allocate resource uniformly to every node in a Hadoop cluster. For this, we store iteration information into a meta-data table in HDFS (Hadoop Distributed File System)\([13]\). Fourthly, we devise a stop condition check mechanism for preventing the unnecessary computation by comparing the current iteration output with the previous iteration output. Finally, we show
the performance analysis by comparing the proposed framework with the existing frameworks. Our contributions can be summarized as follows:

- We propose a new efficient resource management framework for iterative MapReduce processing in large-scale data analysis.
- We design an iterative job state-machine for managing the iterative MapReduce jobs, thus minimizing changes due to Hadoop version updates.
- We propose an invariant data caching mechanism for reducing the I/O costs of loading and shuffling invariant data in the subsequent iterations.
- We devise an iterative resource management technique for allocating resources uniformly to every node in a Hadoop cluster and a stop condition check mechanism for preventing unnecessary computations.
- We evaluate the efficiency of our iterative data processing framework through our extensive experiments.

The rest of this paper is organized as follows. In Sect. 2, we introduce the existing iterative MapReduce processing frameworks. In Sect. 3, we propose a new iterative data processing framework. In Sect. 4, we provide the performance analysis of our framework. Finally, we conclude this paper with future work in Sect. 5.

2. Related Work

Hadoop, an open-source project developed by Apache group, is the most popular framework for processing big data. In MapReduce, the input computation is a list of <key, value> pairs and each map function produces intermediate <key, value> pairs. Hadoop groups the intermediate <key, value> pairs into the buckets of reduce tasks based on a hashing function. The reduce tasks take both an intermediate key and a list of values as input and produce zero or more output <key, value> pairs. Because Hadoop can support good scalability and fault tolerance, it is used as a platform for analyzing big data by many companies and researchers. There are various Hadoop MapReduce applications. Table 1 classifies typical MapReduce applications; non-iterative one and iterative one.

Most non-iterative applications derive a result by doing one phase of map and reduce functions. There are three known non-iterative applications: join, distinct value counting and matrix multiplication. i) The join merges the two or more columns. ii) The distinct value counting counts the total number of records including unique values. iii) The matrix multiplication processes the massive matrix computations by using distributed processing.

On the other hand, iterative applications derive a result by doing the same map and reduce functions repeatedly. There are four known iterative applications: descendant query, k-means clustering, message passing, and page rank algorithm. i) The descendant query retrieves the neighboring nodes from the given input node by increasing the number of depths in the hierarchy. ii) The k-means clustering generates clusters repeatedly based on the mean value of items until the centroid of cluster is not changed. iii) The page rank algorithm iteratively calculates a rank of each node based on the rank of its neighboring nodes.

2.1 Iterative Data Processing Frameworks

The representative Hadoop-based iterative data processing frameworks are HaLoop [15], [25], iMapReduce [16], and iHadoop [17]. First, HaLoop is an enhanced version of Hadoop framework. HaLoop dramatically improves the efficiency of iterative MapReduce job processing by using its loop-aware task scheduler and various caching mechanisms. HaLoop provides not only a loop control method, but also a programming interface for the iterative data analysis. In order to reuse the invariant input data, the task scheduler of HaLoop assigns tasks into the same physical node which stores the data. To cache the invariant input data, HaLoop determines whether or not each input data is changed after the first iteration of a job. By using the cached data, HaLoop can reduce the I/O costs being required for loading and shuffling the input data in all iterations.

Secondly, iMapReduce is a distributed computing framework for iterative data analysis. iMapReduce allows users to specify the iterative computation with the separated map and reduce functions. It can support an automatic iterative processing within a single job. iMapReduce provides an efficient iterative processing framework while reserving the similar MapReduce programming interfaces. To reduce the initialization overhead for each task/job, iMapReduce devised the concept of persistent tasks to avoid repeated task scheduling. Because the input data is loaded into local file system only once, instead of being shuffled among workers many times, iMapReduce can significantly reduce the I/O cost and the network communication overhead. To break a synchronization barrier between MapReduce jobs, it facilitates asynchronous execution of map tasks within the same iteration. As a result, iMapReduce significantly improves the performance of iterative processing by reducing the overhead of creating new MapReduce jobs repeatedly, thus eliminating the shuffling of static data and allowing the asynchronous execution of map tasks.

Finally, iHadoop is a modified version of the MapReduce framework to process iterative applications. iHadoop modifies the task scheduling of the traditional MapReduce model to efficiently support the iterative computations. iHadoop provides better performance by executing iterations asynchronously such that an iteration starts before its preceding iteration finishes. Due to its asynchronous iter-

<table>
<thead>
<tr>
<th>Classification</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Iterative</td>
<td>Join, Matrix Multiplication, Word Count, Distinct Values, Matrix Multiplication</td>
</tr>
<tr>
<td>Iterative</td>
<td>PageRank, Descendant query, k-Means</td>
</tr>
</tbody>
</table>

Table 1 Classification of Hadoop MapReduce applications.
ations, the task scheduler of iHadoop can decide when to schedule a task.

2.2 Stand-Alone Iterative Processing Frameworks

Twister[18] is a distributed in-memory MapReduce framework optimized for iterative MapReduce computations. It uses a publish/subscribe messaging infrastructure to communicate and transfer data. Twister can support long running map/reduce tasks, which can be used in “configure once and use many times” approach. In addition, it manages the invariant data for supporting efficient iterative MapReduce computations and provides programming extensions to MapReduce with the data transfers of broadcast and scatter types. These improvements allow Twister to support iterative MapReduce computations more efficiently, compared to other MapReduce frameworks.

Spark[19] is an open source cluster computing framework for iteration applications. Spark supports rapid application development for big data and allows for code reuse in batch, interactive and streaming applications. Spark also provides advanced execution graphs with in-memory pipelining to speed up end-to-end application performance. In addition, Spark is packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing. These standard libraries can increase a developer’s productivity and can be seamlessly combined to create complex workflows.

3. Iterative Data Processing Framework

3.1 Motivation

The main characteristics of the existing iterative processing frameworks are listed in Table 2. It is noted that since both Twister and Spark provide their own independent structures for iterative processing, they are unable to utilize the existing Hadoop MapReduce applications. So, in this paper, we exclude both Twister and Spark for our consideration.

The drawbacks of the existing iterative processing frameworks are as follows. First, the existing frameworks fail to support efficient resource management for clusters and undermine the cluster’s scalability as they are implemented based on Hadoop v0.20. In other words, because the existing iterative processing frameworks perform both job scheduling and resource management for the entire cluster at the master’s JobTracker, the maximum scope of a cluster that JobTracker can accommodate is limited. Moreover, the existing iterative processing frameworks allow either Mapper or Reducer to monopolize the entire system resources at each node, thus decreasing the use rate of the entire cluster.

Figure 1 illustrates the job scheduling techniques of Hadoop v0.20 and Hadoop v2.2. JobTracker in Hadoop v0.20 handles both job scheduling and resource management, whereas in Hadoop v2.2, job scheduling is handled by MRAppMaster and resource management is handled by ResourceManager. In addition, MRAppMaster requests the containers to ResourceManager and executes the allocated containers at each node in Hadoop v2.2.

Secondly, because iMapReduce and iHadoop do not consider the characteristics of iterative applications, they fail to provide a caching policy for invariant data being used repeatedly while an iterative processing application is running. That is to say, when a job is being performed repeatedly, as in the case of iterative processing, there exist invariant data that are used repeatedly at each iteration. However, the existing frameworks re-transmit the same invariant data being used repeatedly at each iteration, thus causing unnecessary cost of network transmission. Even though the existing iterative processing frameworks offer invariant data caching, they cannot be used on Hadoop v2.2, due to being built on Hadoop v0.20.

Thirdly, because the existing iterative processing frameworks do not consider the entire resources of a cluster, containers for performing a task are concentrated on particular nodes. That is, when a task is not allocated uniformly to every node, it could weaken the performance of the entire MapReduce task. In particular, this issue of container over-concentration becomes worse for iterative applications as iterations are performed. Meanwhile, to address this issue, Facebook has developed FairScheduler which distributes the containers evenly to each node by considering the resource status of all of the nodes. However, although FairScheduler addresses the container overconcentration issue, it fails

<table>
<thead>
<tr>
<th>Framework</th>
<th>Hadoop Compatibility</th>
<th>Job Scheduling</th>
<th>Invariant Data Caching</th>
<th>Stop Condition Check</th>
<th>Resource Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaLoop [15, 25]</td>
<td>v0.20</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>N/A</td>
</tr>
<tr>
<td>iMapReduce [16]</td>
<td>v0.20</td>
<td>P</td>
<td>P</td>
<td>O</td>
<td>N/A</td>
</tr>
<tr>
<td>Hadoop [17]</td>
<td>v0.20</td>
<td>P</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Twister [18]</td>
<td>N/A</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Spark [19]</td>
<td>YARN-level</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>

* P : Partial, N/A : Not Applicable
to support iterative processing. Therefore, FairScheduler is not suitable for iterative applications since MapReduce job re-creation is required and the initial configuration cost at each iteration is needed for this purpose.

Finally, iHadoop undermines the efficiency of iterative applications since it does not support a stop condition check mechanism. Even though the desired output is already obtained during iterative processing, iHadoop would continue to perform the MapReduce job repeatedly until it reaches the pre-designated number of computations. This causes unnecessary computations for an iterative application to perform the entire processing.

To solve these problems, in this paper, we propose a new efficient resource management framework for iterative MapReduce processing in large-scale data analysis. Figure 2 shows the system architecture of our framework which consists of an iterative job state-machine, an invariant data caching mechanism, an iterative resource management technique, and a stop condition check mechanism.

### 3.2 Iterative Job State-Machine

Because the existing frameworks are implemented either on Hadoop v0.20 or on Hadoop v1.0, they fail to support the YARN framework that Hadoop v2.2 offers. Moreover, unlike Hadoop v0.20, job scheduling on Hadoop v2.2 can be performed only through the state-machine of MRAppMaster. To develop a job scheduling technique for iterative processing based on Hadoop v2.2 by considering this limitation, it is essential to implement it on a state-machine of MRAppMaster. Meanwhile, because Hadoop v2.2 is optimized to run a single job, another job must be newly created when it performs the same job again, as shown in Fig. 3.

To address these problems, we propose an iterative job state-machine designed to efficiently handle multiple jobs being performed repeatedly. First, the proposed iterative job state-machine runs multiple jobs being performed repeatedly in a single iterative job, thereby minimizing the cost of re-creating and configuring the jobs. To that end, the proposed iterative job state-machine manages multiple jobs for repeated executions by dividing them into steps, as shown in Fig. 4.

A step is the same unit as a job in the existing Hadoop and it consists of a single map/reduce task pair. While the approach to divide multiple jobs into steps has already been proposed by the existing iterative processing framework, e.g., HaLoop, the proposed system further expands this approach by using the state-machine of Hadoop v2.2.

The proposed iterative job state-machine checks the termination conditions of an iterative application at the newly added `TerminationCheckTransition` and performs `NewIterationTransition` only when the next iteration needs to be performed. Thanks to this feature, it is possible to perform the subsequent task continuously without creating a new job. In addition, the proposed iterative job state-machine can minimize the need to change the source of Hadoop, thereby resolving the compatibility issue upon Hadoop version upgrade. The MRAppMaster of the existing Hadoop has three state-machines: job, task and task attempt state-machine. The proposed framework only modifies the job state-machine of the existing Hadoop.

Table 3 shows the added and modified components of state-machine from Hadoop v2.2. Also, Fig. 5 shows the procedure of iterative job state-machine.
Furthermore, because all the jobs of Hadoop run independently with respect to I/O configuration for map and reduce functions, a user must load data each time a map task is performed. This not only deteriorates system performance but also places a huge burden on the user. To address this problem, the proposed iterative job state-machine stores the outcome of each iteration in HDFS and configures this data as its input at the subsequent map process. This eliminates the process of insertion data for input, thereby reducing the overhead for data loading (Fig. 6).

### 3.3 Invariant Data Caching Mechanism

The proposed invariant data caching mechanism can minimize the cost of unnecessary transmission of invariant data through network by storing invariant data from each iteration in the local disk of each node for a reusing purpose.

Meanwhile, because the data and computation processes used in the iterative applications of MapReduce are different from each other, an optimized caching policy for each iterative application is essential. For this, we propose new caching mechanisms, such as map input cache and reduce input cache.

First, the map input cache is a caching mechanism from the mapper side to cache data transmitted from other nodes within HDFS when the same input data are used for each iteration. For this, our algorithm creates a local index file by using a hash function to find the location of the cached file. Using the local index file, our algorithm stores the splits of the first iteration into the local file. In the next iteration, all the mappers load their local files as an input.

This mechanism can enhance processing efficiency by reducing the cost of transmitting the input data being used repeatedly at each iteration through network. A representative iterative application to which the map input cache can be applied is a k-Means algorithm (Fig. 7).

Figure 8 shows the map input cache algorithm. First, the algorithm reads both a current iteration number and a current step number from the configuration class, i.e., JobConf (lines 1–2). The step consists of a pair of a single map task and a single reduce one for an iteration. If a user configures a map input cache using an iterative processing API, the algorithm stores and reads the cache data. Based on the current number of iteration and step, the algorithm checks the status of the caching data at the current execution stage (lines 3–4). If the caching data is stored at the current stage, a caching file is created by using the local file system of the node involved (lines 5–6). When the caching data is used at the current execution stage, the caching file stored in the local file of the node is read (lines 7–8). Because no cached data exists for the first iteration, the input data given by the...
user is stored into the local disk of the node (lines 9–12). At this time, the caching file is stored through a byte stream in order to minimize the overhead of file access by using the serialization factory library provided by Java. Once the caching file is generated, the given user-defined map function is performed (line 13). Because the cache data exist for the second iteration onwards, the caching data is read and configured as the input data to perform the map function (lines 14–17).

Second, the reduce input cache is a caching mechanism from the reducer side, in order to cache data being repeatedly used at the shuffle stage of MapReduce. This mechanism can reduce the cost of transmitting intermediate data being used repeatedly at each iteration. For this, our algorithm stores the input of the first iteration into the local files of reducers and then creates a local index file by using the key of the input. When the key is found in the local index file, our algorithm merges the input with the existing local files. Otherwise, our algorithm performs the reduce function of Hadoop.

A representative iterative application to which the reduce input cache can be applied is the page rank algorithm (Fig. 9). Meanwhile, data for a certain key need to be identically assigned to each iteration, in order to provide the reduce input cache. This is because when data with different keys are assigned for each iteration, it is impossible to utilize the data being cached from the previous iteration. The proposed framework is designed such that data with the same key are assigned to the same node for each iteration, by the data locality policy of the iterative resource scheduler.

Figure 10 shows the reduce input cache algorithm. If a user configures the reduce input cache using an iterative user API, the algorithm can utilize the cache data. First, the algorithm reads both a current iteration number and a current step number from JobConf (lines 1–2). Based on the current numbers of iteration and step, the algorithm checks the status of the caching data at the current stage (lines 3–4). If the caching data is stored at the current execution stage, a caching file and an index file of the cache data are created by using the local file system of the node (lines 5–7). When data with the same key is searched for the cache data, the index file of the cache data is used for the purpose of combining the existing caching file previously stored and the map outcomes transmitted from the map stage. When the caching data is used at the current execution stage, the caching file and the index file stored in the local file of the node are loaded (lines 8–10). Because no cached data exists for the first iteration, the data being configured as invariant data among map outcomes are stored in the caching file (lines 11–14). At this time, the byte offset of the caching file is recorded into the index file for combining the caching data and map outcomes. Meanwhile, the invariant data are configured by the user through the iterative user API and the caching file is stored into the map input cache, in order to minimize the overhead of file access. Once the caching file is generated, the given user-defined reduce function is performed (line 15). Because the cache data exist for the second iteration onwards, the caching data are read by using the index file and are combined with the transmitted map outcomes through the map execution stage (lines 16–21). Once the combination of the caching data and the map outcomes is completed, the user-defined reduce function is performed (line 22).

### 3.4 Iterative Resource Management

For cluster resource management, Hadoop v2.2 provides FIFO scheduler, Capacity Scheduler, and FairScheduler by default. However, the schedulers provided by the existing Hadoop v2.2 have the following limitations. First, be-
cause FifoScheduler and Capacity Scheduler fail to consider the amount of entire resources of a cluster, containers concentrate too much on particular nodes. Due to this shortcoming, tasks are not evenly distributed among nodes, thus undermining the performance of MapReduce applications. Meanwhile, FairScheduler is a scheduler being developed by Facebook, which evenly assigns the entire resources of a cluster to tasks. This ensures the even distribution of resources, thereby enhancing the usage rate of the entire resources of the Hadoop cluster. However, the problem with FairScheduler is that it requires the initial configuration cost for each iteration because tasks need to be re-created when an iterative application is executed.

Secondly, the schedulers provided by the existing Hadoop v2.2 do not offer the data locality policy that allows data with the same key to be assigned to a particular node for each iteration when an iterative application is being executed. Thus, they cannot support the caching mechanism of invariant data that are used repeatedly. To address the issues, we propose a cluster resource management technique designed for efficient resource management in iterative applications.

### 3.4.1 Meta-Table for Iterative Processing

In order to distribute the containers evenly among nodes, it is essential to have the total number of containers required for a job and the resource status information of the entire cluster. As such, a meta-table for iterative processing can be shared for an iterative application by MRAppMaster and ResourceManager. At this time, the total number of containers required by the job is sent to ResourceManager through a job registration process for the job object of the MapReduce application. In addition, the resource status information of the entire cluster is transmitted when the NodeManager of each node sends the resources status to ResourceManager. Furthermore, the iterative job state-machine of MRAppMaster stores the progress status of the current iterative application and the use of map/reduce cache in the meta-table for iterative processing.

The meta-table for iterative processing consists of four files, i.e., map input cache (map_cache), reduce input cache (reduce_cache), the status of the iterative processing (cur_round), and the total number of map tasks at current iteration (cur_num_maps). Because both map input cache and reduce input cache are not changed during iterative processing, they can be stored into HDFS once when MRAppMaster submits an iterative application to ResourceManager. Meanwhile, the status of the iterative processing and the total number of map tasks at current iteration are updated by MRAPPMaster whenever a new iteration is initialized. To update them in the HDFS, we first delete the existing files and then create new files, due to the immutability of HDFS files. On the other hand, the total number of reduce tasks is not required to be stored into the meta-table because it can be received from the configuration class.

Figure 11 illustrates the procedure of sharing the meta-table between MRAppMaster and ResourceManager. The step ① and ② are performed once when an iterative application is submitted, whereas the step ③ is performed repeatedly at each iteration so as to update both the status of the iterative processing and the total number of map tasks at the current iteration. Moreover, the step ④ is performed periodically whenever sending the resource status of a node to ResourceManager. When the resource status of a node is updated by NodeManager, the step ⑤ is performed once to evenly distribute the containers into nodes.

### 3.4.2 Iterative Resource Scheduler

Like the existing resource schedulers, the iterative resource scheduler is put together as a sub-component of ResourceManager. In addition, the proposed iterative resource scheduler is provided independently from the existing resource schedulers, in order to minimize the need to revise the existing source codes of Hadoop.

To maximize a cluster’s overall usage of resources, the iterative resource scheduler distributes the containers evenly among nodes by considering the usage rate of the entire resources of the cluster. The existing resource scheduler of Hadoop v2.2 has a problem that containers concentrate excessively on particular nodes. Especially with iterative applications, the same number of containers used at the first iteration should be maintained at each subsequent iteration for caching invariant data. Accordingly, when an iterative application uses the resource schedulers of Hadoop with the iterative processing framework, it may aggravate over-concentration of containers. To solve the problem, the proposed iterative resource scheduler distributes the containers evenly among nodes by considering the number of the entire nodes as well as the number of all map/reduce tasks to be performed at the current iteration.

Figure 12 describes the container allocation algorithm of the proposed iterative resource scheduler. In order to allocate the containers, the task type of each container is first checked with ResourceManager (line 1). For this, our algo-
Container Allocation Algorithm

**Input**: NodeResourceInformation nodeInfo, ResourceManagerInformation manageInfo

**Output**: NumberOfAssignedContainer assigned

1: taskType = getPriority(manageInfo)
2: maxContainer = getMaxAllocatableContainers(nodeInfo, manageInfo, taskType)
3: if(maxContainer < 0) return
4: totalMaps, totalReduces = getIterativeInformation()
5: for node[i] in nodeList
6:     map_avg = calculate_map_resource(totalMaps, nodeInfo)
7:     set_maxMapAllocation (node[i], map_avg)
8:     reduce_avg = calculate_reduce_resource(totalReduces, nodeInfo)
9:     set_maxReduceAllocation (node[i], reduce_avg)
10: end of for
11: if(taskType is MapTask)  // for Map Task
12:     nodeLocal = assignNodeLocal (nodeInfo, manageInfo)
13:     rackLocal = assignRackLocal (nodeInfo, manageInfo)
14:     offSwitch = assignOffSwitch (nodeInfo, manageInfo)
15:     else if(taskType is ReduceTask)  // for Reduce Task
16:     reduce = assignReduce (nodeInfo, manageInfo)
17:     else
18:     offSwitch = assignOffSwitch (nodeInfo, manageInfo)
19: return nodeLocal + rackLocal + reduce + offSwitch

**Fig. 12** Container allocation algorithm.

The algorithm obtains the priority value of each container by using `getPriority()` method in `Priority` class. For example, the priority of map task is 5 or 20, whereas that of reduce task is 10. Then the maximum number of containers that can be assigned to the current node is checked, and if the containers cannot be created, the algorithm is terminated (lines 2–3). For even distribution of containers, the meta-table for iterative processing is obtained for the iterative application (line 4). The maximum number of containers that can be allocated to each node is calculated by each node (lines 5–10). Depending on the type of each task, the containers are assigned to each node by using the usage status of node resources and the resource scheduling information (lines 11–18). At this time, the existing policy of Hadoop is maintained as the container allocation policy. That is, the containers in a map task are allocated by considering the locality of each node for input data, whereas a reduce task and MRAppMaster do not consider the locality. Finally, when the containers are successfully allocated, the number of containers generated is returned (line 19).

### 3.5 Stop Condition Check Mechanism

Iterative processing applications run repeatedly as many times as the number of iterations defined by the user. However, if the desired outcome is obtained while an iterative processing application is still running, it is necessary to stop the application to prevent unnecessary execution. For this, we propose a stop condition check mechanism which determines whether or not an iterative application should be run repeatedly by comparing the outcomes of the current iteration with that of the previous iteration. That is, if the difference between them falls within a threshold given by the user, the iterative application is terminated to maximize its efficiency. Because the outcomes of MapReduce need to be compared with all outcomes of the previous iteration, the stop condition checking requires heavy overhead. For the reuse of the outcomes, the proposed mechanism caches the outcomes of the previous iteration in the local disk of each node through reduce output cache. This can not only minimize unnecessary cost involved in disk input and output, but also enhance data processing efficiency.

Figure 13 illustrates how to process the stop condition of the page rank algorithm. In the page rank algorithm, the compute rank job recalculates the weight of each node based on the weights of neighboring nodes. Meanwhile, the rank aggregate job combines the weights of nodes being calculated at the compute rank job and sends the combined results to the compute rank job of the next iteration. At this time, the iterative processing job of the page rank algorithm stops when the changed weight of each node converges within a particular threshold or when it is run as many times as the given number of repetitions.

Because the stop condition for iterative applications generally varies from one application to another, we provide a user with a final-goal stop condition configuration class and the user can configure the stop condition for a given application. If a user does not write a final-goal stop condition configuration class, the stop condition of a given application is configured by using the number of iterations only. When the termination check object of MRAppMaster runs a stop condition check mechanism, the final-goal stop condition configuration class is called. The termination check object is run at the `TerminationCheckTransition` of the iterative job state-machine in Sect. 3.2. Because the proposed framework checks a stop condition through the iterative job state-machine, it does not require any additional job for checking the stop condition. In addition, the proposed framework requires a user’s minimum interference for the modification of the stop condition configuration, thereby providing users, like non-expert developers, with convenience.
### 3.6 Add-On Iterative Framework APIs

To offer an easy-to-use programming interface to the users, our framework provides an add-on iterative framework APIs based on the existing Hadoop’s APIs. The add-on iterative framework APIs are designed to execute the existing Hadoop MapReduce’s applications without modification. Table 4 shows the description of our add-on iterative framework APIs.

Figure 14 shows a pseudo-code of the k-Means application using our add-on iterative framework APIs. By just adding our add-on iterative framework APIs into the main function, a user can reuse the existing iterative applications without the modification of user-defined map/reduce classes (lines 8–15). Therefore, our framework does not require the re-creation of the existing iterative applications. Whereas, Spark should create new iterative applications because it not only utilizes an independent data processing structure, i.e., RDD (Resilient Distributed Dataset) [23], but also uses Scala as a primary programming language.

#### Table 4 Description of our add-on iterative framework APIs.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setIterative</code></td>
<td>Set the add-on iterative framework APIs</td>
</tr>
<tr>
<td><code>setNumIterations</code></td>
<td>Specify how many numbers of iterations are performed maximally</td>
</tr>
<tr>
<td><code>setStepJob</code></td>
<td>Specify which step is performed in an iterative job</td>
</tr>
<tr>
<td><code>setCombiner</code></td>
<td>Specify a step being configured by <code>CombinerClass</code></td>
</tr>
<tr>
<td><code>setSortandGrouping</code></td>
<td>Specify a step being configured by <code>SortComparatorClass</code></td>
</tr>
<tr>
<td><code>setMapInputCacheOption</code></td>
<td>Set the map input cache</td>
</tr>
<tr>
<td><code>setMapCacheSwitch</code></td>
<td>Specify which step of a iterative job read/write the map input cache</td>
</tr>
<tr>
<td><code>setIterationInputOutput</code></td>
<td>Specify a user-defined class for configuring the input/output paths of each iteration</td>
</tr>
<tr>
<td><code>setReduceInputCacheOption</code></td>
<td>Set the reduce input cache</td>
</tr>
<tr>
<td><code>setReduceCacheSwitch</code></td>
<td>Specify which step of a iterative job read/write the reduce input cache</td>
</tr>
<tr>
<td><code>setReducerCacheFilter</code></td>
<td>Specify a user-defined class for establishing the invariant data from the reducer</td>
</tr>
<tr>
<td><code>setReducerOutputCacheOption</code></td>
<td>Set the reduce output cache for stop condition checking</td>
</tr>
<tr>
<td><code>setDistanceMeasure</code></td>
<td>Specify a user-defined class for configuring the fix-point stop condition</td>
</tr>
<tr>
<td><code>setReduceOutputCacheSwitch</code></td>
<td>Specify which step of a iterative job read/write the reduce output cache</td>
</tr>
<tr>
<td><code>setFixpointThreshold</code></td>
<td>Specify a threshold for a fix-point stop condition</td>
</tr>
<tr>
<td><code>setTerminationCheck</code></td>
<td>Specify a user-defined class for configuring the final-goal stop condition</td>
</tr>
</tbody>
</table>

### 3.7 Comparison between Our Framework and HaLoop

HaLoop is the most popular iterative data processing framework which handles the Hadoop-based iterative MapReduce applications. Similar to our framework, HaLoop not only provides loop-aware task scheduling, but also supports invariant data caching and stop condition checking. However, our framework has differences from HaLoop in the viewpoints of such five aspects as i) Hadoop API compatibility, ii) efficient job scheduling for iterative MapReduce, iii) caching intermediate key-value pairs, iv) stop condition checking and v) efficient resource management.

#### 3.7.1 Hadoop API Compatibility

Hadoop provides two MapReduce APIs: the *old API* and the *new API*. Our framework supports the *new API* based on the mapreduce library package, whereas HaLoop supports the *old API*. Although the *old API* is still used, many MapRe-
duce applications tend to use the new API.

3.7.2 Efficient Job Scheduling for Iterative MapReduce

Because HaLoop is implemented based on Hadoop v0.20, it provides the loop-aware task scheduler using the event model based on functional calls. When HaLoop executes multiple MapReduce applications, many threads must wait until the previous event is finished. Therefore, HaLoop is not suitable for handling multiple MapReduce applications. Whereas, our framework designs a new iterative job state-machine using the event-driven concurrency model so that it may efficiently support multiple MapReduce applications.

3.7.3 Caching Intermediate Key-Value Pairs

For invariant data caching, HaLoop offers a data locality policy that allows the invariant data with the same key to be assigned into a particular node. Therefore, HaLoop requires a large amount of changes due to the version update of Hadoop because its data locality policy is tightly coupled with the existing task scheduler of Hadoop. Our framework also offers a data locality policy for invariant data caching mechanism. Meanwhile, our framework provides a stand-alone iterative resource scheduler that supports the data locality policy based on the iterative resource management. For this, we newly design a container allocation index in our iterative resource scheduler. As a result, our framework can minimize the internal changes of the invariant data caching mechanism due to the version update of Hadoop.

3.7.4 Stop Condition Checking

For the stop of iterations, HaLoop provides two fix-point stop conditions, i.e., the maximum number of iterations and the difference between the threshold values of two consecutive iterations. Whereas, our framework provides not only the above two fix-point stop conditions, but also the final-goal stop condition. The final-goal stop condition is used to stop iterations when the final destination is reached. For example, when a moving object moves from a starting point in a graph, the iterative application can be terminated using our final-goal stop condition when the moving object reaches the destination.

3.7.5 Efficient Resource Management

Because HaLoop does not provide an iterative resource scheduler, it has a problem that tasks are not uniformly distributed into nodes, thus leading to performance degradation. Whereas, because our framework provides a new iterative resource scheduler to evenly distribute tasks among nodes, it can greatly improve the overall performance of iterative processing.

4. Performance Analysis

In this section, we present the performance analysis of our iterative data processing framework. We compared our framework with both Hadoop v2.2 and HaLoop. This is because HaLoop shows the best performance among the existing Hadoop-based iterative data processing frameworks [20]. The reasons why we exclude Spark for a competitor for our performance comparison are as follows. First, even though Spark has the YARN-level compatibility, it requires the re-creation of iterative applications for executing them because it not only utilizes an independent data processing structure, i.e., RDD, but also uses Scala as a primary programming language. Second, although Spark can utilize the data stored in the disk, it requires a large amount of memory to persist the RDDs [24], [25]. Whereas, Hadoop requires a relatively small-sized memory for executing various applications with a huge amount of data.

We setup a Hadoop cluster consisting of seven physical machines (1 master node and 6 slave nodes) as specified in Table 5. Table 6 shows our parameter settings of the Hadoop cluster and the remaining parameters are set to the default values of Hadoop.

For performance analysis, we evaluate the performance of our framework for three applications; page rank, descendant query, k-Means. We do performance analysis in terms of the total execution time and the execution times of map/reduce/shuffle phases.

The page rank and the descendant query applications uses the Live Journal graph dataset [21] which represents the relationship of the user community in Live Journal website. It contains 4,847,571 nodes and 68,993,773 edges. The k-Means application uses the Ocean Color Data from MODIS (Moderate Resolution Imaging Spectroradiometer) [22]. It includes geosocial data, e.g., green algae and sea level, which are observed from Terra satellite. The daily green algae data within 4 kilometers are used in our experiment.

4.1 Performance of Page Rank Application

Because the page rank application has an invariant data from
the reducer side, we apply the reduce input cache to our framework. We set up the number of iteration as 10.

Figure 15 shows the total execution time, varying the size of data from 400MB to 4GB. When the data size is 4GB, the total execution time of the proposed framework is about 961 seconds, whereas those of HaLoop and Hadoop are about 4,733 and 7,984 seconds, respectively. As a result, our framework has up to 8.3 times better performance than the existing works. This is because our framework can minimize the cost of re-creating and configuring jobs by using iterative job state-machine. Also, we can maximize a cluster’s overall usage of resources by using our iterative resource management technique. Furthermore, we minimize the I/O cost of shuffling invariant data in subsequent iterations by using invariant data caching mechanism.

Figure 16 shows the execution times of map/shuffle/reduce phases when the data size is 400MB. The execution times of map/shuffle/reduce phases for our framework are about 93, 61, and 78 seconds, respectively. Whereas, the execution times of map/shuffle/reduce phases for Hadoop are about 395, 464, and 971 seconds, respectively. It is shown from our performance results that we can reduce the execution time of shuffle phase by using the reduce input cache. Meanwhile, the execution time of map phase for our framework is 93 seconds, whereas that of HaLoop is 393 seconds. This is because our iterative resource management technique can evenly distribute the containers into nodes by considering the usage status of the entire resources. However, HaLoop has a problem that tasks are not uniformly distributed into nodes because it does not consider the usage status of resources in a cluster.

4.2 Performance of Descendant Query Application

Because the descendant query application requires an invariant data from the reducer side, like the page rank application, we use the reduce input cache and set up the number of iteration as 10.

Figure 17 shows the total execution time, varying the size of data from 400MB to 4GB. When the data size is 4GB, the proposed framework requires the total execution time of about 784 seconds, whereas HaLoop and Hadoop require about 3,261 and 6,209 seconds, respectively. As a result, our framework has up to 7.9 times better performance than the existing works. This is because our framework can minimize the cost of re-creating jobs and the I/O cost for invariant data from the reducer side.

Figure 18 shows the execution times of the map/shuffle/reduce phases when the data size is 400MB. The execution times of the map, shuffle, and reduce phases for the proposed framework is about 133, 97, and 67 seconds, respectively. Whereas, the execution times of the map, shuffle, and reduce phases for Hadoop is about 635, 329, and 32 seconds, respectively. It is shown from our performance
results that we can reduce the execution time of the shuffle phase by using the reduce input cache. As a result, our framework has up to 4.7 times and 10.4 times better performance than the existing works, in terms of the map and shuffle phases. The execution time of the reduce phase for the proposed framework is increased compared to the existing works. This occurs due to the characteristic of the descendant query application. The descendant query application can be expressed into two steps; join step and duplicate elimination step. In join step, a MapReduce job is required to join an input node table and a node linkage table. In duplicate elimination step, another MapReduce job is defined to delete the nodes which can be found in the input node table. In the existing works, because the most of input data is transmitted through the network, the cost of shuffling occupies most of the reducer processing. On the contrary, the proposed framework reuses the caching data in each node by using the reduce input cache. Therefore, the time of loading caching data is increased, compared to the existing Hadoop. Meanwhile, the execution time of map phase for our framework is 133 seconds, whereas that of HaLoop is 635 seconds. Because HaLoop does not consider the usage status of resources in a cluster, the tasks of HaLoop tend to be assigned into particular nodes. Whereas, due to our efficient resource management technique, our framework can evenly distribute containers (i.e., tasks) into nodes.

4.3 Performance of k-Means Algorithm Application

Because the k-Means application requires an invariant data from the mapper side, we use the map input cache and set up the number of iteration as 3.

Figure 19 shows the total execution time, varying the size of data from 391MB to 3GB. When the data size is 3GB, the total execution time of the proposed framework is about 275 seconds, whereas HaLoop and Hadoop require about 428 and 2,155 seconds, respectively. As a result, our framework has up to 7.8 times better performance than the existing works. This is because our framework can minimize the cost of re-creating jobs and the I/O cost for invariant data from the mapper side.

Figure 20 shows the execution times of the map/shuffle/reduce phases when the data size is 391MB. The execution time of the map, shuffle, and reduce phases for proposed framework is about 28, 31, and 13 seconds, respectively. Whereas, the execution time of the map, shuffle, and reduce phases for Hadoop is about 105, 74, and 69 seconds, respectively. It is shown from our performance result that we can reduce the execution time of the map phase by using the map input cache. Furthermore, we can reduce the execution time of both shuffle and reduce phases by using our iterative resource scheduling technique. Meanwhile, the execution time of map phase for our framework is 28 seconds, whereas that of HaLoop is 105 seconds. This is because our framework can maximize resource utilization by uniformly assigning resources into nodes while HaLoop has a problem that tasks tend to be assigned to particular nodes.

4.4 Cost of Accessing Meta-Table

Figure 21 shows the cost of accessing meta-table when the data size of each application is about 400MB. The total access time of meta-table in page rank, descendant query, and k-Means is about 0.179, 0.178, and 0.181 seconds, respectively. Whereas, the total execution time of page rank, descendant query, and k-Means is about 232, 298, and 73 seconds, respectively, as shown in Fig. 15, Fig. 17 and Fig. 19.
Therefore, it is shown that the total access time of meta-table occupies about 0.07% of the total execution time.

The access time of meta-table in a node at each iteration is at most 0.012 seconds, regardless of the data sizes and the types of applications. This is because the meta-table being stored in HDFS can be accessed once when a task is assigned to each node in an iteration. As a result, it is shown that the cost of accessing meta-table is negligible, compared with the total execution time.

5. Conclusions and Future Work

Recently, large-scale data have attracted much attention with the development of information technology. Because large-scale data has a massive volume of complex data, an efficient computing framework for large-scale analysis is required. For this, we proposed a new efficient resource management framework for iterative MapReduce processing in large-scale data analysis. In our framework, the iterative job state-machine can minimize the initialization setup cost. In addition, we devised our invariant data caching mechanism which can reduce the I/O cost of loading and shuffling invariant data in the subsequent iterations. Our iterative resource scheduling technique was designed to maximize the resource utilization by allocating resources uniformly to every node. Furthermore, our stop condition check mechanism prevents the unnecessary computations.

From our performance analyses, it was shown that our framework outperforms the existing works. In case of the page rank and the descendant query applications, the proposed framework shows 7.9 to 8.3 times better performance than the existing Hadoop, by using the reduce input cache. In case of the k-Means application, the proposed framework shows 7.8 times better performance by using the map input cache. As a result, our iterative data processing framework is suitable for iterative applications because it can provide the efficient job and resource scheduling with both our invariant data caching and stop check mechanisms.

As future research, we plan to study on an indexing technique that can automatically detect invariant data in our caching mechanism.

Acknowledgments

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0113-16-0005, Development of an Unified Data Engineering Technology for Large-scale Transaction Processing and Real-time Complex Analytics) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0101-16-0081, Research and Development of Dual Operating System Architecture with High-Reliable RTOS and High-Performance).

References


Seungtae Hong received the B.S. and M.S. degrees in Computer Engineering from Chonbuk National University in 2008 and 2010, respectively. Also, he earned his Ph.D. degree from Chonbuk National University in 2015. He is currently a senior member of engineering staff in the Electronics and Telecommunications Research Institute (ETRI). His research interests include embedded software, big data and cloud computing.

Kyongseok Park received the B.S. and M.S. degrees from University of Seoul in 2000 and 2002, respectively. Also, he earned his Ph.D. degree from Korea University of Technology and Education in 2012. He is currently a manager of scientific data technology laboratory in the Korea Institute of Science and Technology Information (KISTI). His research interests include distributed computing, mathematical analysis and machine learning.

Chae Deok Lim received the Ph.D. degree from Chungnam National University in 2005. During 1989, she is a principal member of engineering staff in the Electronics and Telecommunications Research Institute (ETRI). From 2012 to 2016, she was the mananaging director of embedded software research department in the ETRI. Her research interests include system software, embedded system and dual operating system.

Jae-Woo Chang received the B.S. and M.S. degrees from Korea Advanced Institute of Science and Technology in 1986 and 1991, respectively. From 1996 to 1997, he was a visiting scholar of University of Minnesota. Also, from 2003 to 2004, he was a visiting scholar of Pennsylvania State University. He is currently a professor of department of information technology in the Chonbuk National University. His research interests include database system, big data and cloud computing.