Bin Saadon and Mokhtar *J Big Data* (2017) 4:24
DOI 10.1186/s40537-017-0086-3

Journal of Big Data

RESEARCH Open Access

iiHadoop: an asynchronous distributed framework for incremental iterative computations

Afaf G. Bin Saadon* and Hoda M. O. Mokhtar

*Correspondence:
eng.afaf.fci@gmail.com
Faculty of Computers and Information, Cairo University, Giza, Egypt

Abstract

It is true that data is never static; it keeps growing and changing over time. New data is added and old data can either be modified or deleted. This incremental nature of data motivates the development of new systems to perform large-scale data computations incrementally. MapReduce was recently introduced to provide an efficient approach for handling large-scale data computations. Nevertheless, it turned to be inefficient in supporting the processing of small incremental data. While many previous systems have extended MapReduce to perform iterative or incremental computations, these systems are still inefficient and too expensive to perform large-scale iterative computations on changing data. In this paper, we present a new system called iiHadoop, an extension of Hadoop framework, optimized for incremental iterative computations. iiHadoop accelerates program execution by performing the incremental computations on the small fraction of data that is affected by changes rather than the whole data. In addition, iiHadoop improves the performance by executing iterations asynchronously, and employing locality-aware scheduling for the map and reduce tasks taking into account the incremental and iterative behavior. An evaluation for the proposed iiHadoop framework is presented using examples of iterative algorithms, and the results showed significant performance improvements over comparable existing frameworks.

Keywords: Big data, Distributed systems, Hadoop framework, Iterative processing, Incremental computation

Introduction

Today, a large amount of data is being produced in many areas including: e-commerce, social network, finance, health-care, and education. This increase in data volume consequently increases the need for an efficient computing framework to process this data and transform it into meaningful information. In the past years, many distributed computing frameworks [1–6] have been developed to perform large-scale data processing. MapReduce [2] (with its open-source implementation, Hadoop [7]) is the most widely used framework because of its simplicity, scalability, efficiency, and reliability. It was proposed by Google in 2004 to enable the distribution and processing of large-scale data over a cluster of commodity machines. The framework takes care of the distributed execution, fault tolerance, load balancing, and job scheduling; therefore, developers and programmers do not need to worry about these issues and can simply perform their tasks.

MapReduce is designed for batch parallel data processing such as log analysis and text processing. However, many data analysis techniques need iterative processing such as machine learning, graph analysis, and data mining algorithms [8–10]. MapReduce does not directly support these iterative data analysis algorithms. It requires users to implement iterative algorithm as a chain of jobs where the output from each job is used as input to the next job. This approach is inefficient and too expensive because it wastes resources on launching, scheduling, and cleaning up these jobs, and as a result leads to long processing time.

Iterative data analysis algorithms often process large volume of data and consequently consume long time to complete their tasks. Input data to these algorithms may change over time and hence their previously computed results become stale and inaccurate. For example, search engines periodically crawl the web and perform different computations on their input (e.g., PageRank), often with small modifications to previous input data. It is hence desirable to periodically refresh the iterative computation so that the new changes can be quickly reflected in the computed results. However, starting the computations from scratch is too expensive; therefore, it is necessary to develop new systems to tackle this situation in an efficient way.

Recently, a variety of systems have been proposed to address this challenge. Unfortunately, some of these systems designed new programming models to support incremental processing that differ from the one used by MapReduce (e.g., Percolator [11], Naiad [12]). Other systems extended the MapReduce framework but they only support one-step incremental computations (e.g., Incoop [13] and IncMR [14]). Recently, i^{2}MapReduce was proposed in [15, 16]. i^{2}MapReduce supports key-value pair level incremental processing for both one-step and iterative computations. However, i^{2}MapReduce uses a static scheduling policy where tasks stay alive across multiple iterations even if they remain idle, and requires equal number of map and reduce tasks to allow asynchronous execution which is not efficient.

In this paper, we present a new framework called iiHadoop, that extends both the traditional Hadoop MapReduce and i^{2}MapReduce to support efficient incremental iterative computations. iiHadoop overcomes the limitations of existing systems and efficiently implements incremental computation using key-value pair level approach presented in i^{2}MapReduce [16]. iiHadoop presents the following main contributions:

- It offers a new programming interface to express iterative and incremental applications. Despite the fact that users need to slightly modify their MapReduce program to use the iterative and incremental functionalities of iiHadoop, this modification is modest compared to re-implementing their algorithms on a different programming model, such as Naiad.
- It modifies the task scheduler to be aware of the incremental and iterative behavior. Since the changes affect only a small fraction of data, the scheduler creates only the map and reduce tasks that perform incremental computations on these changing data rather than the whole data, and kill them when they are completed. In addition, the scheduler assigns the map and reduce tasks to the nodes taking into account the location of their input data, and delays the execution of the task if the nodes that holds its input data have no free slot.

- It modifies the dataflow of the traditional MapReduce framework so that it is able to execute iterations asynchronously, where an iteration can start before its preceding iteration finishes without violating the overall computation logic. The asynchronous execution works efficiently with different number of map and reduce tasks in contrast to other systems that require same number of map and reduce tasks to allow asynchronous execution like i^{2}MapReduce. This feature enhances the overall computation efficiency.
- It is implemented on Hadoop 1.x and Hadoop 2.x with YARN, and many experiments are conducted to evaluate the results of these two implementations against each other and against other earlier comparable approaches.

To evaluate the performance of iiHadoop, a series of experiments are conducted on a virtual machine cluster built on Amazon EC2 cloud. Several well-known iterative algorithms are used in the experiments including: PageRank, k-means, connected components, and single source shortest path. The experimental results show that iiHadoop accelerates the incremental iterative processing and achieves better performance compared to traditional Hadoop.

The rest of this paper is organized as follows. "Problem statement and background" section describes the research problem and gives an overview of the main concepts. "Related work" section discusses the related work. "System design" section presents the design and implementation of iiHadoop. "Experimental evaluation" section evaluates iiHadoop performance. Finally, "Conclusion and future work" section concludes the paper and presents directions for future work.

Problem statement and background

Problem statement

Iterative computation is generally needed for a large and important class of machine learning and data mining algorithms. Many of these algorithms frequently need to apply iterative computations on a small incremental data. It is hence necessary to develop a good solution to efficiently apply incremental iterative computations on a large-scale datasets. While a variety of systems have been proposed to address this challenge; however, they have the following significant limitations:

- Some of these systems support only one-step incremental computations while many important algorithms need extensive iterative computation.
- Some of these systems require a new programming model that differs from the one used by MapReduce, and to use their incremental and iterative functionalities, programmers need to rewrite their MapReduce programs and re-implement their algorithms.
- Some of these systems use static scheduling policy where tasks stay alive across multiple iterations. This policy may cause waste of resources because it limits and reserves some resources of the available infrastructure to a certain job even if they remain idle. In this case, other jobs may not be able to start until the first job is completed and its resources are released.

- Some of these systems work in a synchronous mode where map tasks start when all reduce tasks of the previous iteration are completed. This approach introduces unnecessary wait time that may eventually causes an overall increase in computation time. Some systems overcome this limitation and execute iterations asynchronously by creating reduce tasks equal to map tasks, co-locating each pair on the same node, and then passing the output of each reduce task directly to its corresponding map task. Unfortunately, this approach is inefficient because it adds heavy load to the nodes due to the large number of tasks.

In this paper, we overcome these limitations and present iiHadoop, an optimized version of Hadoop, to efficiently execute incremental iterative computations.

Background

In this section, we give an overview of the main concepts that are employed in this paper.

Hadoop framework

Hadoop is an open source distributed framework designed for running scalable data-intensive applications on a large cluster of commodity machines. Hadoop is composed of a computational part known as MapReduce which processes very large datasets in a distributed manner, and storage part known as hadoop distributed file system (HDFS) which stores data across thousands of commodity machines [7].

MapReduce programming model

The MapReduce program consists of a Map function and a Reduce function [2]. The signatures of these functions are:

$$Map\text{: } (k1, v1) \rightarrow [(k2, v2)],\ Reduce\text{: } (k2, [v2]) \rightarrow [(k3, v3)]$$

The map function is applied on every (k1, v1) pair and produces a list of intermediate (k2, v2) pairs. The shuffle phase groups together all intermediate values associated with the same key k2 and passes them to the reduce function. The reduce function then is applied on all intermediate pairs with the same key and outputs a list of (k3, v3) pairs.

Hadoop distributed file system

Hadoop uses a distributed file system called HDFS [17] to store input data, intermediate results, and final results. HDFS divides the input data into equal-sized (e.g., 64 or 128 MB) blocks and stores them on the local disk of worker nodes. It also replicates the data on multiple nodes so that a node failure does not affect the computations [7].

Task scheduling

Hadoop schedules user's jobs using the default FIFO scheduler, where jobs are scheduled based on their priorities in a first-come first-serve manner. Each job consists of a set of tasks, and is assigned by the scheduler to a set of available worker nodes taking into account the location of their input data. The scheduler attempts to assign the map task on a node that contains a replica of its input data [2]. Hadoop runs a JobTracker process

on the master node to monitor the job progress, and a set of TaskTracker processes on worker nodes to perform the actual Map and Reduce tasks.

Apache YARN

YARN stands for "Yet Another Resource Negotiator". It is a generic framework which developed to run any kind of distributed application on top of it [18]. YARN enhances Hadoop by separating resource management functions from the programming model. In Hadoop 2.x, the MapReduce is responsible for performing data processing while YARN is responsible for managing the resources of the cluster (e.g., memory and CPU). Hadoop 2.x changes its architecture to be suitable for YARN. The responsibilities of the JobTracker/TaskTracker are split up into separate entities including: ResourceManager, ApplicationMaster, NodeManager, and Container running on a NodeManager.

Iterative processing

The term "Iterative Processing" is used to describe the situation where a sequence of instructions is executed and repeated multiple times until the desired results are achieved. Each repetition of the process is called an iteration. Iterative processing uses the output from one iteration as an input to the next iteration, and as the number of iterations increases, the processing comes closer to convergence [19]. Several important and well-known data analysis applications process data iteratively such as: PageRank, k-means clustering, logistic regression, single source shortest path, adsorption, connected components, descendant query, and Breadth-First Search [10]. Later in this paper we test the performance of iiHadoop using a set of these algorithms.

Incremental processing

Data is incremental by nature, it grows over time as new entries are added and existing entries are deleted or modified. Incremental processing is the process of performing some computations on only the latest, unprocessed data instead of the entire dataset. Incremental processing offers an efficient and good solution to avoid recomputation over the whole dataset after every small change.

Related work

In this section we explore several related research activities that have been conducted in four main directions: (1) iterative processing, (2) incremental processing, (3) incremental iterative processing, and (4) task scheduling. Those four dimensions are the main pillars for our proposed work.

In the iterative processing domain, a variety of systems have been proposed to support iterative algorithms. Pregel [4] is a distributed system for large-scale graph computing. It uses a message passing model, and keeps intermediate data in memory to process graphs efficiently. In each iteration, a vertex can update its own state and that of its outgoing edges, change the graph topology, and send messages to other vertices for use in the next iteration.

Twister [20] is a distributed in-memory MapReduce runtime. It uses a publish/subscribe mechanism to handle all communications and data transfers, and supports long-running tasks to avoid reloading data in every iteration.

Spark [6, 21] is a new computing framework that supports iterative and interactive applications. It introduces an abstraction called resilient distributed datasets (RDD) which is a read-only collection of objects partitioned across a set of machines and supports fault recovery. Spark depends on memory for fast iterative computation. It caches RDDs in memory and reuses them in multiple parallel operations (Note that: the proposed iiHadoop uses the nodes' local disk to store all generated data during iterative processing in contrast to Spark approach).

HaLoop [22] is a parallel and distributed system built on top of Hadoop to support iterative computations. It caches and indexes loop-invariant data on local disks, and makes the task scheduler assign tasks to the same nodes across multiple iterations. In addition, HaLoop introduces efficient fixpoint evaluation by comparing the current iteration output with the cached output of the previous iteration.

iMapReduce [23] improves the performance of iterative computations by eliminating the shuffle of static data, allowing asynchronous execution of map tasks, and making map/reduce tasks persistent. It keeps tasks alive during the whole iterative process to reduce task initialization overhead. In addition, iMapReduce assigns every map task and its corresponding reduce task to the same node to avoid network communication overhead.

iHadoop [24] is a modified MapReduce model built on top of HaLoop. It supports iterative computations by executing iterations asynchronously and scheduling the tasks with a producer/consumer relation on the same machine.

Maiter [25] is a general framework implemented based on Piccolo [26] to support accumulative iterative updates. It accelerates computations by iteratively accumulating the intermediate results from both the previous and current iteration rather than updating the new result with the old result. This approach allows iterative computation to be performed asynchronously and converge much faster than traditional computation. Maiter also loads and processes data in local memory of machines.

Finally, the approach proposed in [27] implements bulk and incremental iterations in Stratosphere [1]. It executes iterative algorithms efficiently by allowing only the state that needs to be modified to arrive at the iteration's result.

In the incremental processing domain, many systems have been developed to support efficient incremental computations for *one-step* applications.

Percolator [11], a system developed by Google, uses distributed transactions and notifications for processing data in an incremental manner. The application is structured as a series of observers that are triggered by the system whenever user data changes. Despite the performance benefits of this system, it uses a new programming model that differs from the one used by MapReduce.

The work introduced in [28] proposes a new data analysis platform to support incremental one-pass analytics. It replaces the sort-merge implementation in MapReduce with a purely hash-based framework that enables fast in-memory processing of the reduce function. In addition, it employs a new frequent key based technique to extend in-memory processing to workloads that require a large key-state space.

MapReduce Online [29] is an extension of the Hadoop MapReduce framework that supports online aggregation, which allows users to see "early returns" from a job as it is

being computed. It also supports continuous queries where jobs that run continuously can accept new data as it becomes available and analyze it immediately.

Incoop [13] executes a normal MapReduce job in an incremental way by automatically detecting input changes, and reusing the intermediate results from previous computations to update the outputs. It executes the map task only for the splits that have been changed, and the reduce task only for the changed intermediate results. To achieve this, it uses an incremental HDFS, performs a stable partitioning of the input data, and adds a contraction phase to divide the reduce task into smaller subtasks that can be re-used.

IncMR [14] is another framework that supports incremental processing by combining the previous intermediate results (called state) with the new data. The map tasks are created only for new data splits while the reduce tasks take their input from both the current map output and the map output of the previous computation.

Finally, Marimba [30] is a distributed framework designed for making MapReduce jobs incremental. It detects automatically the input changes and computes them using two strategies: overwrite and incremental installation. Overwrite installation reads deltas as well as the previous result as input and overwrites the old result with the calculated value. On the other hand, incremental installation reads only the deltas as input and increments the old result by adding the calculated value.

In the incremental iterative processing domain, the most recently proposed systems are Naiad and i^{2}MapReduce.

Naiad [12] is a distributed system for executing cyclic dataflow programs. It supports efficient execution of incremental and iterative computations by using a new computational model called timely dataflow that differs from MapReduce model. The timely dataflow model allows stateful computations and arbitrarily nested iterations. It distributes computations over many workers, and accelerates iterative computations as they converge. Unlike iiHadoop, Naiad is an in-memory system that relies on memory to store and index data. It is written in C# using the .NET Framework which differs from Hadoop Framework.

i^{2}MapReduce [16] performs key-value pair level incremental processing. It saves recomputation by starting from the previously converged state and performing incremental updates on the changing data. It supports both one-step and iterative computations, and uses a set of techniques to reduce I/O overhead for accessing preserved computation states. The limitation of this system is that it uses a static scheduling policy which differs than the dynamic scheduling policy used in iiHadoop.

Finally, in the task scheduling domain, many scheduling algorithms have been designed and implemented to improve Hadoop scheduling performance.

Delay scheduling [31] is a simple algorithm proposed to achieve high data locality. It allows tasks to wait for a certain amount of time when a local task cannot launch, increasing the chance of having a node with local data.

SARS [32] is another optimal scheduling policy that manages reduce tasks' start times in Hadoop. It decreases the completion time of reduce tasks by deciding the start time of each reduce task dynamically according to each job context, such as the task completion time and the size of map output.

Despite the fact that several systems were proposed in the different dimensions, in this paper we propose a new approach that handles both iterative and incremental

processing in an efficient way. The proposed approach overcomes many of the limitation of existing systems and at the same time provides many new features to enhance the overall computation. In the following section we discuss and highlight the features of the proposed iiHadoop in details.

System design

In this section, the design and implementation of iiHadoop are introduced. First, the basic idea of the proposed incremental iterative approach is outlined. Then, the significant functionalities of iiHadoop including iterative processing, incremental processing, and asynchronous iteration execution are explained. After that, the task scheduling enhancement and optimization are presented. Finally, four well-known iterative algorithms are presented to validate the effectiveness of iiHadoop, and their MapReduce implementations are described at the end of this section.

Basic idea

Incremental iterative computation means processing the changed data iteratively based on the previous computed results. iiHadoop performs incremental iterative computation on the level of key-value pairs as presented in [16]. The main idea is to save the intermediate results in initial computation, reuse them in subsequent computation, and recompute only the intermediate values that are affected by the changed input data.

Consider two different runs: the initial run A where iterative computation is performed on input data D, and a subsequent incremental run A' where the same computation is performed on new input D'. $D' = D + \Delta D$, where ΔD consists of the inserted, deleted, and updated key-value pairs (k1, v1). In initial run A, the intermediate results IR of the last iteration is preserved to be used in incremental run A'.

In the first iteration of A', the input of the Map tasks is ΔD. Since the other input key-value pairs are not changed, their Map computation would remain the same. Let ΔIR be the intermediate results of the Map tasks which includes a list of intermediate (k2, v2) pairs. The Reduce tasks recompute the intermediate values for each k2 in ΔIR. Thus, the preserved values from IR for each k2 are retrieved and merged with the corresponding values from ΔIR to obtain the updated input of Reduce tasks. The other k2 in the preserved IR is not changed and therefore would generate the same final result. Then, the Reduce tasks recompute the merged values for each affected k2 to generate the updated final results ΔFR. In the next iteration, ΔFR is now the delta input of Map tasks and is processed in the same way as the first iteration. This process is repeated until the termination condition is satisfied [16].

To prove the efficiency of this incremental iterative approach, the execution time of this approach is compared with that of recomputation from scratch. For simplicity, we assume that the execution time of each iteration $I = TM + TR$, where TM is the computation time of map phase and TR is the computation time of reduce phase. Now, we can approximate the execution time of each approach as

$$T_{recomp} \approx \sum_{i=1}^{n} (TM_i + TR_i) \tag{1}$$

$$T_{incr} \approx \sum_{i=1}^{n} (TM'_i + TR'_i) \tag{2}$$

where T_{recomp} is the execution time using the recomputation approach, T_{incr} is the execution time using the incremental approach, n is the number of iterations, and TM' and TR' are the execution time of increment map and reduce phases respectively. Note that $TM' < TM$ and $TR' < TR$.

iiHadoop accelerates the incremental iterative computation by executing iterations asynchronously (i.e., two iterations can concurrently process their data). The execution of reduce phase in iteration i is overlapped with the execution of map phase in iteration $i + 1$. Using this optimization, the execution time of increment approach will be

$$T_{asynIncr} \approx \sum_{i=1}^{n} (TM'_i + TR'_i) - \sum_{i=2}^{n} T_{overlap\,(i-1,i)} \tag{3}$$

where $T_{overlap}$ is the overlapping time between two consecutive iterations $i-1$ and i.

From the above equations, it is obvious that the execution time of incremental approach is much less than that of recomputation approach since the first processes only the delta data ΔD and ΔFR while the second processes the entire input data D' in each iteration. Moreover, the asynchronous iteration execution significantly improves the time making the proposed incremental iterative approach the best choice for this class of computation.

Iterative processing

In iterative algorithms, the same operation is performed in each iteration, and the output of previous iteration is passed as input to the next iteration. To support this paradigm, the data is separated into two types: *static data* which remains unchanged over iterations, and *dynamic data* which is updated in every iteration. For example, in PageRank algorithm, the page ranking score is the dynamic data, while the page linkage information is the static data.

The static and dynamic data are divided into multiple partitions using the same hash function. The master node assigns these partitions to the worker nodes to start the map tasks. To avoid shuffling the static data partitions in every iteration, iiHadoop keeps them stored on the local disk of the worker nodes. The map task takes the input from both the static and dynamic data; therefore, the static and dynamic data partitions are joined together using hash join.

The hash join algorithm proceeds in two phases: *the build phase* and *the probe phase*. In the *build phase*, a hash table of the dynamic data partition (small data) is built. In the *probe phase*, the algorithm scans the static data partition and finds its relevant dynamic data by looking in the hash table.

The joined data are then sent to the map tasks to process them. The dynamic data produced by the map tasks are shuffled and sent to the reduce tasks. Later, the updated dynamic data from the reduce tasks are passed to the nodes that hold their static data to start the map tasks of the next iteration. In some iterative algorithms, the map task may

need the outputs of all reduce tasks (e.g., k-means); therefore, each reduce task passes its output to all map tasks. This process is repeated in each iteration until the termination condition is satisfied.

iiHadoop preserves the intermediate and final results of the last iteration to be used later in the incremental processing, as will be described in the next section.

Incremental processing

iiHadoop performs incremental computation on the level of key-value pairs as presented in i^{2}MapReduce [16]. The input to the incremental processing is the changed data (delta) that contains the inserted, deleted and updated key-value pairs. iiHadoop processes only these data and the data that is affected by changes rather than the whole data. The delta data are represented as follows:

- The added key-value pair is represented as (k, v, +).
- The deleted key-value pair is represented as (k, v, −).
- The updated key-value pair is represented as deletion followed by addition (k, v, −), (k, v, +).

To perform the incremental processing in an efficient way, the intermediate results from the previous computation are preserved and indexed. Each record in the intermediate results is represented as (K, v2), where K uniquely identifies each intermediate record. Figure 1 shows the dataflow in the case of incremental processing.

In the first iteration, the inputs of the map tasks are the delta static data and previously converged dynamic data. The delta and dynamic data are partitioned using the same hash function, and each two partitions from the delta and dynamic data that have the same partition number are then joined using hash join. iiHadoop invokes the map function for every record in the joined data. The intermediate results from the map tasks are shuffled and merged with the preserved intermediate results before sending them to the reduce tasks.

The inputs to the merge operation are the intermediate results (IR) from the map tasks and the preserved intermediate records with the same keys that appear in IR. iiHadoop merges them as follows: the new key-value pairs are added to the preserved intermediate

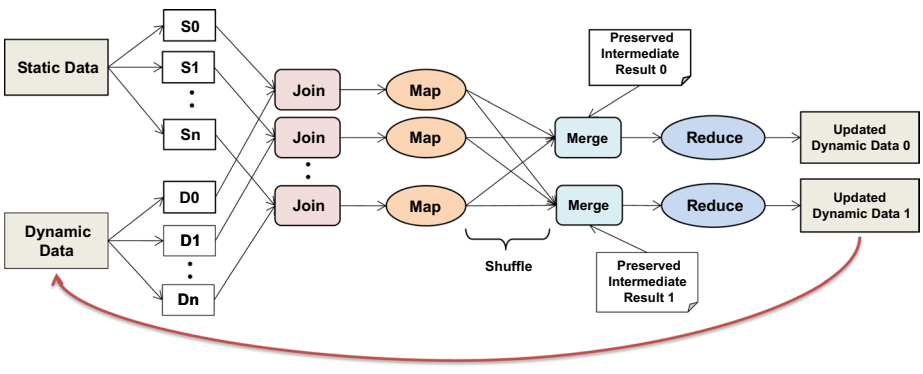

Fig. 1 Dataflow in incremental processing

records, the deleted key-value pairs are removed from the preserved intermediate records, and the updated key-value pairs replace the corresponding preserved intermediate records. After the merge operation, iiHadoop updates the preserved intermediate results file by appending the merged data to the end of the file and updating the index to reflect the new positions. As a result of iterative computations, the file may contain multiple versions of the same record; therefore, old versions are removed to reduce local disk space usage.

The reduce tasks take the updated intermediate results from the merge operation as input, recompute and update the dynamic data, and then pass them to the map tasks of the next iteration. Each iteration is processed in the same way until the termination condition is met, which will be discussed in "Termination check procedure" section.

Asynchronous iterations execution

Traditional Hadoop is not optimized to process iterative processes as a map task of next iteration cannot start its execution until all reduce tasks of previous iteration are completed. However, the map task can start its execution whenever its input data is available (i.e., without the need to wait for the completion of all reduce tasks) as long as the overall computation logic is not violated.

In this paper we propose a new approach that accelerates the execution of iterative algorithms by allowing iterations to execute in an asynchronous manner. Asynchronous iterations execution means that two iterations are allowed to process their data concurrently. Asynchronous execution is achieved by sending the output of an iteration to the next one as soon as it is produced. Consider two consecutive iterations i and $i+1$. When any reduce task of iteration i completes and produces its output, it sends the output immediately to the map tasks of iteration $i+1$. Now, there is nothing preventing the map tasks from executing since their input data are available. The map tasks of iteration $i+1$ start their execution and process their data, even while some of the reduce tasks of iteration i are still processing their input. Then, when any map task finishes, the reduce tasks of iteration $i+1$ start shuffling their intermediate data, and they execute the reduce function as soon as all map tasks finish. Figure 2 shows the execution of asynchronous iterations.

As shown in Fig. 2, after $Reduce_0$ in iteration i completes its execution, it sends the output to Map_0 of iteration $i+1$ even while $Reduce_1$ is still in execution. When Map_0 finishes processing its input, $Reduce_0$ and $Reduce_1$ start collecting intermediate data from Map_0, and start the execution of the reduce function when all map tasks of iteration $i+1$ complete their execution (e.g., Map_1).

The asynchronous execution of incremental iterative computation significantly enhances the overall performance of iiHadoop; however, this approach suffers from some limitations. First, it cannot work for some iterative algorithms where the map task needs the outputs of all reduce tasks (e.g., k-means). In this case, the map tasks of the next iteration cannot start their execution until all reduce tasks of the previous iteration are completed. Moreover, the proposed approach works efficiently on a specific storage architecture [i.e., direct attached storage (DAS)] since iiHadoop employs locality-aware scheduling and keeps the static data partitions stored on the local disk of machines to avoid shuffling and loading them in every iteration. In other architectures

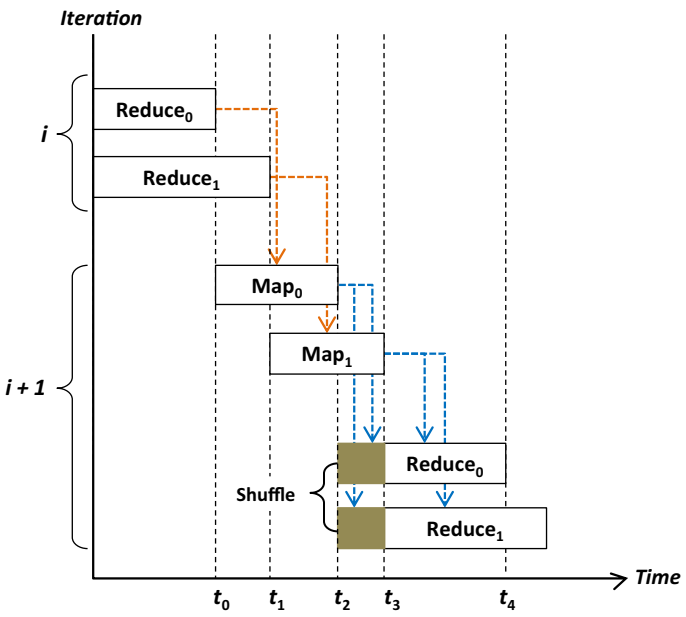

Fig. 2 Asynchronous execution

such as storage area network (SAN) and network attached storage (NAS), the proposed approach still works but suffers from additional communication overhead since these architectures require data transfer through the network. Fortunately, the transferred data is very small because iiHadoop processes only the changed data and the data that is affected by changes rather than the entire data; thus, with a high bandwidth network the communication overhead is minimal and even negligible.

Termination check procedure

Iterative algorithms process data iteratively until a termination condition is satisfied. The termination condition may be a fixed number of iterations or a user-defined distance between two consecutive iterations.

iiHadoop performs the termination check at the end of each iteration. If the number of iterations is fixed, the master node will terminate the tasks when the iterations exceed this number. On the other hand, to calculate the distance between two consecutive iterations, the reduce tasks save the results from two consecutive iterations and calculate the distance. To obtain the global distance, the master node merges the values of the local distance from each reduce task. If the global distance is less than the user-specified threshold, then the master node will terminate the tasks.

To support asynchronous execution of iterative algorithms, we assume that the termination condition will not be satisfied at every iteration; therefore, the next iteration can start even if the previous iteration is still running. When the termination condition is met, the master node sends a termination signal to the running tasks to terminate their execution. This asynchronous execution could result in wasted computation; however, it is of negligible value compared to the achieved speed up and time saving.

Task scheduling

Hadoop task scheduler assigns map and reduce tasks to the nodes taking into account node availability, cluster topology, and input data locality. However, this scheduler is not good for incremental iterative processing because it does not take into account the locality of previously computed intermediate results. In this paper, we extend Hadoop scheduling algorithm so that the location of these data is considered during the scheduling process.

Hence, in iiHadoop we implement a locality-aware task scheduler which schedules the map and reduce tasks on the nodes that hold all or most of their input data. This scheduling policy is efficient because it prevents the unnecessary movement of data and reduces the network communication overhead.

The scheduler keeps record of the location of the input data partitions and the intermediate results on each node, and accordingly uses this information to schedule the tasks. The map tasks are scheduled using the default MapReduce scheduler, which schedules the task on the node that has a local replica of its input data. The reduce tasks are also scheduled using the same policy. The scheduler assigns the reduce task to the node where most of its input or its intermediate results are stored. Algorithm 1 shows the pseudo-code for the task scheduling algorithm.

Algorithm 1 Task Scheduling

```
Input: n: node, W: wait time
Output: unlaunched task T
when a heartbeat is received from n
for each unlaunched task t do
    if t.skipped == true then
        increase t.wait
    end if
end for
if n has a free slot then
    T ← get local task in n
    if T == null then
        for each t in unlaunced tasks do
            if t.wait == 0 then
                set t.skipped = true
            else if t.wait >W then
                set T = t
                break
            else
                continue
            end if
        end for
    end if
end if
return T
```

To improve data locality, the scheduler delays the execution of the task if there are no free slots in the nodes that hold its input data [31]. The task will wait for a short period of time until a free slot is available. If still no free slot is available, the scheduler allows the task to be assigned to another node rather than the node that holds its data. In this case, its input data will be transferred over the network.

APIs

We implement iiHadoop through modifying Hadoop 1.2.1 and Hadoop 2.7.3. Some new MapReduce APIs are introduced and some existing APIs are modified to support incremental and iterative processing. We explain these APIs in this section.

To implement an iterative or incremental job, users should implement the following interfaces:

- `void map(key, staticValue, dynamicValue)` The map function has one input key and two input values. The system joins the static and dynamic data automatically.
- `void reduce(key, dynamicValue)` The reduce function interface is not changed, but the input value should be the dynamic data that is updated in each iteration.
- `float distance(key, prevValue, CurrValue)` Specifies the distance measurement using previous dynamic value and current dynamic value of a key. The returned values for different keys are accumulated to obtain the global distance value between two consecutive iterations' results.

In addition to the default job submission of Hadoop, we introduce two kinds of job submission:

- `runIterativeJob( )` run job iteratively until a termination condition is satisfied.
- `runIncrementalIterativeJob( )` run iterative job in incremental way using the preserved intermediate results and converged results of prior iterative run.

Test cases

PageRank

PageRank [8, 33] is an iterative algorithm that calculates the ranking score for web pages. It iteratively calculates the rank of each web page p based on the rank of its incoming links (the set of pages that points to p). The algorithm stops when the termination condition is met (e.g., a specified number of iterations is performed).

In this paper, the MapReduce implementation of the PageRank algorithm proposed in [16] is modified as shown in Algorithm 2. The inputs of the map function are the page id p as the key, and its linkage set N_p and ranking score R_p as the value. The map function divides the rank of page among its links evenly, and emits records with the id l and rank contribution R_l for each link. The reduce function accumulates the rank contributions of each incoming link to the page, and emits records with the page id and its updated ranking score. The updated ranking score and the linkage set of each page are then passed as input to the next iteration.

Algorithm 2 PageRank in MapReduce

Map Phase
Input: $<p, N_p|R_p>$
Output: $<l, N_l, \{R_l\}>$
1: **for** all *l* in N_p **do**
2: $R_l = \frac{R_p}{|N_p|}$
3: return $<l, R_l>$
4: **end for**
5: return $<p, N_p>$
Reduce Phase
Input: $<l, N_l, \{R_l\}>$
Output: $<l, N_l|R_l>$
6: $R_l = d\sum R_l + (1 - d)$
7: return $<l, N_l|R_l>$

Figure 3a shows the iterative processing of PageRank in iiHadoop. In the first iteration, the linkage data and ranking score data are partitioned and joined before the execution of map tasks. The ranking score data from the map tasks are updated in the reduce task and then passed to the next iteration. In this example, after three iterations (i.e., the stopping condition), the final ranking scores are calculated and preserved with the intermediate results to be used in the incremental processing, as shown in Fig. 3b.

The converged ranking scores and changed linkage data are the inputs for incremental processing, and only one map task is required to process these data. The intermediate results from the map task are sent to the reduce task after the merge operation. As shown in the example, the changes in the linkage data affect only a subset of the intermediate results; therefore, the affected records from the preserved intermediate results (records with the keys 2, 3, 4, 5) are retrieved and merged with the intermediate results from the map task. Then the reduce function re-computes the ranking score of the updated intermediate results. In the next iteration, the ranking scores of pages (2, 3, 4, 5) are joined with their linkage data and processed in the same way as the first iteration until the number of iterations exceeds three.

Single source shortest path

Single source shortest path (SSSP) [10] is the problem of finding the shortest distance from a source node to all other nodes in a graph. Given a weighted, directed graph $G = (V, E, W)$, where V is the set of nodes, E is the set of edges, and $W(i, j)$ is the weight of edge from node i to node j. The shortest distance from the source node s to each node u in the graph is computed iteratively using Dijkstra's algorithm [34]. Initially, the distance of the source node is set to 0 (i.e., $d(s) = 0$), and the minimum distance from s to any other nodes is set to ∞ (i.e., $d(u) = \infty$).

The shortest path iterative computation can be implemented in the MapReduce model as shown in Algorithm 3. The input key of the map function is the node id u, and the input value is composed of the weights of its outgoing links $W(u, *)$, and its minimum distance $d(u)$. For each node v in the outgoing links set, the map function computes the distance and outputs a key-value pair of the node id v and the received distance $d(v)$. Then, the reduce function selects the minimum value between the node current distance $d_{curr}(v)$ and all received distance, and updates $d(v)$ with the minimum value.

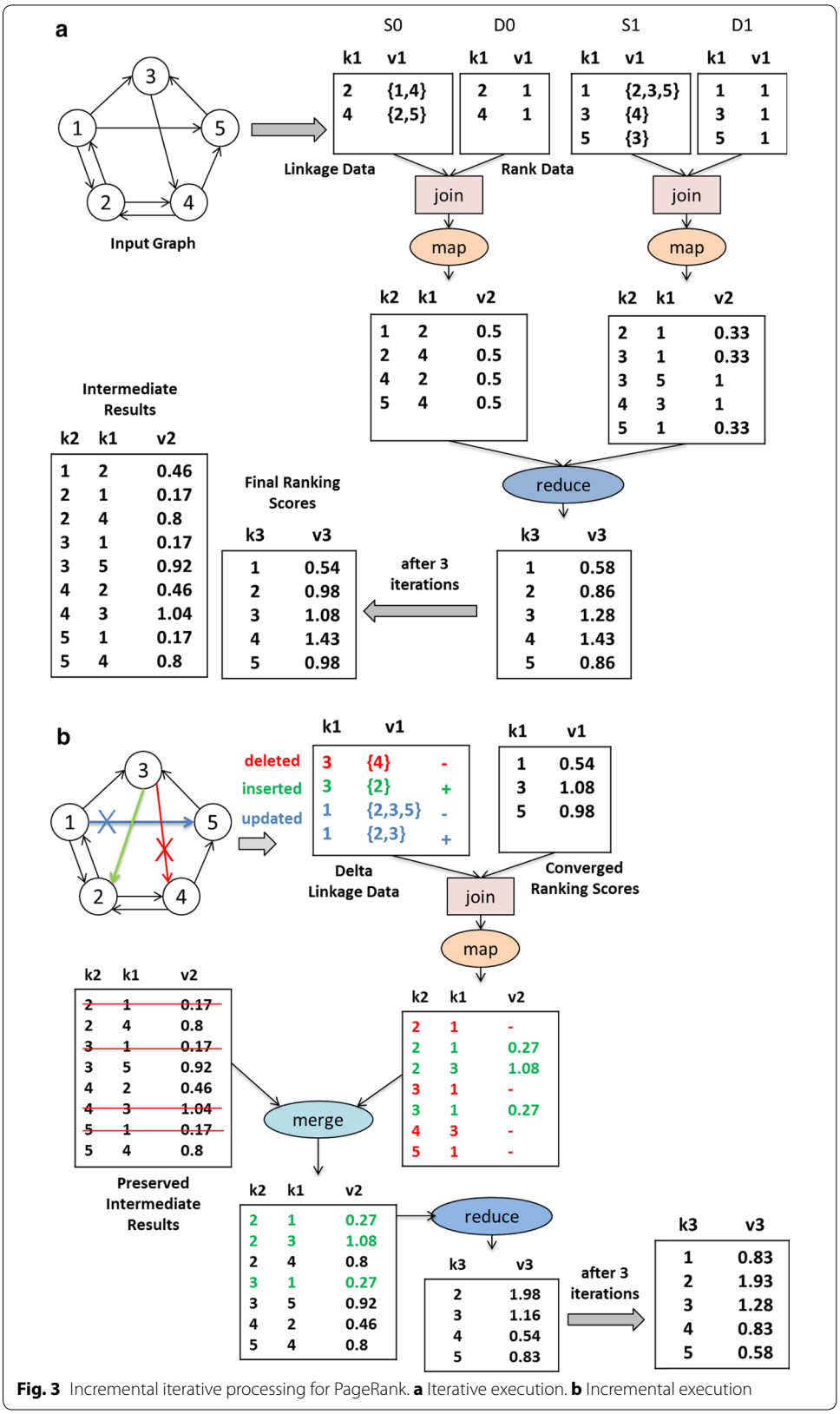

Fig. 3 Incremental iterative processing for PageRank. **a** Iterative execution. **b** Incremental execution

Algorithm 3 SSSP in MapReduce

Map Phase
Input: $<u, W(u, *)|d(u)>$
Output: $<v, W(v, *), \{d(v)\}>$
1: **for** all v in the outgoing links set **do**
2: $d(v) = d(u) + W(u,v)$
3: return $<v, d(v)>$
4: **end for**
5: return $<u, W(u, *)>$, $<u, d_{curr}(u)>$
Reduce Phase
Input: $<v, W(v, *), \{d(v)\}>$
Output: $<v, W(v, *)|d(v)>$
6: $d(v) \leftarrow \min \{d_{curr}(v), \{d(v)\}\}$
7: return $<v, W(v, *)|d(v)>$

Connected components

Connected components [35] is an algorithm for finding the connected components in a large graph. In an undirected graph, each node is associated with a component id, which is initially its own node id. Then, each node propagates its current component id to its neighbors, and updates its own component id by selecting the minimum value between its current component id and the received component ids from its neighbors. This process is repeated until no node in the graph updates its component id.

The iterative MapReduce implementation of the connected components algorithm is shown in Algorithm 4. In each iteration, the input of the map function is the node id v, its neighbors set N_v, and its component id C_v. The map function propagates the component id of each node to its neighbors, and emits records with the node id w and the component id C_w for each neighboring node. The reduce function updates the component id of each node, and outputs key-value pairs of the node id and its updated component id.

Algorithm 4 Connected Components in MapReduce

Map Phase
Input: $<v, N_v|C_v>$
Output: $<w, N_w, \{C_w\}>$
1: **for** all w in N_v **do**
2: $C_w \leftarrow C_v$
3: return $<w, C_w>$
4: **end for**
5: return $<v, N_v>$, $<v, C_{vcurr}>$
Reduce Phase
Input: $<w, N_w, \{C_w\}>$
Output: $<w, N_w|C_w>$
6: $C_w \leftarrow \min \{C_{wcurr}, \{C_w\}\}$
7: return $<w, N_w|C_w>$

k-means clustering

K-means [36, 37] is another popular iterative algorithm that partitions points into k clusters, so that the points in the same cluster are more similar to each other than those points in other clusters.

The algorithm starts by selecting k random points as cluster centroids set. Then it assigns each point to the nearest cluster centroid. After that, it updates the k cluster centroids by calculating the average value of points that belong to the same cluster centroid. The last two steps are repeated until convergence is reached (i.e., a specified number of

iterations are performed or the number of points that move between clusters is less than a threshold).

In k-means, the points set is the static data and centroids set is the dynamic data. In each iteration, the map function takes point id (*pid*), point coordinate (*pcoord*), and cluster centroids set {*cid*, *ccoord*} as input, and outputs the closet cluster id (*cid*) along with the point coordinate (*pcoord*). The reduce function updates the cluster centroid coordinate by calculating the average of all points coordinates that belong to it. The updated centroids set is then sent to all map functions of the next iteration. Algorithm 5 shows a modified MapReduce implementation of the k-means algorithm proposed in [16].

Algorithm 5 k-means in MapReduce

Map Phase
Input: <*pid*, *pcoord*|{*cid*, *ccoord*}>
Output: <*cid*, *pcoord*>
1: *cid* ← find the nearest centroid of *pcoord* in {*cid*, *ccoord*}
2: return <*cid*, *pcoord*>
Reduce Phase
Input: <*cid*, {*pcoord*}>
Output: <*cid*, *ccoord*>
3: *ccoord* ← compute the average of {*pcoord*}
4: return <*cid*, *ccoord*>

Experimental evaluation

In this section, we evaluate the performance of iiHadoop using the four iterative algorithms discussed in the previous section namely, PageRank, SSSP, connected components, and k-means. We compare the performance of iiHadoop to traditional Hadoop implementation [7], HaLoop implementation [22], and i^{2}MapReduce implementation [16]. In addition, since users have the option to turn on/off the asynchronous execution in iiHadoop, we compare Async-iiHadoop with Sync-iiHadoop to see the performance improvement from using asynchronous iteration execution.

Experimental setup

Cluster environment

The experiments are conducted on a virtual machine cluster built on Amazon EC2 cloud. The cluster consists of 31 nodes where each node has the following specifications: 4 GB of memory, 2 compute unit, 150 GB of storage, High Frequency Intel Xeon Processor, and runs 64-bit Linux operating system. A node in the cluster is always designated as a master node, and all experimental results are obtained without any node failures.

By default, Hadoop block size is set to 128 MB, and Hadoop heap size is set to 2 GB. For all systems except i^{2}MapReduce, the number of reduce tasks is set to the number of the slave nodes in the cluster, and the maximum number of map and reduce tasks per node is set to 4 and 2 respectively. i^{2}MapReduce creates the maps and reduce tasks at the first iteration, and keeps them alive across multiple iterations. In addition, its scheduling policy requires the map and reduce tasks to be the same for many algorithms including: PageRank, SSSP, and connected components. Therefore, the number of reduce tasks and the maximum number of map and reduce tasks per node varies depending on the size of input data.

Datasets

We generate four synthetic massive datasets for the four algorithms using two real-world data sets: ClueWeb Category B [38] and BigCross [39]. ClueWeb Category B consists of 428,136,613 nodes, 454,075,638 edges, and its total size is 4.8 GB. BigCross consists of 11,620,300 individuals and each is with 57 dimensions, and its total size is 1.6 GB. The four generated datasets are described in Table 1.

The **ClueWeb** dataset is a semi-synthetic dataset generated from the ClueWeb Category B dataset. The original dataset was very small. Thus, we randomly generated new pages each with 10 outgoing links, and added them to the original dataset graph structure to make it larger. The total size of the new dataset is 30 GB.

The **ClueWeb1** dataset is another semi-synthetic dataset generated from the original ClueWeb dataset. Since the graph for the SSSP algorithm are directed and weighted, and the ClueWeb Category B graph is not originally weighted, we assigned random weight to each edge in the graph. The resulted dataset is 10 GB.

The **ClueWeb2** dataset is also generated from the ClueWeb Category B dataset. The connected components algorithm takes undirected graph as input, so we simply made the directed ClueWeb graph undirected through adding an inverse direction for each directed link. The total size of the ClueWeb2 dataset is 12 GB.

The **BigCross** dataset is a semi-synthetic dataset generated from the real-world BigCross dataset. Since the total size of the original dataset is 1.6 GB, we make it larger by repeating the original dataset nine times. The generated dataset is 16 GB. We randomly pick 60 points from the whole dataset and use them as initial centers.

For incremental processing, a delta input was generated from each dataset. The delta input is generated by randomly changing a small portion of the input data.

Performance evaluation

In this set of experiments, maximum number of iterations is used rather than threshold distance to specify the termination conditions. In addition, to make the experimental results more accurate, each experiment is repeated 3 times and the average of the results obtained is used. For all conducted experiments, the execution time of a job is the key factor and main measure to assess the effectiveness of iiHadoop.

Iterative computation

Figure 4 shows the runtime of the four algorithm on the traditional Hadoop, HaLoop, i^{2}MapReduce, Iter-iiHadoop, and Async-Iter-iiHadoop.

For PageRank, the performance of iiHadoop and other systems in the first iteration is approximately the same. The reason is that iiHadoop performs additional work in

Table 1 Datasets used in the experiments

Algorithm	Dataset	Size (GB)	Description
PageRank	ClueWeb	30	616,516,725 pages, 2,903,017,060 edges
SSSP	ClueWeb1	10	428,136,613 pages, 454,075,638 edges
Connected components	ClueWeb2	12	428,136,613 pages, 530,014.595 edges
K-means	BigCross	16	104,582,700 points, 57 dimensions

the first iteration. It caches the static data on each Mapper local disk, and caches the grouped data on each Reducer local disk. As shown in Fig. 4a, iiHadoop performs better than Hadoop and HaLoop systems. It outperforms Hadoop and HaLoop with and without using the asynchronous execution, and achieves about 2 times speedup. The runtime is mainly saved by avoiding static data shuffling. On the other hand, it is surprising to see that HaLoop performs worse than the traditional Hadoop. This is because HaLoop runs an extra MapReduce job in each iteration to join the static data with the dynamic data. iiHadoop avoids this overhead by automatically partitioning and joining the static and dynamic data using hash join algorithm.

In addition, iiHadoop performs better than i^{2}MapReduce through performing asynchronous iteration execution, which reduces the running time to 23% of i^{2}MapReduce, as shown in Fig. 4a. The performance gain also comes from the number of the running reduce tasks in each systems. iiHadoop runs 30 reduce tasks in each iteration (i.e., one reduce task per node) which is a good configuration for the given nodes specification. On the other hand, i^{2}MapReduce requires the number of map and reduce tasks to be the same; therefore, it runs 232 reduce tasks (i.e., 8 reduce task per node). This configuration adds heavy load to the node, and as a result, the performance of each node with i^{2}MapReduce is slower than its performance with iiHadoop.

For SSSP and connected components, the iterative behavior of iiHadoop is similar to that for PageRank algorithm; therefore, the performance improvement is also the same as shown in Fig. 4b, c.

For k-means, the asynchronous execution functionality is not used because each map task in the algorithm needs all reduce tasks' data. Fig. 4d shows the runtime of k-means algorithm on all comparable systems. It is clear from the figure that HaLoop, i^{2}MapReduce, and iiHadoop exhibit similar behavior. They outperform Hadoop because of their similar optimization mechanisms such as caching static data. Nevertheless, iiHadoop achieved speedup for k-means is less significant than the speedup achieved for PageRank, SSSP, and connected components. This result is expected since the implementation of k-means needs to shuffle the static data in each iteration and cannot benefit from the asynchronous execution feature.

Incremental iterative computation

Figure 5 shows the runtime of the four iterative algorithms on Hadoop, HaLoop, i^{2}MapReduce, Incr-iiHadoop, and Async-Incr-iiHadoop when 10% of input data is changed.

For PageRank, SSSP, and connected components, iiHadoop reduces the runtime of computation by approximately 60% compared to Hadoop and HaLoop, as shown in Fig. 5a–c. This time-saving behavior comes from the caching of static data as discussed in the previous section. In addition, Hadoop and HaLoop start the iterative computation from scratch on the whole data, while iiHadoop only applies the iterative computation on the changed data which represents a small fraction of data. On the other hand, iiHadoop outperforms i^{2}MapReduce because of the performance gain from the asynchronous execution, and the optimized scheduler that reduces the communications overhead by assigning tasks to the nodes with local data.

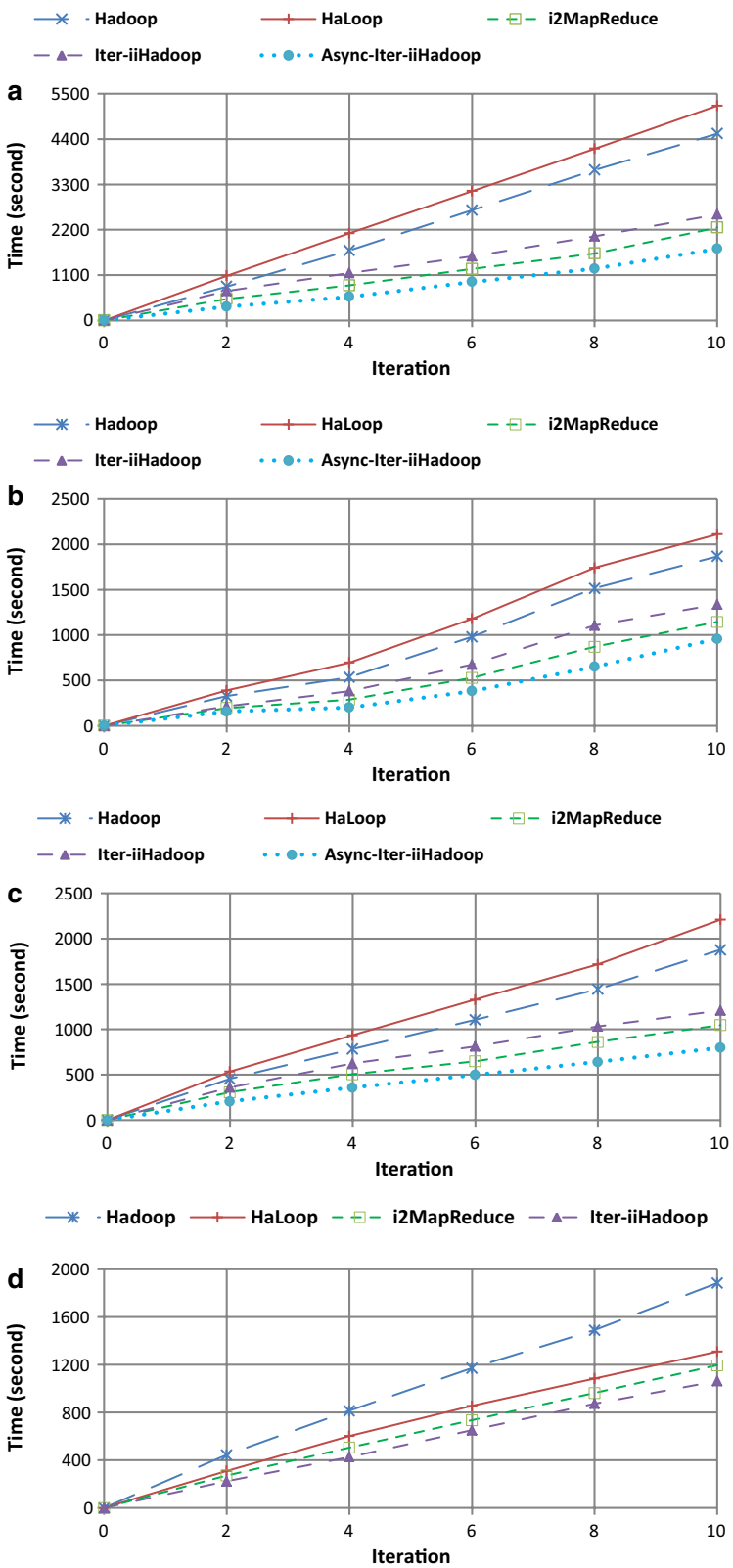

Fig. 4 Total running time of iterative processing. **a** PageRank. **b** SSSP. **c** Connected components. **d** K-means

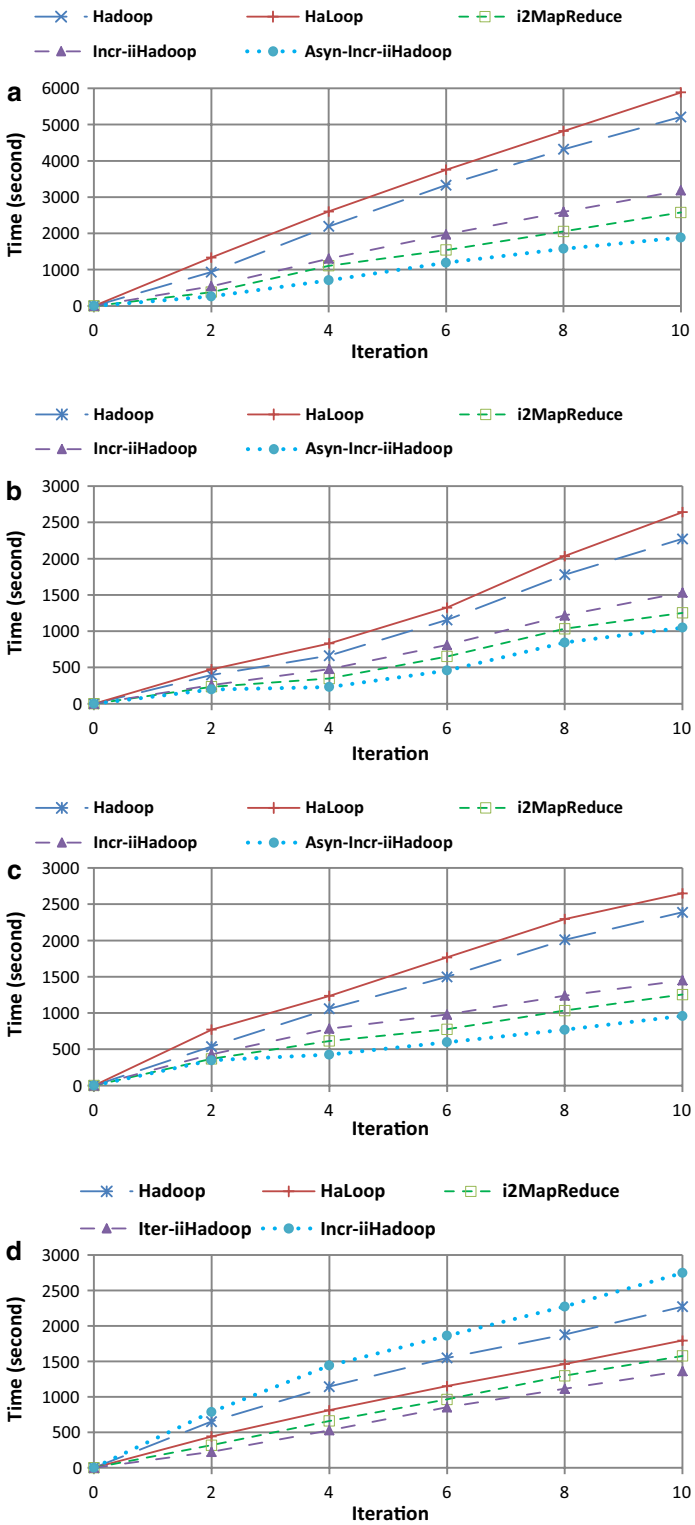

Fig. 5 Total running time of incremental processing. **a** PageRank. **b** SSSP. **c** Connected components. **d** K-means

For k-means, we compare the performance of Hadoop, HaLoop, i^{2}MapReduce, Iter-iiHadoop, and Incr-iiHadoop. As shown in Fig. 5d, the performance of HaLoop, i^{2}MapReduce, and Iter-iiHadoop is similar, while the performance of Incr-iiHadoop is worse than Hadoop. The small portion of changes in input data led to a total re-computation in Incr-iiHadoop which increases the overall runtime. The reason for this increase is that the proposed incremental approach needs to retrieve and update the preserved intermediate results in each iteration which consumes a lot of time. Therefore, we allow users to turn on/off the use of preserved intermediate results to be able to run the k-means algorithm using the proposed iterative mechanism rather than the incremental mechanism because it is not suitable for this situation.

Implementation of iiHadoop on YARN

In this section, we evaluate the performance of iiHadoop implemented on YARN. To see the effect of YARN on the proposed approach, we implement iiHadoop through modifying Hadoop 2.7.3. The iterative and incremental runtime of the four algorithms on iiHadoop 1.2.1 and iiHadoop 2.7.3 are shown in Fig. 6.

As seen in the previous section, iiHadoop 1.2.1 reduces the run time of iterative and incremental computations by approximately 23% compared to i^{2}MapReduce. Implementing the proposed approach on YARN leads to additional improvements in the performance of iiHadoop. As shown in Fig. 6, iiHadoop 2.7.3 reduces the run time of iterative and incremental computations by approximately 17 and 19% respectively compared to iiHadoop 1.2.1. These additional improvements come from the improved architecture of YARN that enhances the proposed approach as follow:

First, in iiHadoop 1.2.1, scheduling map/reduce tasks in each iteration, tracking their progress, and performing the termination check at the end of each iteration is all done by the JobTracker. Using YARN allows the proposed approach to split these major functionalities between the ResourceManager and ApplicationMaster. In each iteration, the ResourceManager uses the proposed scheduling algorithm to schedules the resource containers that run map or reduce tasks as requested by the ApplicationMaster. Then, the ApplicationMaster tracks the status of these containers, and performs the termination check procedure. This improvement lightens the load on the master node and enhances the overall run time of iiHadoop 2.7.3.

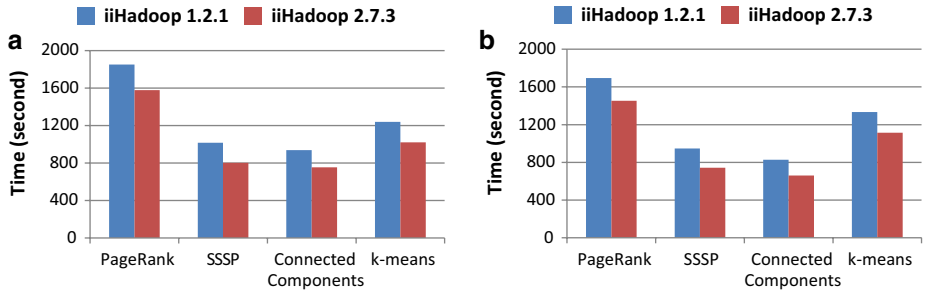

Fig. 6 Total running time of iiHadoop on YARN. **a** Iterative processing. **b** Incremental processing

Second, iiHadoop 1.2.1 uses fixed number of map and reduce slots for each TaskTracker. These configurations prevent the idle map slots from being used to run reduce tasks and vice versa. The NodeManager in YARN has a number of containers each of which can run a map or reduce task as requested. These dynamic configurations enable iiHadoop 2.7.3 to run more map tasks in parallel in each iteration compared to iiHadoop 1.2.1 (i.e., fixed number of map tasks run in parallel) which improves the run time of map phase by approximately 27% with respect to the map run time and 8% with respect to the total job execution time.

Finally, Hadoop 2.x made significant improvements in the MapReduce runtime itself. For example, the sort and shuffle implementation of Hadoop 2.x is improved compared to Hadoop 1.x. These improvements affect the run time of shuffle and reduce phases of iiHadoop 2.7.3 in each iteration. It reduces the run time of shuffle phase by approximately 24% compared to iiHadoop 1.2.1 shuffle time and 3% compared to the total job execution time. In addition, these significant improvements enhance the run time of reduce phase by approximately 17 and 6% compared to the reduce time and total job execution time respectively.

Different stages of computation

To better understand the performance gains that iiHadoop achieves over comparable frameworks, the execution time of PageRank algorithm across 10 iterations is broken down into different stages, namely, data loading, join, map, shuffle, and reduce, as shown in Fig. 7.

For the data loading stage, iiHadoop, i^{2}MapReduce, and HaLoop improve the run time by separating the input data into static and dynamic data, loading the static data from HDFS in the first iteration, and caching them on the machine's local disks to avoid loading them in every iteration. On the other hand, Hadoop treats each iteration as a separate job and hence, Hadoop needs to load the whole input data in every iteration.

For the join stage, HaLoop runs an extra MapReduce job in each iteration to join the static data with the dynamic data which consumes a lot of time. iiHadoop and i^{2}MapReduce automatically join the static and dynamic data before invoking of the map function so the join stage is part of the map task. Hadoop does not separate data into two types; therefore, there is no join stage.

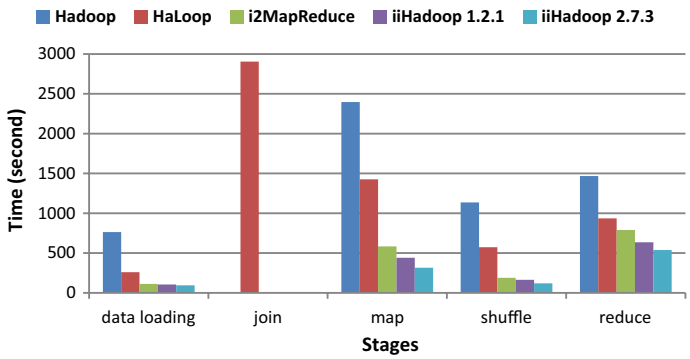

Fig. 7 Running time of different stages in PageRank algorithm

For the map stage, iiHadoop and i^{2}MapReduce reduce the run time by processing only the changed data which represents a small fraction of data, while Hadoop and HaLoop start the iterative computation from scratch on the entire dataset. iiHadoop further improves the performance by efficiently overlapping the time of the reduce and map tasks of two consecutive iterations, and using Hash Join algorithm to join the static and dynamic data as discussed in "Iterative processing" section. The hash join technique is more efficient than the join mechanism of i^{2}MapReduce which reads the record from the sorted static partition file and the sorted dynamic partition file and then joins them together. On top of that, YARN improves the run time of iiHadoop 2.7.3 due to significant improvements in the resource utilization.

For the shuffle stage, the main reason behind the time saving of HaLoop, i^{2}MapReduce, and iiHadoop is that those systems avoid the shuffling of static data. In addition, iiHadoop and i^{2}MapReduce further reduce the shuffle time by shuffling only the intermediate results that are affected by input changes. Also the improvements in the MapReduce model of Hadoop 2.x enhances the shuffle time of iiHadoop 2.7.3.

Finally, for the reduce stage, iiHadoop and i^{2}MapReduce consume some time to merge the intermediate key-value pairs with the preserved intermediate results; however, they still achieve better performance compared to Hadoop and HaLoop. Moreover, iiHadoop outperforms i^{2}MapReduce because it runs the right number of reduce tasks in every node while i^{2}MapReduce requires the number of map and reduce tasks to be the same and the infrastructure should accommodate all the long-running tasks simultaneously which adds heavy load to each node.

For SSSP and connected components, the behavior of iiHadoop in different stages is similar to that for PageRank algorithm; therefore, the performance improvement is also the same. The data loading time is affected by the size of data. These algorithms need less time to load data compared to PageRank since the size of their datasets is less than that of PageRank as shown in Table 1. For k-means, the shuffle and reduce stages cannot benefit from the above improvements since the implementation of k-means needs to shuffle the static data in each iteration which increases the shuffle time as Hadoop. In addition, the total re-computation in incremental k-means increases the merge time in reduce stage and as a result leads to a large increase in the runtime of the reduce phase.

Scalability tests

To examine the scalability of iiHadoop, the PageRank algorithm is executed on the ClueWeb dataset for five iterations in the following two experiments.

In the first experiment, we examined the performance of iiHadoop when varying the input size. The PageRank algorithm is executed on additional three datasets generated from ClueWeb dataset with a total size of 10, 20, and 40 GB respectively. The runtime results of Hadoop, HaLoop, i^{2}MapReduce, iiHadoop 1.2.1, and iiHadoop 2.7.3 are shown in Fig. 8a. The results show improvement in iiHadoop performance when the size of data increases.

In the second experiment, we compared the performance of iiHadoop 1.2.1, iiHadoop 2.7.3, Hadoop, HaLoop, and i^{2}MapReduce when varying the number of nodes in the cluster. For all systems, one master node is always used, and the number of reduce tasks is set to the number of slave nodes except i^{2}MapReduce which requires the number of

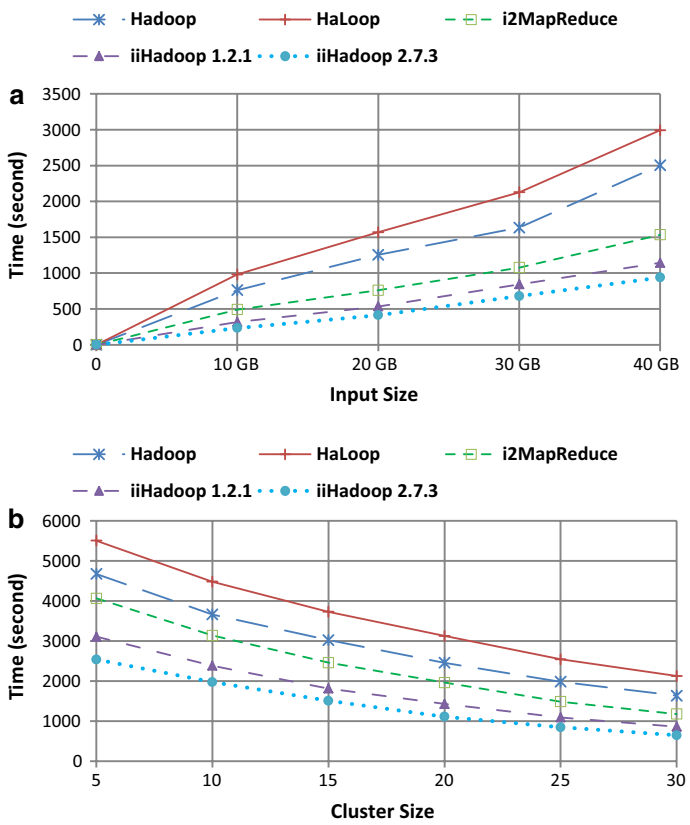

Fig. 8 Scalability tests. **a** Variable input size. **b** Variable cluster size

reduce tasks to be equal to the number of map tasks. As shown in Fig. 8b, iiHadoop performs better when the number of nodes increases. These two experiments show that iiHadoop scales as well as Hadoop, HaLoop, and i^{2}MapReduce.

Different change percent

This experiment is conducted to compare the performance of the proposed iterative and incremental mechanisms in iiHadoop when varying the incremental change percent. The connected components algorithm is executed for 5 iterations on the ClueWeb2 dataset when 5, 10, 15, 20, and 25% of input data are changed. Figure 9 shows the results of Async-Iter-iiHadoop and Async-Incr-iiHadoop.

You can see that with the small increase in incremental change percent which ranges from 5 to 15%, Incr-iiHadoop achieves substantial performance gains and speedup, and shows an obvious advantage over Iter-iiHadoop. The acceleration becomes apparent because a small fraction of input data is re-computed. On the other hand, as the size of incremental change becomes larger (i.e., 20% and more), the achieved gain and speedup of Incr-iiHadoop decreases because larger changes lead to more re-computations. This is expected since with large changes, the delta data size is increased and the proportion of the entire data that is affected by the delta input is increased as well.

We can conclude that with larger incremental changes, it is better to process data using the iterative mechanism rather than the incremental mechanism because retrieving and

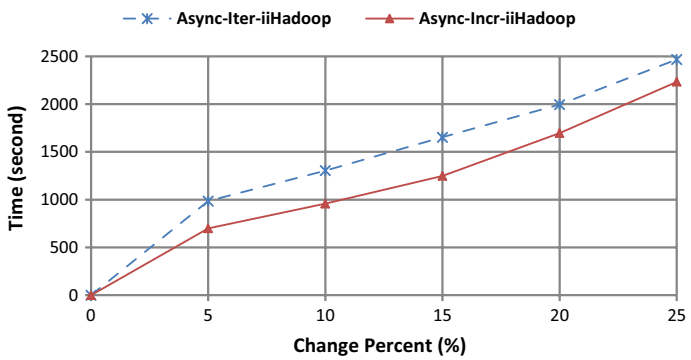

Fig. 9 Total running time with different incremental change

updating the preserved intermediate result in incremental processing will consume more time than starting the computations from scratch.

Multiple jobs submission

In this experiment, we evaluate the performance of iiHadoop and i^{2}MapReduce when two jobs are submitted at the same. The first job executes the k-means algorithm, and the second job executes the SSSP algorithm. For this experiment, the maximum number of map and reduce tasks per node is set to 5, so the total number of map/reduce tasks that can execute simultaneously on the cluster is equal to 150. The other configurations of each system are described in Table 2.

Figure 10 illustrates the execution behavior and total running time for each job when executed using iiHadoop and i^{2}MapReduce. It is clear that iiHadoop can run the two job concurrently because iiHadoop scheduler reserves a free slot to launch each map/reduce task, and releases it immediately when the task is completed. This behavior allows each running job to have a fair portion of the computing resources, and utilizes the resources in an advanced and efficient way. On the other hand, i^{2}MapReduce scheduler allows tasks to stay alive across multiple iterations, so the reserved slots are released only at the end of job. As seen in Fig. 10, most of the resources are occupied by the first job (i.e., k-means Job) and the other job is not able to start until the first job is completed.

Conclusion and future work

Many applications need executing iterative algorithms on incremental data. In this paper, we introduce iiHadoop, a MapReduce based framework for efficient incremental iterative computation over large-scale datasets. iiHadoop mainly works by performing computations over a small subset of data that is affected by input changes instead of

Table 2 Configurations of jobs

Job name	Size (GB)	iiHadoop		i^{2}MapReduce	
		Map	Reduce	Map	Reduce
k-means job	16	256	30	256	30
SSSP job	10	160	30	160	160

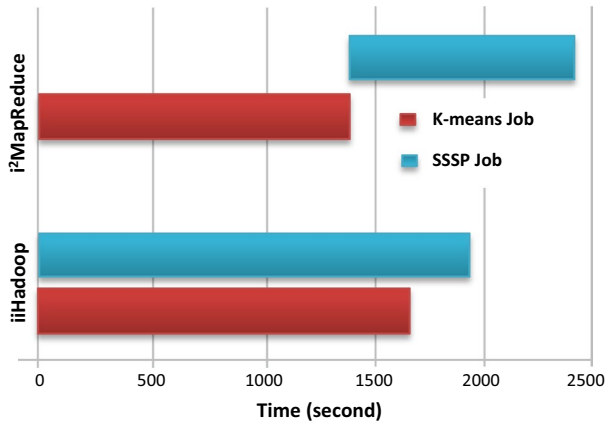

Fig. 10 Concurrent jobs execution

computing over the entire dataset. iiHadoop proposes several optimizations to reduce the execution time and improving the performance of incremental iterative computation. In particular, iiHadoop introduces more parallelism by executing iterations asynchronously where two iterations can concurrently process their data. It also employs an efficient locality-aware scheduler that takes the location of input data and previously computed intermediate results into account; in addition, it implements a delay scheduling technique.

Several experiments are conducted on Amazon EC2 cloud to evaluate iiHadoop performance using real and synthetic datasets. The results of a set of iterative and incremental algorithms indicate that the system overall performance and scalability is greatly improved through the above optimizations. The results demonstrate that iiHadoop improves the runtime substantially and achieves significant performance gains compared to that achieved by Hadoop and HaLoop. iiHadoop also achieves satisfactory improvements in execution time of the PageRank, SSSP, and connected components algorithms compared to i^{2}MapReduce. In addition, YARN achieves an advanced utilization of resources which affects the performance of iiHadoop and improves it significantly.

For future work, there is still more work that can be conducted to optimize iiHadoop to support fast in-memory processing, and comparing it with other in-memory systems such as Spark and Naiad. In addition, improving iiHadoop to support iterative and incremental processing for specific class of applications that employ recursively reducible jobs still needs further investigation. Finally, applying the proposed optimizations to Spark, conducting more experiments using more complex algorithms, and determining the optimal settings for iiHadoop through exploring different measures are interesting future research directions.

Authors' contributions

This paper is part of requirements toward the MSc degree in Information Systems with the Faculty of Computers and Information, Cairo University for student Afaf G. Bin Saadon under supervision of Dr. Hoda M. O. Mokhtar. Both authors read and approved the final manuscript.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The datasets supporting the conclusions of this article are available in [The ClueWeb09 Category B Dataset. http://www.lemurproject.org/clueweb09/webGraph.php], and [The BigCross Dataset. http://www-old.cs.uni-paderborn.de/en/research-group/ag-bloemer/research/abgeschlossene/clustering/streamkmpp.html].

The systems used in the experiments are:

Apache Hadoop 1.2.1. http://hadoop.apache.org/docs/r1.2.1/.

Apache Hadoop 2.7.3. https://hadoop.apache.org/docs/r2.7.3/.

HaLoop. https://code.google.com/archive/p/haloop/.

i^{2}MapReduce. https://code.google.com/archive/p/incr-iter-hadoop/.

Publisher's Note

Received: 4 February 2017 Accepted: 18 July 2017
Published online: 24 July 2017

References

1. Battré D, Ewen S, Hueske F, Kao O, Markl V, Warneke D. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. Proceedings of the 1st ACM Symposium on cloud computing. New York: ACM; 2010. p. 119–30.
2. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
3. Isard M, Budiui M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Op Syst Rev. 2007;41(3):59–72.
4. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. Proceedings of the 2010 ACM SIGMOD International Conference on management of data. New York: ACM; 2010. p. 135–46.
5. Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson Ú, Gunda PK, Currey J. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. Proceedings of the 8th USENIX Conference on operating systems design and implementation. Berkeley: USENIX Association; 2008. p. 1–14.
6. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. Proceedings of the 2Nd USENIX Conference on hot topics in cloud computing. Berkeley: USENIX Association; 2010. p. 10.
7. Apache Hadoop. http://hadoop.apache.org/. Accessed 26 Dec 2016.
8. Brin S, Page L. The anatomy of a large-scale hypertextual web search engine. Comput Netw ISDN Syst. 1998;30(1–7):107–17.
9. Chu C, Kim SK, Lin YA, Yu Y, Bradski G, Ng AY, Olukotun K. Map-reduce for machine learning on multicore. Proceedings of the 19th International Conference on neural information processing systems. Cambridge: MIT Press; 2006. p. 281–8.
10. Cormen TH, Leiserso CE, Rivest RL, Stein C. Introduction to algorithms. 3rd ed. Cambridge: MIT Press; 2001.
11. Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. Proceedings of the 9th USENIX Conference on operating systems design and implementation. Berkeley: USENIX Association; 2010. p. 251–64.
12. Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M. Naiad: a timely dataflow system. Proceedings of the Twenty-Fourth ACM Symposium on operating systems principles. New York: ACM; 2013. p. 439–55.
13. Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R. Incoop: Mapreduce for incremental computations. In: Proceedings of the 2nd ACM Symposium on cloud computing. New York: ACM; 2011. p. 7:1–7:14.
14. Yan C, Yang X, Yu Z, Li M, Li X. Incmr: Incremental data processing based on mapreduce. In: 2012 IEEE Fifth International Conference on cloud computing. New York: IEEE; 2012. p. 534–41.
15. Zhang Y, Chen S. i2mapreduce: Incremental iterative mapreduce. In: Proceedings of the 2Nd International Workshop on cloud intelligence. New York: ACM; 2013. p. 3:1–3:4.
16. Zhang Y, Chen S, Wang Q, Yu G. i2mapreduce: incremental mapreduce for mining evolving big data. IEEE Trans Knowl Data Eng. 2015;27(7):1906–19.
17. Borthaku D. The hadoop distributed file system: architecture and design. Hadoop Project Website. 2007;11:1–14.
18. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E. Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on cloud computing. ACM: New York; 2013. p. 5:1–5:16.
19. Iteration. https://en.wikipedia.org/wiki/Iteration. Accessed 2 Dec 2016.
20. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G. Twister: a runtime for iterative mapreduce. Proceedings of the 19th ACM International Symposium on high performance distributed computing. New York: ACM; 2010. p. 810–8.
21. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Conference on networked systems design and implementation. Berkeley: USENIX Association; 2012. p. 2.
22. Bu Y, Howe B, Balazinska M, Ernst MD. The haloop approach to large-scale iterative data analysis. VLDB J. 2012;21(2):169–90.
23. Zhang Y, Gao Q, Gao L, Wang C. imapreduce: a distributed computing framework for iterative computation. J Grid Comput. 2012;10(1):47–68.

24. Elnikety E, Elsayed T, Ramadan HE. ihadoop: asynchronous iterations for mapreduce. Proceedings of the 2011 IEEE Third International Conference on cloud computing technology and science. Washington: IEEE; 2011. p. 81–90.
25. Zhang Y, Gao Q, Gao L, Wang C. Accelerate large-scale iterative computation through asynchronous accumulative updates. Proceedings of the 3rd Workshop on scientific cloud computing date. New York: ACM; 2012. p. 13–22.
26. Power R, Li J. Piccolo: building fast, distributed programs with partitioned tables. Proceedings of the 9th USENIX Conference on operating systems design and implementation. Berkeley: USENIX Association; 2010. p. 293–306.
27. Ewen S, Tzoumas K, Kaufmann M, Markl V. Spinning fast iterative data flows. Proc VLDB Endow. 2012;5(11):1268–79.
28. Li B, Mazur E, Diao Y, McGregor A, Shenoy P. A platform for scalable one-pass analytics using mapreduce. Proceedings of the 2011 ACM SIGMOD International Conference on management of data. New York: ACM; 2011. p. 985–96.
29. Condie T, Conway N, Alvaro P, Hellerstein JM, Elmeleegy K, Sears R. Mapreduce online. Proceedings of the 7th USENIX Conference on networked systems design and implementation. Berkeley: USENIX Association; 2010. p. 21–21.
30. Schildgen J, Jörg T, Hoffmann M, Deßloch S. Marimba: a framework for making mapreduce jobs incremental. In: Proceedings of the 2014 IEEE International Congress on big data. New York: IEEE; 2014. p. 128–35.
31. Zaharia M, Borthaku D, Sarma JS, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. Proceedings of the 5th European Conference on computer systems. New York: ACM; 2010. p. 265–78.
32. Tang Z, Jiang L, Zhou J, Li K, Li K. A self-adaptive scheduling algorithm for reduce start time. Future Gener Comput Syst. 2015;43(C):51–60.
33. Langville AN, Meyer CD. Deeper inside pagerank. Internet Math. 2004;1(3):335–80.
34. Sneyers J, Schrijvers T, Demoen B. Dijkstra's algorithm with fibonacci heaps: an executable description in chr. In: Proceedings of the 20th Workshop on logic programming; 2006. p. 182–91.
35. Kang U, Tsourakakis CE, Faloutsos C. Pegasus: a peta-scale graph mining system implementation and observations. Proceedings of the 2009 Ninth IEEE International Conference on data mining. Washington: IEEE; 2009. p. 229–38.
36. Jain AK, Murty MN, Flynn PJ. Data clustering: a review. ACM Comput Surv. 1999;31(3):264–323.
37. Xu R, Wunsch D. Survey of clustering algorithms. IEEE Trans Neural Netw. 2005;16(3):645–78.
38. ClueWeb09 Dataset. http://www.lemurproject.org/clueweb09/webGraph.php.
39. BigCross Dataset. http://www-old.cs.uni-paderborn.de/en/research-group/ag-bloemer/research/abgeschlossene/clustering/streamkmpp.html.