Future Generation Computer Systems 51 (2015) 98–110

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

CloudFlow: A data-aware programming model for cloud workflow applications on modern HPC systems

Fan Zhang[a,*], Qutaibah M. Malluhi[b], Tamer Elsayed[b], Samee U. Khan[c], Keqin Li[d], Albert Y. Zomaya[e]

[a] *Kavli Institute for Astrophysics and Space Research, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*
[b] *KINDI Center for Computing Research, Qatar University, Doha, Qatar*
[c] *Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58108-6050, USA*
[d] *Department of Computer Science, State University of New York, New Paltz, NY 12561, USA*
[e] *School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia*

HIGHLIGHTS

- CloudFlow programming model is designed for cloud workflow on modern HPC systems.
- CloudFlow is not only data-aware, but also shared-data-aware.
- The programming model supports multiple Map and Reduce functions.
- Theoretical analysis proves the correctness and uniqueness of each desired output.
- Results show the speedup of CloudFlow exceeds 4X compared to traditional MapReduce.

ARTICLE INFO

Article history:
Received 14 July 2014
Received in revised form
30 September 2014
Accepted 25 October 2014
Available online 6 November 2014

Keywords:
Concurrency
Data aware
MapReduce
HPC
Programming model

ABSTRACT

Traditional High-Performance Computing (HPC) based big-data applications are usually constrained by having to move large amount of data to compute facilities for real-time processing purpose. Modern HPC systems, represented by High-Throughput Computing (HTC) and Many-Task Computing (MTC) platforms, on the other hand, intend to achieve the long-held dream of moving compute to data instead. This kind of data-aware scheduling, typically represented by Hadoop MapReduce, has been successfully implemented in its Map Phase, whereby each Map Task is sent out to the compute node where the corresponding input data chunk is located. However, Hadoop MapReduce limits itself to a one-map-to-one-reduce framework, leading to difficulties for handling complex logics, such as pipelines or workflows. Meanwhile, it lacks built-in support and optimization when the input datasets are shared among multiple applications and/or jobs. The performance can be improved significantly when the knowledge of the shared and frequently accessed data is taken into scheduling decisions.

To enhance the capability of managing workflow in modern HPC system, this paper presents CloudFlow, a Hadoop MapReduce based programming model for cloud workflow applications. CloudFlow is built on top of MapReduce, which is proposed not only being data aware, but also shared-data aware. It identifies the most frequently shared data, from both task-level and job-level, replicates them to each compute node for data locality purposes. It also supports user-defined multiple Map- and Reduce functions, allowing users to orchestrate the required data-flow logic. Mathematically, we prove the correctness of the whole scheduling framework by performing theoretical analysis. Further more, experimental evaluation also shows that the execution runtime speedup exceeds 4X compared to traditional MapReduce implementation with a manageable time overhead.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

1.1. *Background introduction*

Traditional HPC systems [1], characterized by high-frequency processor design, efficient caching system, large input/output

* Corresponding author.
E-mail addresses: f_zhang@mit.edu (F. Zhang), qmalluhi@qu.edu.qa (Q.M. Malluhi), telsayed@qu.edu.qa (T. Elsayed), samee.khan@ndsu.edu (S.U. Khan), lik@newpaltz.edu (K. Li), albert.zomaya@sydney.edu.au (A.Y. Zomaya).

http://dx.doi.org/10.1016/j.future.2014.10.028
0167-739X/© 2014 Elsevier B.V. All rights reserved.

capacity and efficient cooling techniques [2], have been exclusively referred to as Supercomputing [3] or Grid Computing [4]. However, upgrading hardware to achieve high performance is not only expensive, but also requires high-performance applications to leverage the hardware capacity. Modern HPC systems, represented by High-Through Computing (HTC) [1] and Many-Task Computing (MTC) [5], suggest an otherwise direction. Instead of scaling up compute resources to meet accelerated computing demand, scaling out compute resources has gained more attention. Scaling out is typically represented by incorporating more commodity compute nodes to achieve scalability purpose.

To fit in the needs of scaling compute resource out, different programming models have been proposed to maximize the scaling capacity. Among these, one of the most successful programming models is MapReduce (MR) [6]. It has been widely accepted for large-scale and data-intensive applications [7,8]. Hadoop [9], an open source implementation of MR, gains popularity due to its reliability and scalability in providing a parallel yet simple framework [10]. Hadoop MapReduce splits input data into chunks, allocating each task to a compute node where its input data resides. In this way, it achieves data locality naturally, and the task number can be configured to utilize all compute nodes.

However, splitting input dataset to realize parallelization is mostly simple and straightforward. Due to the ever-increasing complexity of execution logic in real-life applications, more and more MR applications involve multiple correlated jobs. They are encapsulated and executed in a predefined order. For example, a PageRank job [11] involves two iterative MR sub-jobs; the first job joins a rank table and a linkage table, and the second job calculates the aggregated rank of each URL. Two non-iterative MR sub-jobs for counting out-going URLs and assigning initial ranks are also included in the PageRank job.

Though the formulation of each MR sub-job is easy, the overall solution may involve many redundant MR sub-jobs. The problem becomes even worse when the MR sub-jobs involve multiple straightforward but time-consuming tasks. A concrete example demonstrating such redundant MR sub-jobs is given in Section 1.1.

As a rule-of-thumb, MR has proven to be effective in processing large amount of dividable unshared input datasets. Different and concurrently running MR jobs may share input datasets. Linkage table in the PageRank job is an example of the shared input dataset. Performance of MR applications can be improved if the sharing of input datasets is considered. Therefore, we identify two types of data sharing: data sharing among multiple sub-jobs of the same job, and data sharing among multiple distinct jobs.

Traditionally, one MR job involves multiple Map and Reduce Tasks. Each Map Task executes the same Map Function and processes its own input data splits based on the instructions defined in the Map Function. Each Reduce Task executes the same Reduce Function and processes its own key–value pair partitions. This one-map-to-one-reduce framework, though simple enough, loses its flexibility when applications require complex logic. For such complex scenarios, users employ multiple jobs, such as the implementation of PageRank and Descendant Query in [11].

In this paper, we propose Concurrent MapReduce for cloud worklow, *CloudFlow*; a novel programming model that supports multiple and heterogeneous Map and Reduce Functions. Optimizations over shared data are offered at the function level and at the job level.

1.2. Motivation example

First, we discuss a case study of increasing salary for all employees in a multi-national research company for the coming year. Fig. 1(a) depicts a scenario that the new salary for each employee is based on either his or her current title or evaluation score. The left table (Employee Information or EI) has the information of each employee including ID, name, title, current salary, and score. The upper-right table (Title Rate or TR) depicts the titles and their corresponding salary increase rates. The lower-right table (Evaluation Rate or ER) has the evaluation scores and their corresponding rates. The company wants to compare the two methods and find out the method that costs less.

In Fig. 1(b), the salary is raised based on which branch the employee is working at. The two tables on the left (EI and NC) should be joined. There are two different rate strategies as shown in the two tables on the right (CR1 and CR2).

In traditional MR, for case 1 (in Fig. 1(a)), a total of six MR jobs are launched. The first job processes EI and outputs <ID, Title, Current Salary> for each employee. The second processes TR and outputs <Title, Rate>. The third job joins the outputs and calculates the salary for each employee. Similarly, the other three follow-up MR jobs are launched to process the EI and ER and join them. Based on this, the following points can be observed:

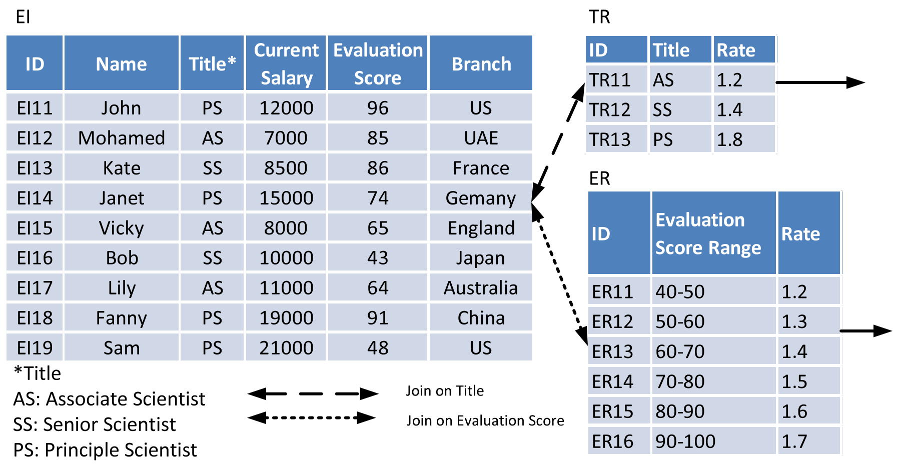
(1) The Map stage on both TR and ER does nothing but pass the data to the Reduce Tasks to join. This observation is not new, e.g., in Hadoop Sort benchmark [1]. However, the launching of the Map tasks, though it does nothing, causes extra overhead by allocating Map slots and creating JVMs [12].
(2) There is no reason to sequentially execute the two MR jobs illustrated by dashed-line arrows in Fig. 1(a). Once the output of EI is generated, the two MR jobs can be executed concurrently. This is useful if there are more than one table that needs to be joined, such as TR and ER in Fig. 1(a). This is also useful if multiple versions of one table are to be joined (such as CR1 and CR2 in Fig. 1(b)). We hereby refer to the two concurrent joining functions as **Reduce Functions**.
(3) In Fig. 1(b), tables EI and BC should be joined before being shuffled to the two Reduce Functions. However, traditional MR supports only one-type of map, which cannot be performed on two different tables or datasets. In this paper, we support concurrent Map Functions performed on different input datasets, such as EI, NC, CR1 and CR2. We hereby refer to them as **Map Functions**.
(4) The shared EI table is read twice by the two Map Functions, but in reality, it would be more efficient to read it only once and then supply it to the two Map Functions. This significantly reduces the I/O overhead.

In this paper, the CloudFlow programming model reduces job execution time by launching concurrent and heterogeneous Map and Reduce Functions. Each Map and Reduce Function, as in traditional MR, involves a set of homogeneous Map and Reduce Tasks.

Different from the above examples where data sharing takes place among function level, data can be shared also among multiple jobs that are submitted by different users. For example, two different managers who are unaware of each other's job submit their own jobs. The first manager submits the job of case 1 in Fig. 1(a) and the second manager submits the job of case 2 in Fig. 1(b). The input dataset EI is shared by the two different jobs.

Other typical examples of job-level concurrency include scientific simulations [13], which normally need multiple runs on a set of same input data but with different application configurations or parameters. All these jobs can run concurrently.

Traditional MR offers job switching optimization mechanisms, such as the use of fair or capacity scheduler, but no such techniques exist for managing shared data among different jobs. In this paper, we propose an optimization mechanism based on the access frequency of different datasets.



EI

ID	Name	Title*	Current Salary	Evaluation Score	Branch
EI11	John	PS	12000	96	US
EI12	Mohamed	AS	7000	85	UAE
EI13	Kate	SS	8500	86	France
EI14	Janet	PS	15000	74	Gemany
EI15	Vicky	AS	8000	65	England
EI16	Bob	SS	10000	43	Japan
EI17	Lily	AS	11000	64	Australia
EI18	Fanny	PS	19000	91	China
EI19	Sam	PS	21000	48	US

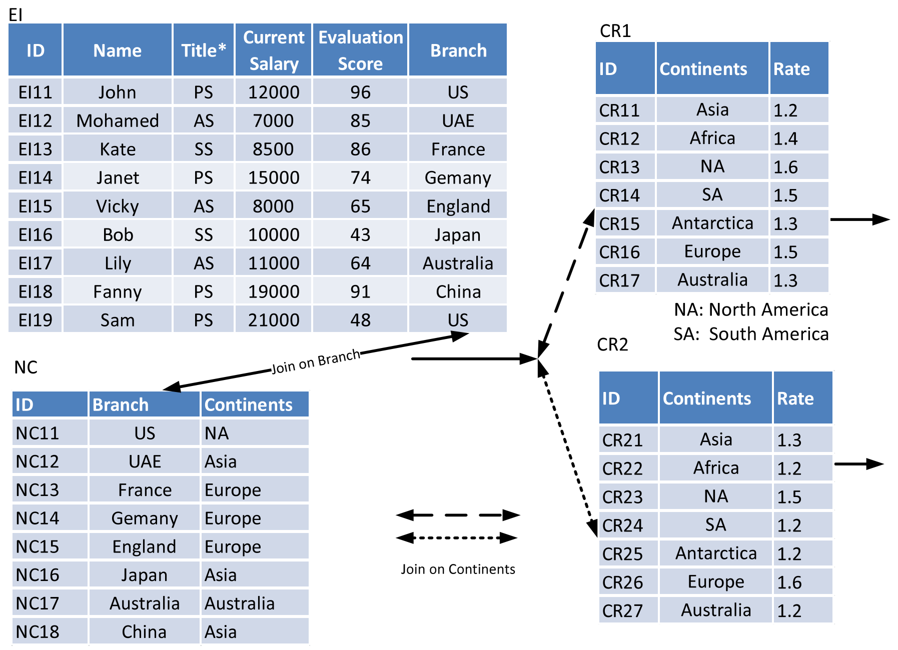
TR

ID	Title	Rate
TR11	AS	1.2
TR12	SS	1.4
TR13	PS	1.8

ER

ID	Evaluation Score Range	Rate
ER11	40-50	1.2
ER12	50-60	1.3
ER13	60-70	1.4
ER14	70-80	1.5
ER15	80-90	1.6
ER16	90-100	1.7

(a) Case 1: Two Map Functions for EI and TR. Two other Map Functions for EI and ER. Two reduce functions for the two join operations.



EI

ID	Name	Title*	Current Salary	Evaluation Score	Branch
EI11	John	PS	12000	96	US
EI12	Mohamed	AS	7000	85	UAE
EI13	Kate	SS	8500	86	France
EI14	Janet	PS	15000	74	Gemany
EI15	Vicky	AS	8000	65	England
EI16	Bob	SS	10000	43	Japan
EI17	Lily	AS	11000	64	Australia
EI18	Fanny	PS	19000	91	China
EI19	Sam	PS	21000	48	US

CR1

ID	Continents	Rate
CR11	Asia	1.2
CR12	Africa	1.4
CR13	NA	1.6
CR14	SA	1.5
CR15	Antarctica	1.3
CR16	Europe	1.5
CR17	Australia	1.3

CR2

ID	Continents	Rate
CR21	Asia	1.3
CR22	Africa	1.2
CR23	NA	1.5
CR24	SA	1.2
CR25	Antarctica	1.2
CR26	Europe	1.6
CR27	Australia	1.2

NC

ID	Branch	Continents
NC11	US	NA
NC12	UAE	Asia
NC13	France	Europe
NC14	Gemany	Europe
NC15	England	Europe
NC16	Japan	Asia
NC17	Australia	Australia
NC18	China	Asia

(b) Case 2: Six Map Functions and two reduce functions.

Fig. 1. A concrete example to explain both function-level and job-level concurrency.

1.3. Contribution and organization of this paper

The above observation motivates the design of CloudFlow as a new programming model to benefit applications that have shared data on both function and job levels, while retaining the simplicity of traditional MR. CloudFlow has the following advantages:

(1) It defines a framework that supports multiple different Map and Reduce Functions running concurrently, where each function has either shared or unshared data. Each Map or Reduce Function corresponds to the Map or Reduce stage of traditional MR, which has multiple Map or Reduce Tasks running on multiple nodes. This is further explained in Section 2.
(2) It supports optimized data sharing by utilizing a *Shared Function Data Handler*, which supplies the shared data to different Map Functions that need them. For the same Map Functions that share the data, it further provides one copy of output to reduce the shuffle overhead. This is further explained in Section 4.2.
(3) It supports optimized data sharing by utilizing a *Shared Job Data Handler*, which identifies multiple jobs of different users, finds out the frequently- or partially-shared data items, and copies them to the local file system for future job use. This is further explained in Section 4.1.
(4) We conduct theoretical analysis of implementing the CloudFlow architecture, proving that the correctness and uniqueness of each key–value output that can be generated as desired in Section 3.

The rest of the paper is organized as follows: Section 2 introduces the architecture of our CloudFlow programming model. We conduct a theoretical analysis of the methodology in Section 3. Optimization techniques that are applied at the function- and job-levels to enhance this programming model are described in Section 4. In Section 5, we introduce the experimental setting and comparative results on two real applications. Finally, we conclude with a review of related work and summarize our technical contributions.

2. CloudFlow overview

In this section, we analyze the example in Section 1 to demonstrate the essence of sharing data in our programming model,

Table 1
Symbols, definitions and illustrations.

Symbol	Definitions and illustrations
EI	A table demonstrates all the employees information
TR	A table demonstrates all employee's title and the increased salary rate based on the title
ER	A table demonstrates all employees' year-end evaluation scores and the increased salary rate based on each category of the evaluation score
CR	A table demonstrates all employees' working branch and the increased salary rate based on the continent each branch locates
NC	A table demonstrates all employees' working branch city and the continent each city belongs to
MR	Short for MapReduce
MF/MT	Map Function and Map Task
RF/RT	Reduce Function and Reduce Task
m/M	The mth Map Function/The number of Map Functions in a CloudFlow job
r/R	The rth Map Function/The number of Reduce Functions in a CloudFlow job
$G : E_G(MF_m, RF_r) = 1$	Output of the mth Map Function connect to the rth Reduce Function
$\text{Rel}(MF_m, RF_r)$	A list of Map Functions that all these Functions other than MF_m that connect to RF_r
$\text{Rel}(MF_m)$	All Reduce Functions that Map Function MF_m connects to
$\text{iRel}(MF_m)$	All Reduce Functions that Map Function MF_m does NOT connect to
f	A function that maps a key–value pair to a set of M-tuples
CS_i	A Reduce Function cell set that key/value pair (k_t, v_t) is shuffled to

and then we introduce the architecture of CloudFlow. To help the reader, we use Table 1 to cover all the symbols and their definitions in order to refresh readers' memory.

2.1. Illustration of handling shared data for CloudFlow

Different Map and Reduce Functions may share data at the function level. The shared data can be the input for different Map Functions, or intermediate key–value pairs for different Reduce Functions. At the job level, the shared data are the inputs for different jobs. The shared data can be the whole dataset, or part of it. Before formally introducing the CloudFlow architecture, we propose two scenarios using the cases introduced in Section 1.

Scenario 1: In Section 1.2, case 1 and case 2 are two distinct jobs that might be submitted by two different programmers, who are unaware of each other's job. In this scenario, the two cases are two concurrent jobs illustrated in Fig. 2(a) and (b).

The four Map Functions for case 1 are denoted as MF01, MF02, MF03 and MF04, which process TR, EI, EI and ER respectively and output key–value pairs. The first Reduce Function RF01 collects the key–value pairs from MF01 and MF02 to output the new salaries based on title. Similarly, the second Reduce Function RF02 collects the key–value pairs from MF03 and MF04 to output the salaries based on evaluation scores.

Very similar to the first job, the second job has six Map Functions, MF07, MF08, …, MF12 for CR1, EI, NC, EI, NC and CR2 respectively. As shown in Fig. 2(b), the first Reduce Function RF03 joins data from MF07, MF08 and MF09 to output the new salaries based on the CR1 rates. Similar map and Reduce Functions are shown for the CR2 rate.

Scenario 2: In this scenario, all the Map and Reduce Functions are treated as one job. These functions can be written by one single programmer, who is aware of all the tables and datasets and writes the source code of this job.

Thus, there are 12 Map Functions, MF01, MF02, …, MF12, each one representing one input entry of Fig. 2. For the join part at the right side of Fig. 2, there are four Reduce Functions named as RF01, RF02, RF03 and RF04.

There is one major difference between the two scenarios. In scenario 1, the programmers of the two jobs are unaware that table EI is being shared. They just submit their jobs and wait for the results. It is CloudFlow's responsibility to discover the shared data and provide mechanisms to optimize the concurrent access. In scenario 2, the programmer has prior knowledge about all the tables and datasets that are used and shared. Thus, he/she can explicitly indicate the data sharing and the CloudFlow uses this information to provide optimization mechanisms for efficient data access.

2.2. Terms of the CloudFlow programming model

To further clarify the concepts, we formally define all the terms used in this paper.

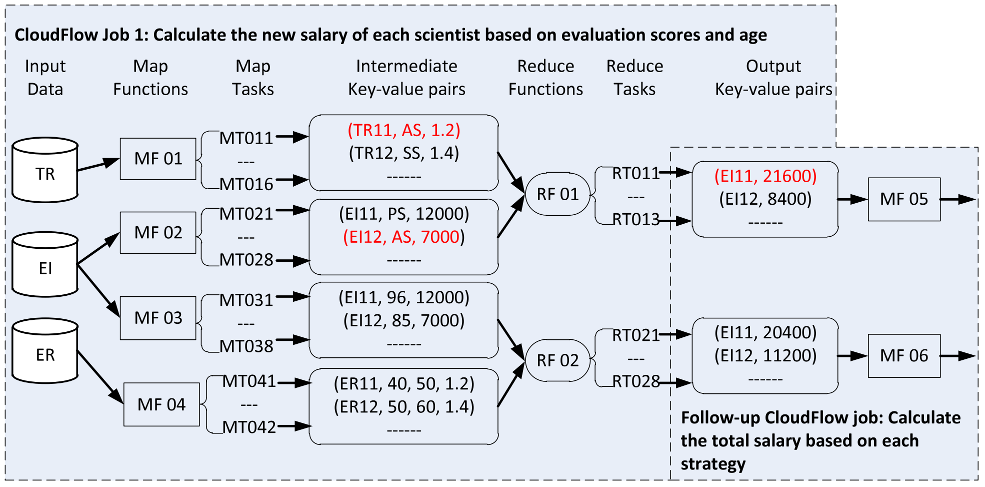
(1) **Input Data**: The datasets associated with the whole job. MapReduce can process various types of data formats, from plain text, XML to database files. Input data is divided into many data splits for Map tasks to process. Different from traditional Hadoop MapReduce, input data in CloudFlow serves multiple heterogeneous Map Functions, with each Map Function launching a few homogeneous Map Tasks. Therefore, the input data, even though located in HDFS in general, is logically partitioned to server different Map Functions. Details of the Map Functions are introduced below.

(2) **Map Function**: One Map Function (MF), or a Mapper, is composed of a set of homogeneous Map Tasks (MT) that work on different input data splits. It defines execution logics in its body function. Each Map Function is associated with a set of input data, or has no input data at all, such as Pi estimation. One Map Task is an instance and implementation of its Map Function. It resides in Java Virtual Machine (JVM) launched by tasktracker.

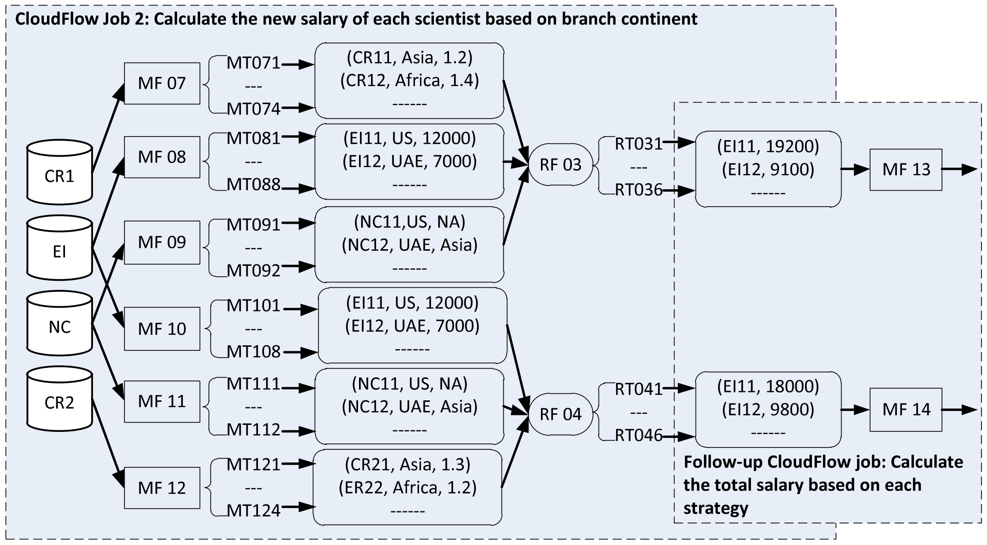
Even though the definition of Map Function and Map Task are exactly the same in CloudFlow as their definitions used in traditional MapReduce, the only difference is that CloudFlow supports a variety of heterogeneous Map Functions. These Map Functions can be different from each other in the following three aspects: (a) different input datasets (e.g. MF01 and MF02), or (b) same input dataset but different Map Functions (e.g. MF02 and MF03), or (c) same input dataset and same Map Function (e.g. MF08 and MF10). It is necessary to keep (c) here since the reduce stage requires the intermediate key–value pairs to be shuffled to different Reduce Functions. We will show optimization techniques along this line later.

(3) **Reduce Function**: One Reduce Function (RF), or a Reducer, is a set of Reduce Tasks that process on their own partition of data. One Reduce Task (RT) is also an instance and implementation of its Reduce Function. Similarly, tasktracker launches JVM to hold Reduce Tasks.

In tradition MapReduce, a Reduce Task receives partitions of intermediate key–value pairs from different Map Tasks of one Map Function. In CloudFlow, however, one Reduce Function looks more like a joiner, which merges the intermediate key–values pairs generated by different Map Functions. For example, RF01 calculates the new salaries based on the title of each employee and RF02 does the same thing based on the evaluation scores. Each Reduce Task performs the actual join task and output the data to HDFS.


CloudFlow Job 1: Calculate the new salary of each scientist based on evaluation scores and age
Input Data
Map Functions
Map Tasks
Intermediate Key-value pairs
Reduce Functions
Reduce Tasks
Output Key-value pairs
TR
EI
ER
MF 01
MF 02
MF 03
MF 04
MT011 --- MT016
MT021 --- MT028
MT031 --- MT038
MT041 --- MT042
(TR11, AS, 1.2) (TR12, SS, 1.4) ------
(EI11, PS, 12000) (EI12, AS, 7000) ------
(EI11, 96, 12000) (EI12, 85, 7000) ------
(ER11, 40, 50, 1.2) (ER12, 50, 60, 1.4) ------
RF 01
RF 02
RT011 --- RT013
RT021 --- RT028
(EI11, 21600) (EI12, 8400) ------
(EI11, 20400) (EI12, 11200) ------
MF 05
MF 06
Follow-up CloudFlow job: Calculate the total salary based on each strategy

(a) Illustration of scenario 1, in which four Map Functions and 2 Reduce Functions are used for calculating new salaries based on evaluation scores and titles.


CloudFlow Job 2: Calculate the new salary of each scientist based on branch continent
CR1
EI
NC
CR2
MF 07
MF 08
MF 09
MF 10
MF 11
MF 12
MT071 --- MT074
MT081 --- MT088
MT091 --- MT092
MT101 --- MT108
MT111 --- MT112
MT121 --- MT124
(CR11, Asia, 1.2) (CR12, Africa, 1.4) ------
(EI11, US, 12000) (EI12, UAE, 7000) ------
(NC11,US, NA) (NC12, UAE, Asia) ------
(EI11, US, 12000) (EI12, UAE, 7000) ------
(NC11, US, NA) (NC12, UAE, Asia) ------
(CR21, Asia, 1.3) (ER22, Africa, 1.2) ------
RF 03
RF 04
RT031 --- RT036
RT041 --- RT046
(EI11, 19200) (EI12, 9100) ------
(EI11, 18000) (EI12, 9800) ------
MF 13
MF 14
Follow-up CloudFlow job: Calculate the total salary based on each strategy

(b) Illustration of scenario 2, in which six Map Functions and two Reduce Functions are used for calculating new salaries based on two versions of branch continent strategy.

Fig. 2. Illustration of the key–value pairs and data flow of the Map Functions and Reduce Functions in the two CloudFlow jobs. All the Map Tasks and Reduce Tasks launched by their own Map and Reduce Functions. A follow-up job, which calculates the total salary, is also partially shown.

(4) **Job**: A CloudFlow job is composed of a Map phase and its subsequent Reduce phase. As shown in Fig. 2, there are two CloudFlow jobs. Map phase consists of multiple Map Functions running concurrently. Multiple Reduce Functions are also applied to the Reduce phase. In traditional MapReduce, a single job consists of one Map Function and one Reduce Function only.

(5) **Workflow**: A workflow is a pipeline of CloudFlow jobs that are executed to serve a certain purpose. For example, there are two consecutive CloudFlow jobs in Fig. 2(a), which is a workflow. Traditional MapReduce can also be chained into a list of jobs by instantiating the JobControl object. Dependencies of jobs are specified as a Directed Acyclic Graph (DAG). The only difference, as we noted previously, is the component of each single job makes the difference.

2.3. CloudFlow architecture

Fig. 3 illustrates the architecture of CloudFlow framework. From top down, we can see application, CloudFlow framework and file system tiers. A vertical line in the middle separates job related illustrations (at the left side) from function and task related illustrations (at the right side).

In the CloudFlow framework, the Job Analyzer module decomposes each job into a set of Map and Reduce Functions. Then it identifies the input data from the configuration file. For example, job 1 in Fig. 3 is decomposed into a set of Map Functions (MF11, MF12, ...) and Reduce Functions (RF11, RF12, ...). The Job Scheduler manages the submission, monitoring, and runtime states of all the submitted jobs. The Shared Job Data Handler manages the shared data at the job level. In the previous example, it discovers the shared table EI and copies the data to the local file system. This mechanism is further discussed in Section 4.1.

The Shared Function Data Handler, on the other hand, serves and optimizes shared data for Map and Reduce Functions. For heterogeneous Map Functions that share the same input data, this handler creates a daemon on each host the data locates, reads the data and supplies them to all the Map Tasks associated with these Map Functions. For homogeneous Map Functions that share the same input data, this handler outputs only one copy. We will discuss the detail mechanism in Section 4.2.

Each job tracker, similar to traditional MR, sends Map and Reduce Tasks to all the nodes. The difference is that it has to

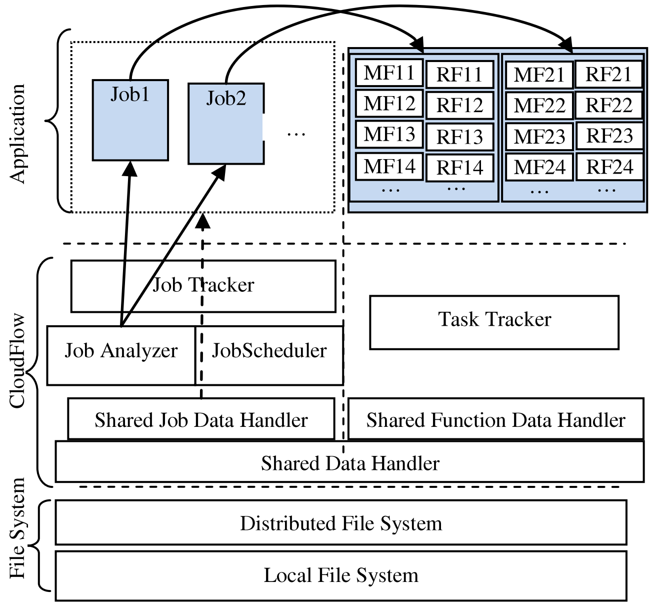


Fig. 3. Architecture of the CloudFlow framework.

manage the tasks of the entire Map and Reduce Functions. The job tracker has to communicate with the name node to locate the input data and the task trackers. Each task tracker is also configured with a bunch of slots for Map and Reduce Tasks. An individual job scheduler manages the execution order of the tasks. For example, EI should be firstly joined with NC and then with CR1 in job 2 of scenario 1.

The distributed file system is built on top of the local file system. Unlike traditional MR where the input data are read from HDFS, CloudFlow copies the shared and most frequently used data to local file system to expedite data access.

In the next section, we formally introduce the CloudFlow programming model.

2.4. CloudFlow programming model

The CloudFlow enables the definition of multiple heterogeneous Map and Reduce Functions. We formulate the representation as below:

$$\text{Map}: (k_m, v_m) \rightarrow \{(k'_m, v'_m)\}, \quad m \in [1, M]$$

Reduce:

$$\{\{(k'_{(i,r)}, [v'_{(i,r)}])\}, \dots, \{(k'_{(j,r)}, [v'_{(j,r)}])\}\} \rightarrow [v''_r]$$
$$(i, r), \dots, (j, r) \in [1, M], \quad r \in [1, R].$$

This representation includes M Map Functions and R Reduce Functions. The Reduce Functions are different from traditional MR programming model in that they take input key/value pairs from multiple key–value pair outputs of Map Functions. (i, r) or (j, r) denotes a mapping from a tuple to a number between 1 and M. Symbol r represents the index of the Reduce Function. Symbol pairs (i, r) and (j, r) represent the first and last Map Function that feed the rth Reduce Function.

For example, Map Functions on input key/value pairs $(k_{(i,1)}, v_{(i,1)}), \dots, (k_{(j,1)}, v_{(j,1)})$ produce an output of the form $\{(k'_{(i,1)}, v'_{(i,1)})\}, \dots, \{(k'_{(j,1)}, v'_{(j,1)})\}$. All these outputs are joined as the first Reduce Function and the corresponding output dataset is $[v''_1]$.

More specifically, we use job 1 in scenario 1 as an example, (k_1, v_1) and (k_3, v_3) represent the input dataset EI. (k_2, v_2) and (k_4, v_4) represent the input dataset TR and ER respectively. All the intermediate key–value pairs $(k'_1, [v'_1])$, $(k'_2, [v'_2])$, $(k'_3, [v'_3])$ and $(k'_4, [v'_4])$ output by the four Map Functions are shown in the Fig. 2(a).

In the reduce stage, $(k'_1, [v'_1])$, $(k'_2, [v'_2])$ are joined and the results are outputted as $[v''_1]$. These are also similarly applied to $(k'_3, [v'_3])$ and $(k'_4, [v'_4])$, where the reduce output is represented $[v''_3]$. All the real key–value pairs of the reduce outputs are also shown in Fig. 2(a).

Similar as traditional MR, a configuration file of all the functions and modules are as follows. A CloudFlow programmer defines this file in the job configuration file. The CloudFlow reads this file and initialize the job with the configuration specifications accordingly.

```
Class CloudFlowJob{
   Class Map_m(key, value); m ∈ [1, M]
   Class Reduce_r(key, value); r ∈ [1, R]
   MapFunctionList[] addMap(Map_m);
   ReduceFunctionList[] addReduce(Reduce_r);
   Boolean[][] maptoReduce(mapFunctionList,
   reduceFunctionList);
   defineJoinRules(reduceFunctionList);
   addInputDataDir(Map_m);
   setOutputDir(Dir);
   sharedFunctionDataHandler[]
   sharedFunctionDataHandler();
}
```

Map_m: the mth user-defined Map Function class. It implements a Map Function, which accepts a key–value pair and transforms it to a set of new key–value pairs. In CloudFlow, a set of M Map Functions is defined as a Map Function list.

Reduce_r: the r^h user-defined Reduce Function class. It implements a Reduce Function, which accepts intermediate key–value pair partitions generated by one or a set of Map Functions and outputs merged results. Similarly, a set of R Reduce Functions is defined as a Reduce Function List.

***addMap* (MapClass)**: This function adds a set of Map Functions to a Map Function list, and returns the list entry. Each Map Function is defined in *Map_m*.

***addReduce* (ReduceClass)**: This function adds a set of Reduce Functions to a Reduce Function list, and returns the list. Each Reduce Function is defined in *Reduce_r*.

***maptoReduce*()**: This function defines relationship between Map and Reduce Functions. In CloudFlow, multiple Map Functions may be followed by multiple Reduce Functions, and this function manages the multiple-map to multiple-reduce mapping relationship. It returns a M-row, R-column Boolean matrix whose (i, j) is true if output of *Map_i* feeds to the input of *Reduce_j*.

***defineJoinRules* (reduceFunctionList)**: There are many different ways to perform Reduce Functions. It can be any theta join ($<, \leq, =, >, \geq$) type from multiple Map Functions. *DefineJoinRules* is set by the programmer to define how the Map Functions should be joined.

***addInputDataDir* (mapFunctionList)**: This function associates each Map Function with its input dataset. One Map Function can also be associated to a set of input data located in different places.

***sharedTaskDataHandler* (SharedMapClassList)**: It gets a list of Map Functions that have the same input data and returns a handler to manage the input data for all those Map Functions. Detailed introduction of this function is shown in the following section.

3. Theoretical analysis

Even though we have identified the framework and architecture of the CloudFlow programming model, the correctness, meaning that if it delivers desirable and possible output is still yet to be proved. In this section, we perform a theoretical analysis to prove that: Given the current CloudFlow design, each required and expected output key/value pair is generated one and only one time. From Definition and Term (1) to (10) below, we list the major concepts that we utilize, and we follow it up by proving the correctness and uniqueness of output key/value pair in Lemmas 1, 2 and the formal deduction.

(1) Let $\{\boldsymbol{MF}\} = \{MF_0, \ldots, MF_{M-1}\}$ be the set of M Map Functions.

(2) Let $\{\boldsymbol{RF}\} = \{RF_0, \ldots, RF_{R-1}\}$ be the set of R Reduce Functions.

(3) Let G be the bipartite graph that represents the dependency (or links) between (the output of) the Map Functions and (the input of) the Reduce Functions.

$$\mathbf{G} : \boldsymbol{E_G(MF_m, RF_r)} = \mathbf{1} \text{ if the output of } MF_m \text{ is an input to } RF_r.$$

(4) Relevant Map Functions for Map Function MF_m associated with Reduce Function RF_r,

$$\mathbf{Rel}(\boldsymbol{MF_m}, \boldsymbol{RF_r}) : \{MF_i\},$$
$$\forall MF_i \in \mathrm{Rel}(MF_m, RF_r), E_G(MF_m, RF_r) = 1 \wedge E_G(MF_i, RF_r) = 1$$
$$\forall MF_i \notin \{MF\} - \{MF_i\}, E_G(MF_i, RF_r) = 0.$$

(5) All relevant Map Functions for Map Function MF_m:

$$\mathbf{Rel}(\boldsymbol{MF_m}) : \{MF_i\}, \quad \exists RF_r \in S_R, E_G(MF_m, RF_r) = 1 \wedge E_G(MF_i, MF_m) = 1.$$

Typically, $\mathrm{Rel}(MF_m) = \mathrm{Rel}(MF_m, RF_0) \cup \mathrm{Rel}(MF_m, RF_1) \cup \cdots \cup \mathrm{Rel}(MF_m, RF_R)$.

(6) Irrelevant Map Functions for Map Function MF_m associated with Reduce Function RF_r,

$$\mathbf{iRel}(\boldsymbol{MF_m}, \boldsymbol{RF_r}) : \{MF_i\}, \{MF\} - \mathrm{Rel}(MF_m, RF_r) - MF_m.$$

(7) Irrelevant Map Functions for Map Function m:

$$\mathbf{iRel}(\boldsymbol{MF_m}) : \{MF_i\}, \{MF\} - \mathrm{Rel}(MF_m) - MF_m.$$

Typically, $\mathrm{iRel}(MF_m) = \mathrm{iRel}(MF_m, RF_0) \cap \mathrm{iRel}(MF_m, RF_1) \cap \cdots \cap \mathrm{iRel}(MF_m, RF_R)$.

(8) Let $(\boldsymbol{k_m}, \boldsymbol{v_m})$ be a key and value pair of the output of map function MF_m.

(9) The shuffling space is $\boldsymbol{M}$-dimension discrete space. The length of dimension $\boldsymbol{j}$ is $\boldsymbol{u_j}$.

(10) $\boldsymbol{f}$ is the function that maps a key–value pair to a set of M-tuples.
$\boldsymbol{f} : (\boldsymbol{k_m}, \boldsymbol{v_m}) \rightarrow \{(\boldsymbol{i_1}, \ldots, \boldsymbol{i_M})\}$, $\boldsymbol{i_j} \in \{\mathbf{0}, \mathbf{1}, \ldots, \boldsymbol{u_j} - \mathbf{1}\}$, u_j is the number of Reduce units in dimension j of the shuffling space. $\{(i_1, \ldots, i_M)\}$ for (k_m, v_m) is named as Cell Set CS_m.

The set of M-tuples is determined as follows:

$$i_m = h(k_m)\%u_m \tag{1}$$
$$i_j = \text{all of } \{0, \ldots, u_j - 1\} \quad \text{if } MF_j \in \mathrm{Rel}(MF_m, RF_r) \tag{2}$$
$$i_j = g(j) \quad \text{if } MF_j \in \mathrm{iRel}(MF_m, RF_r), \tag{3}$$

g is another hash function that takes a dimension index j returns a value in $\{0, \ldots, u_j - 1\}$.

We need to prove that: Suppose $MF_m \in \{MF_m\}, \forall RF_r \in \{RF_r\}$

$$\mathrm{Rel}(MF_m, RF_r) = \{MF_0, MF_1, \ldots, MF_{m'}\}, \tag{4}$$

then $|f(k_m, v_m) \cap f(k_0, v_0) \cap f(k_1, v_1) \cap \cdots \cap f(k_{m'}, v_{m'})| = 1$.

Lemma 1. *If* $\mathrm{Rel}(MF_m, RF_r) = \{MF_0, MF_1, \ldots, MF_{m'}\}$, *then:*

(1) *for each* $j \notin \{m, 0, 1, 2, \ldots, m'\}$, $MF_j \in \mathrm{iRel}(MF_m, RF_r)$;
(2) *If* $MF_j \in \mathrm{iRel}(MF_m, RF_r)$, $j \notin \{m, 0, 1, 2, \ldots, m'\}$.

The proof of this lemma is straightforward, which is not necessarily given here.

Lemma 2. *If* $\mathrm{Rel}(MF_m, RF_r) = \{MF_0, MF_1, \ldots, MF_{m'}\}$, *then:*

$$\mathrm{Rel}(MF_0, RF_r) = \{MF_m, MF_1, \ldots, MF_{m'}\},$$
$$\mathrm{Rel}(MF_1, RF_r) = \{MF_0, MF_m, \ldots, MF_{m'}\}, \ldots,$$
$$\mathrm{Rel}(MF_{m'}, RF_r) = \{MF_0, MF_1, \ldots, MF_m\}.$$

Problem we need to prove:

We first prove there is at least one M-tuple $(i_0, \ldots, i_{M-1})$ which satisfies

$$|f(k_m, v_m) \cap f(k_0, v_0) \cap f(k_1, v_1) \cap \cdots \cap f(k_{m'}, v_{m'})| = 1, \tag{5}$$

and then prove the uniqueness of the M-tuple.

Proof. (1) For (k_t, v_t), by definition of the f function, it is shuffled to a Cell Set CS_t, which is denoted by:

$$S_t = h(k_t)\%u_t \tag{6}$$
$$\{S_j\} = \text{all of } \{0, \ldots, u_j - 1\} \quad \text{if } MF_j \in \mathrm{Rel}(MF_t) \tag{7}$$
$$S_j = g(j) \quad \text{if } MF_j \in \mathrm{iRel}(MF_t)$$
$$t \in \{m, 0, 1, 2, \ldots, m'\}. \tag{8}$$

Then there is one M-tuple $(i_0, \ldots, i_{M-1}) \in CS_m \cap CS_0 \cap CS_1 \cap \cdots \cap CS_{m'}$, which is:

$$i_m = h(k_m)\%u_m \tag{9}$$
$$i_0 = h(k_0)\%u_0 \tag{10}$$
$$i_1 = h(k_1)\%u_1 \tag{11}$$
$$\ldots$$
$$i_{m'} = h(k_{m'})\%u_{m'} \tag{12}$$
$$i_j = g(j) \quad \text{if } j \notin \{m, 0, 1, 2, \ldots, m'\}. \tag{13}$$

To prove $(i_0, \ldots, i_{M-1}) \in CS_m \cap CS_0 \cap CS_1 \cap \cdots \cap CS_m$, we only need to prove $(i_0, \ldots, i_{M-1}) \in CS_m$ since the relevant Map Functions are replaceable.

(1.1) For each cell in CS_m, the mth index S_m equals to $h(k_m)\%u_m$ based on Eq. (6). Given Eq. (9), we have $i_m = S_m$.

(1.2) Since $\exists RF_r, MF_0, MF_1, \ldots, MF_{m'} \in \mathrm{Rel}(MF_m, RF_r)$ based on Eq. (4), each element in $\{S_j\}$ in Eq. (7) covers all possible values. Thus, $i_0 \in \{S_0\}$, $i_1 \in \{S_1\}, \ldots, i_{m'} \in \{S_{m'}\}$.

(1.3) If $j \notin \{m, 0, 1, 2, \ldots, m'\}$, Then, $MF_j \in \mathrm{iRel}(MF_m, RF_r)$, thus $i_j = S_j = g(j)$ based on Eqs. (8) and (13).

Concluded above, we have $(i_0, \ldots, i_{M-1}) \in CS_m$. Thus, we prove Eq. (5).

(2) For the uniqueness, proof is given here.

$$f(k_0, v_0) \cap f(k_1, v_1) \cap \cdots \cap f(k_{m'}, v_{m'})$$
$$= \{S_m\} = \text{all of } \{0, \ldots, u_m - 1\} \tag{14}$$
$$S_0 = h(k_0)\%u_0 \tag{15}$$
$$S_1 = h(k_1)\%u_1 \tag{16}$$
$$\ldots$$
$$S_{m'} = h(k_{m'})\%u_{m'} \tag{17}$$
$$S_j = g(j) \quad \text{if } MF_j \in \mathrm{iRel}(MF_t). \tag{18}$$

Based on Eq. (1), the mth index of $f(k_m, v_m)$ is $h(k_0)\%u_0 \in \{S_0\}$;

Based on Eq. (2), the 0th, 1th, ... m'th index of $f(k_m, v_m)$ are "all of $\{0, \ldots, u_0-1\}$", "all of $\{0, \ldots, u_1-1\}$",... "all of $\{0, \ldots, u_{m'}-1\}$" respectively. Then, we have $S_0 \in$ all of $\{0, \ldots, u_0 - 1\}$, $S_1 \in$ all of $\{0, \ldots, u_1 - 1\}, \ldots, S_{m'} \in$ all of $\{0, \ldots, u_{m'} - 1\}$.

Based on Eq. (3), we have the same value for index at j where $MF_j \in \mathrm{iRel}(MF_t)$ in Eq. (18).

Concluded above, we have $|f(k_m, v_m) \cap f(k_0, v_0) \cap f(k_1, v_1) \cap \cdots \cap f(k_{m'}, v_{m'})| = 1$, and it is given by Eqs. (11)–(15).

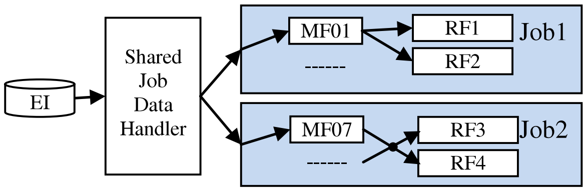


Fig. 4. Illustration of two CloudFlow jobs in scenario 2. Input dataset EI, which is shared by two different jobs, is handled by the Shared Job Data Handler.

Note: h is the function that maps the M-tuple to a processing node.

$h : (i_1, \ldots, i_M) \rightarrow n,\ n \in \{1, 2, \ldots, N\}$, where N is the total number of processing nodes.

4. Shared data handler

In this section, we introduce the shared data handler, which includes both the Shared Job and Shared Function Data Handlers.

4.1. Shared job data handler

The Shared Job Data Handler processes the jobs, which share the same input dataset. As being shown in Fig. 4, there are two concurrent jobs that share the same table: EI. In this case, we define a handler to orchestrate and manage the shared input data among multiple jobs. CloudFlow finds out that EI is frequently used. Therefore, it computes the input splits only once to feed the Map Tasks, and copies the shared input data to local file systems of the system nodes. Future jobs that need EI can read it from local nodes.

This handler is implemented in the CloudFlow framework since the programmer does not know how other jobs define their input dataset. In Algorithm 1, we define a mechanism to discover the frequently used data by recently submitted jobs. The steps of handling shared data are illustrated below.

```
Algorithm 1. Shared Job Data Handler
Input:
CloudFlowJobList[] CloudFlowJobList // Recent submitted jobs
Output:
SharedJobDataHandler sharedJobDataHandler
Procedure:
inputDataSetList = AnalyzeInputDataSet(CloudFlowJobList[i]);
Foreach inputData in inputDataSetList
        If isFrequentUsedData(inputData)
            sharedJobDataHandler.AddData(inputData);
            Copy inputData to Local File System;
EndFor
```

The procedure above demonstrates the steps to handle the shared input data among multiple MR jobs. In the first line, the input data of a new coming job is decomposed into an inputDataSetList. Then each input file in this list is iterated via a isFrequentUsedData() function. This function defines if the file is used frequently recently.

Different solutions can be applied to define the frequently used data. For example, the data with an accessing statistic average value above a threshold over all the submitted jobs can be deemed as frequently used. A shorter time window which considers the newest submitted jobs can also be applied. In our late experiment, we set the data as frequently used if the newest five submitted jobs have at least visited the input data three times.

Other than the replication, 'dereplication' can also be applied. Imagine the case in which some data has not been accessed by the five last jobs, but was previously marked as 'frequently used', the data can be deleted from local file system to save storage space of each individual nodes. This 'dereplication' strategy is not tested in this paper, which will be touched in our future work.

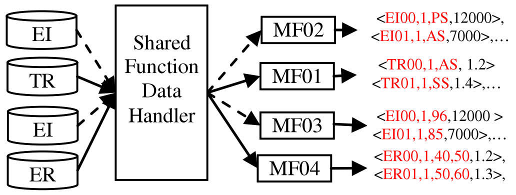


Fig. 5. Illustration of the Shared Function Data Handler for heterogeneous Map Functions in one job. Instead of the two Map Functions reading the shared data EI separately, the handler supplies EI to both MF02 and MF03.

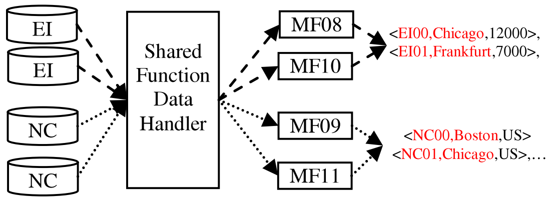


Fig. 6. Illustration of the Shared Function Data Handler for homogeneous Map Functions in one job. The handler supplies the shared data EI and NC to the Map Functions that need them. It also generate one copy of the output data.

4.2. Shared function data handler

In Fig. 5, we illustrate the Shared Function Data Handler. Since EI is shared by MF11 and MF13, it would be better to push the data to the two Map Functions instead of pulling the data by the two Map Functions themselves. In this case, the handler creates one copy of its implementation at each slave node where data EI is located, and supplies the data to the local Map Tasks of MF11 and MF12.

For a Map Task, the jobtracker takes into account the tasktracker's location and assigns a Map Task on the nodes where its input dataset is located. The Map Tasks of MF02 and MF03 that process the same split of data of EI can be co-located at the same node. Thus, the implementation of the Shared Function Data Handler can be located at the same host and serves the shared data for both MF02 and MF03.

In Fig. 6, we illustrate an optimization of the Shared Function Data Handler. Notice this handler is currently used for the same Map Function on the same input datasets. For example, two pairs of Map Functions (MF08&MF10 and MF09&MF11) perform exactly the same in case 2 of scenario 1. Since the input datasets EI and NC are shared, we still can use the method in Fig. 5 to allocate the handler.

Furthermore, instead of generating two copies of outputs for EI as shown in Fig. 2(b), the Shared Function Data Handler for Map Functions generates only one output copy. This one output does not have to include the identifier key "1" or "2" to indicate where the data is from. This is because the Reduce Function does not have to care about the source of the input data if the Map Functions are exactly the same.

This handler makes sure that the key–value pairs for EI are shuffled to Reduce Functions only once. This optimization makes the framework much more lightly weighted by reducing the overhead of producing and shuffling the intermediate data multiple times. The same approach applies to handling the NC table.

The Shared Function Data Handler is different from the Shared Job Data Handler. First, the latter discovers the shared data by calculating the recently used data among different jobs while the former does not have to. Second, the latter provides optimization by copying the shared data to the local file system, while the former supplies data to different Map Tasks. Algorithm 2 below illustrates the procedure of handling shared function data among multiple Map Functions. The time complexity is O($n \wedge 2$) when n is the length of the Map Function list.

In the algorithm, the input data of each Map Function in the mapFunctionList is derived. Then pairwise comparison of input data between all pairs of Map Functions (*Map_i* and *Map_j*) are carried out subsequently. If same input data is discovered, sharedInputMapClassList[*i*][*j*] is used to store the *Map_j* which shares the same input data as *Map_i*. *Map_i* and all the members in sharedInputMapClassList[*i*][] will be further handled by *SharedTaskDataHandler*[*i*].

Algorithm 2: Shared Function Data Handler

```
Input:
MapFunctionList[] mapFunctionList // All Map Functions
MapFunctionList[][] sharedInputMapFunctionList
Output:
SharedFunctionDataHandler[] sharedFunctionDataHandler
Procedure:
sharedInputMapFunctionList [1].AddMap(Map_1)
Foreach Map_i in mapFunctionList
    InputData_i =AnalyzeInputDataSet(Map_i);
    Foreach Map_j in mapFunctionList && (j > i);
        InputData_j =AnalyzeInputDataSet(Map_j);
        If(InputData_i ==InputData_j)
            sharedInputMapFunctionList[i].
             AddMap(Map_j);
    EndFor
EndFor
Foreach mapFunctionList in sharedInputMapFunctionList
    If (mapFunctionList.length > 0)
        sharedFunctionDataHandler[i] =
        SharedFunctionDataHandler(mapFunctionList);
EndFor
```

5. Experimental evaluation

In this section, we introduce the settings of our experiments and make performance comparisons between CloudFlow and the traditional MR approach which does not consider data sharing. We have not yet implemented the whole CloudFlow until now but we are evaluating the data sharing and optimization part. Here we are evaluating the efficiency of the data handler parts, which we have done.

5.1. Experimental settings

The experiments are carried out using 1, 2, 4 and 8 standard instances on Amazon EC2 cloud. Each instance is an m1.small with 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of local instance storage, 32-bit platform on deployed Ubuntu AMI, EBS boot, 32-bit architecture with Amazon EC2 AMI Tools.

We use two benchmark applications to demonstrate the effectiveness of the data sharing mechanism mentioned in CloudFlow programming model. For each application, we use small, moderate and large input dataset as testing cases to show the scalability against the input data size. We measure up to four jobs for each application.

Single Source Shortest Path (SSSP) is an iterative MR application that involves multiple runs of a Hadoop job. We use the tuple (number of graph nodes—nodes connectivity degree) to measure the graph size. For example, we use 10 000–0.01 to demonstrate the input graph with 10,000 nodes, and each node has $10{,}000 \times 0.01 = 100$ adjacent neighbors. Input data size is defined as small, moderate and large graphs with sizes of 10 000–0.01, 100 000–0.001, 1000 000–0.0001 respectively.

In this SSSP benchmark, we test the job-level concurrency, which runs multiple jobs over shared graph, we copy the shared data among multiple jobs to local file system to ensure data locality.

String Matching is the other benchmark we use. A very big size string (typically a few GB) is located in HDFS. This string is called a reference string. Multiple MR jobs with different users' strings, which are far shorter than the reference string, are used to match the reference string in order to find the number of occurrences of this substring in the reference string. This string matching benchmark is important in genome sequence alignment, where the string is a typical genome sequence.

For the string matching benchmark, we use both the shared data at job-level and function-level to test the efficiency of this framework.

5.2. Experimental results

In Figs. 7 and 8, we show the effectiveness of the Shared Data Handlers for the two benchmark applications under varied input data size. Fig. 7((a)–(c)) demonstrates the execution time in milliseconds for small, moderate and large input data size for SSSP benchmark. Fig. 8((a)–(c)) shows the execution time for String Matching benchmark. The execution time shown excludes the shared data copy time for CloudFlow.

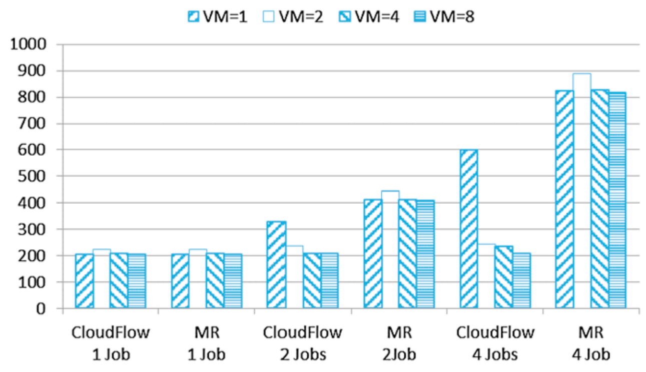
In both applications, the results demonstrate clear advantage of the CloudFlow data sharing approach as compared to the MR solution. Moreover, this advantage is much greater as the number of jobs increases.

In the one-job case, no matter how large the cluster size is, the execution times of the two methods are similar. This is because no concurrency can be utilized for the CloudFlow to take advantage of.

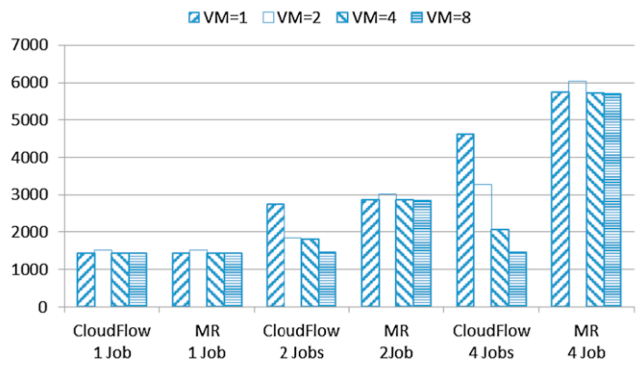
In the multiple job cases as shown below, we can see a performance speedup of up to 4X for the case of a cluster of size eight (VM = 8) and job number equals to 4 (see Fig. 7(a)). The traditional MR performs the job one by one, which delays the whole execution time of the four jobs. With 210 s in CloudFlow for running four concurrent jobs, the execution time is 816 s for running the four jobs in a standard MR manner. Similar results also apply to the same configuration in Fig. 7(b). Non-Concurrent MR takes 5709 s on moderate data size for the four jobs on eight VMs, while CloudFlow takes 1468 s instead.

Another parameter that affects the performance is the cluster size. Using larger cluster size normally shows better performance than using a smaller one. This benefit is better illustrated for larger input data size. In Fig. 7(a) and (b), the execution time of MR for the four-jobs case is very close across different cluster sizes. In the large input data size case, as shown in Fig. 7(c), the advantage of using varied cluster size appears.

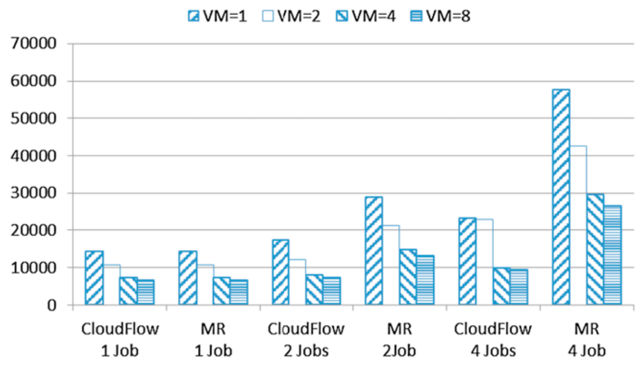
In CloudFlow, the large cluster size benefits over MR in SSSP cases. Except in using large input data size as in Fig. 7(c), the cluster size has very limited effect on MR as shown for the 2-job and 4-job cases. This does not apply to CloudFlow. This is easy



(a) Small input data size (10 000–0.01).

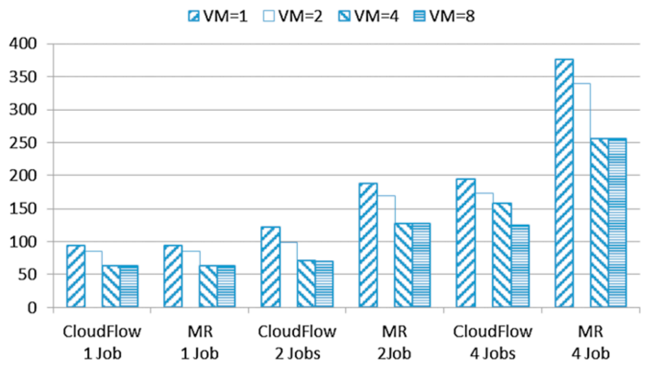


(b) Moderate input data size (100 000–0.001).

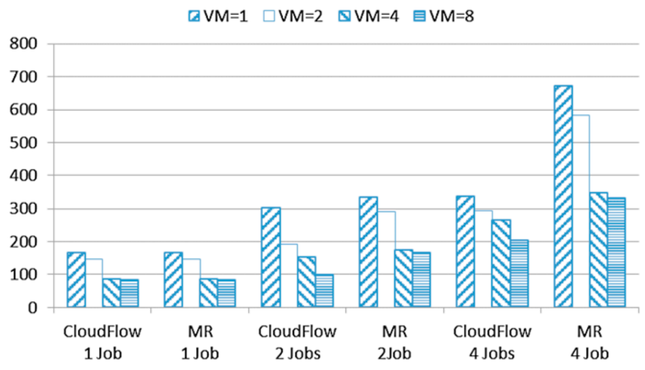


(c) Large input data size (1000 000–0.0001).

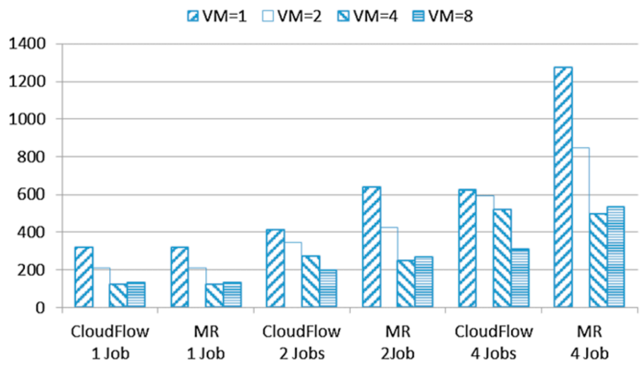
Fig. 7. Execution time (seconds) of CloudFlow running on SSSP benchmark.



(a) Small input data size (1 GB).



(b) Moderate input data size (2 GB).



(c) Large input data size (4 GB).

Fig. 8. Execution time (seconds) of CloudFlow running on string matching benchmark.

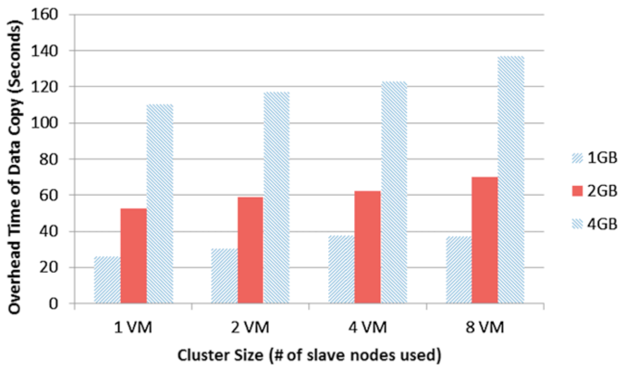


Fig. 9. Overhead time of copying the shared input data from HDFS to each individual VM instance.

to explain. CloudFlow explores more parallelism when running multiple jobs. Large cluster size would benefit more from this mechanism coupled with large input data size.

In short, CloudFlow performs better than its counterpart. It favors large number of jobs running on large cluster size. Parallelism in job-level and function-level data sharing are better exploited and utilized in this case.

In Fig. 9, we measure the overhead time used to copy the shared input dataset from HDFS to individual VMs. As we can see, copying 4 GB shared data to 8VMs in parallel takes around 140 s. This is a one-time overhead, which means the shared data would be copied only once. It benefits all the following concurrent jobs. As shown in Fig. 8, the more jobs we run, the more benefit we get from this data copying. For example in a 4 job case, CloudFlow has reduced the execution time more than 600 s at the cost of only 125 s for the shared data copying.

5.3. Discussion on scalability on big-data

The analysis above has proven the one-time overhead should have limited impact on the performance with multiple job runs. We argue that, therefore inevitably, the proposed framework should be scalable for larger input datasets. In other words, CloudFlow has divided the execution time into two parts, one for data moving and one for computing and shuffling. The data-moving portion is a fixed cost, and the much lower computing and shuffling portion is subject to the scaling of the big-data. Nevertheless, performance of traditional MapReduce is subject to the data variation without any fixed portion, and that it definitely scales much worse than the CloudFlow.

In Fig. 9, we can also learn that the overhead time in terms of scaling the cluster size is not manageable. This is understandable since moving a chunk of data to the HDFS of a four-VM cluster should not be significantly slower than moving the same amount of data to an eight-VM cluster. However, we do obverse a few slowdowns when a larger cluster is used. This is because larger cluster size would lead to the data spreading over more nodes. The maintenance, including the metadata management, data replication for availability, etc. over a larger cluster should incur more overhead than a small cluster size. This slightly larger overhead, compared with the significantly improved performance over the other computing and shuffling stages, still triumphs traditional MapReduce in general.

6. Related work

In the past, many endeavors have focused on proposing extended or new programming models for big-data problems other than the widely accepted MR. We are targeting this problem by exploring the data sharing at function/job level, and providing a framework to exploit the advantages of this sharing of data.

Efficient and effective programming models for different computing infrastructures have been of interest to both academia and industry. The advent of MapReduce [1] has spawned new research efforts on the development of programming models for big-data processing. Dryad [14,15] is proposed as an extension of MR. It employs Directed Acyclic Graph (DAG) to provide more flexibility to the programmer. SCOPE [16] is introduced by Microsoft, which is motivated by the structure of SQL, for efficient execution on very large clusters.

MR has also been enhanced by a number of research efforts. HaLoop [11], a revised programming model that integrates loop-based processing for MR applications, has demonstrated significant performance improvement when the input data is partially shared. Map-Reduce-Merge [17] introduces a ''merge'' stage after traditional MR to merge the outputs of multiple MR jobs.

The motivation for the two papers are similar to ours' in function-level concurrency. Unlike them, we explore data sharing in job-level concurrency as well to efficiently manage concurrent jobs submitted by different users.

Along the line of MR extensions, other research efforts addressed data streaming environments. Nova [18], due to its support for stateful incremental processing leveraging Pig Latin [19], deals with continuous arrival of streaming data. Incoop [20] is proposed as an incremental computation to improve the performance of MR framework. Deduce [21] and S4 [22] are other new programming models for streaming data applications.

Pregel [23] is a message-based programming model to deal with distributed large graph programs. It uses vertex, messages and multiple iterations in order to provide a completely new programming mechanism. GraphLab [24,25] is proposed to deal with scalable algorithms in data mining and machine learning that run on multicore clusters. Goal programming [26,27] approaches is also proposed for joint optimization of energy consumption and response time.

In our previous work, we have also identified quite a few interesting research merits along the MapReduce line, including the input dataset size scaling and its impact on performance [12], cluster size scaling and its impact on performance [38] and performance cost analysis on both public and private cloud [28]. Moreover, we have also identified the ConMR [29], a similar programming model but with much less theoretical proof and formulation.

In this paper, we focus on an improvement on traditional rigid map and Reduce Functions organized in any MR job by a more flexible framework, while exploiting job and function level to enhance the overall performance.

For a Reduce Function whose input datasets are produced by two or more Map Functions, joining among different datasets is widely applied in CloudFlow. There are a number of research papers to address the problem of joins using MR related solutions. Broadcast join is proposed in [30] for one dataset is relatively small. This dataset can be copied to each computing machine for conducting join operations at local file systems. Different join algorithms are compared in [31]. One-phase-join mentioned in [32] uses a filtering-join-aggregation model to deal with Map-Join-Reduce. Research papers [33,34] optimize multiple joins using Lagrange relaxation method. In [35], the authors propose matrix-based method to balance the input and output dataset size for different Reduce Tasks. In [36], Hilbert Space Filling Curve is proposed to deal with multi-way theta-join problem.

Restore [37] has been used to keep intermediate results for future workflow use. Unlike this method, our data reuse is share-aware and corresponding optimization is offered.

7. Conclusions and future work

In this paper, we propose CloudFlow, an improved MR programming model for modern HPC systems. CloudFlow addresses the need to define multiple independent MR sub-jobs to provide a solution to a complex problem. Besides this, CloudFlow reduces the amount of work by taking into account data sharing at the job and function levels. To summarizes the main contributions:

(1) We propose a concurrent MR framework CloudFlow with multiple heterogeneous Map and Reduce Functions. The shared data of different Map and Reduce Functions are defined by programmers to guarantee the data sharing is implemented in an efficient manner.
(2) CloudFlow takes into account the data sharing at the job and function levels. For job-level data sharing, the framework manages the data by replicating data to ensure data locality. For function-level data sharing, it provides optimization by supplying the shared data to multiple Map Tasks from different Map Functions. It also merges the output of the same Map Functions on the shared data.
(3) We have evaluated the framework using two benchmarks; Single Source Shortest Path and String Matching. Results have demonstrated up to 4X performance speedup compared to traditional MR.

Future work will focus on the following three directions.

(1) For the Shared Function Data Handler to process different Map Functions, we can optimize it further by merging the outputs of the intermediate key–value pairs. For example, we can merge the outputs of MF02 and MF03 in Fig. 2(a). The intermediate key–value pairs can be like <EI00, PS, 12 000, 96> et al.
(2) Explore the effectiveness of our method using a large cluster size on both a public cloud and a private cloud. Currently we are using up to eight instances on Amazon EC2. The advantage of the programming model will be further amplified given a larger cluster size.
(3) Work on the development of ConHadoop software toolkit, which supports the CloudFlow programming model with built-in support for job- and function-level optimization.

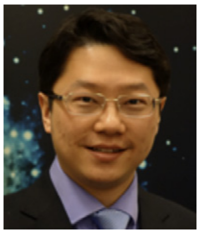
Acknowledgments

This publication was made possible by NPRP grant # 09-1116-1-172 from the Qatar National Research Fund (a member of Qatar Foundation). This work was also supported in part by Ministry of Science and Technology of China under National 973 Basic Research Program (Grant No. 2013CB228206), National Natural Science Foundation of China (Grant Nos. 61472200 and 61233016). The statements made herein are solely the responsibility of the authors.

References

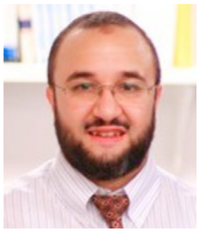
[1] R. Buyya, High Performance Cluster Computing, Prentice Hall PTR, NJ, USA, 1999.
[2] H. Jin, T. Cortes, R. Buyya, High Performance Mass Storage and Parallel I/O: Technologies and Applications, Wiley, 2001.
[3] K. Hwang, Z. Xu, Scalable Parallel Computing: Technology, Architecture, Programming, McGraw-Hill, Inc., New York, NY, USA, 1998.
[4] I. Foster, I. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, 2002.
[5] I. Raicu, I. Foster, Y. Zhao, Many-task computing for grids and supercomputers, in: IEEE Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS08, 2008.
[6] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proc. of 19th ACM Symp. on Operating Systems Principles, OSDI 04, December 2004, pp. 137–150.
[7] L. Wang, J. Tao, R. Ranjand, H. Martenc, A. Streitc, J. Chene, D. Chen, G-Hadoop: MapReduce across distributed data centers for data-intensive computing, Future Gener. Comput. Syst. 29 (2013) 739–750.
[8] T. Jie, H. Marten, A. Streit, S.U. Khan, J. Kolodziej, D. Chen, MapReduce across distributed clusters for data-intensive applications, in: The 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS, Shanghai, China, 2012.
[9] Hadoop. http://hadoop.apache.org/.
[10] R. Moraveji, J. Taheri, M.R.H. Farahabady, N.B. Rizvandi, A.Y. Zomaya, Data-intensive workload consolidation for the Hadoop distributed file system, in: The Proceedings of the ACM/IEEE 13th International Conference on Grid Computing, Grid 12, September 2012, pp. 95–103.
[11] Y. Bu, B. Howe, M. Balazinska, M.D. Ernst, HaLoop: efficient iterative data processing on large clusters, in: Proc. of the 36th International Conference on Very Large Data Bases, VLDB 10, September 2010, pp. 11–17.
[12] F. Zhang, M.F. Sakr, Dataset scaling and MapReduce performance, in: Workshop on Large-Scale Parallel Processing (LSPP'13) Held at the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Boston, USA, May, 2013.
[13] A. Gosavi, Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning, Springer, 2003.
[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: European Conference on Computer Systems, EuroSys 07, March 21–23, 2007, pp. 59–72.
[15] L. Popa, M. Budiu, Y. Yu, M. Isard, DryadInc: reusing work in large-scale computations, in: Workshop on Hot Topics in Cloud Computing, HotCloud 09, 2009.
[16] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, SCOPE: easy and efficient parallel processing of massive datasets, in: Proc. of the 34th International Conference on Very Large Data Bases, VLDB 08, August 2008, pp. 24–30.
[17] H. Yang, A. Dasdan, R. Hsiao, D.S. Parker, Map-Reduce-Merge: simplified relational data processing on large clusters, in: Proc. of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 07, June 2007, pp. 1029–1040.
[18] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V.B.N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, X. Wang, Nova: continuous Pig/Hadoop workfows, in: Proc. of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 11, June 2011, pp. 1081–1090.
[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin: a not-so-foreign language for data processing, in: Proc. of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 08, June 2008, pp. 1099–1110.
[20] P. Bhatotia, A. Wieder, R. Rodrigues, U.A. Acar, R. Pasquin, Incoop: MapReduce for incremental computations, in: Proc. of the 2nd ACM Symposium on Cloud Computing, SoCC 11, October 2011.
[21] V. Kumar, H. Andrade, B. Gedik, K.L. Wu, DEDUCE: at the intersection of MapReduce and stream processing, in: Proc. of the 13th International Conference on Extending Database Technology, EDBT 10, March 2010, pp. 657–662.
[22] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: distributed stream computing platform, in: Proc. of the International Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms, KDCloud 10, December 2010, pp. 170–177.
[23] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proc. of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 10, June 2010, pp. 135–146.
[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, GraphLab: a new framework for parallel machine learning, in: Proc. of the 26th Conference on Uncertainty in Artificial Intelligence, UAI 10, Catalina Island, July 2010.
[25] Y. Low, J. Gonzalez, A. Kyrola, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed GraphLab: a framework for machine learning and data mining in the cloud, J. Proc. VLDB Endow. 5 (2012) 716–727.
[26] S.U. Khan, A goal programming approach for the joint optimization of energy consumption and response time in computational grids, in: 28th IEEE International Performance Computing and Communications Conference, IPCCC, Phoenix, AZ, USA, December 2009, pp. 410–417.
[27] S.U. Khan, N. Min-Allah, A goal programming based energy efficient resource allocation in data centers, J. Supercomput. 61 (3) (2012) 502–519.
[28] F. Zhang, M.F. Sakr, Performance variations in resource scaling for MapReduce applications on private and public clouds, in: Proceedings of The 7th IEEE International Conference on Cloud Computing, IEEE Cloud 2014, Alaska, USA, June 2014.
[29] F. Zhang, Q.M. Malluhi, T. Elsayed, ConMR: concurrent MapReduce programming model for large scale shared-data applications, in: 42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October, 2013.
[30] S. Chen, Cheetah: a high performance, custom data warehouse on top of MapReduce, J. Proc. VLDB Endow. 3 (2010) 1459–1468.

[31] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, Y. Tian, A comparison of join algorithms for log processing in MapReduce, in: Proc. of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 10, June 2010, pp. 975–986.
[32] D. Jiang, A. Tung, G. Chen, Map-join-reduce: towards scalable and efficient data analysis on large clusters, IEEE Trans. Knowl. Data Eng. 23 (2010) 1299–1311.
[33] F.N. Afrati, J.D. Ullman, Optimizing joins in a Map-Reduce environment, in: Proc. of the 13th International Conference on Extending Database Technology, EDBT 10, March 2010, pp. 99–110.
[34] F.N. Afrati, J.D. Ullman, Optimizing multiway joins in a Map-Reduce environment, IEEE Trans. Knowl. Data Eng. 23 (2011) 1282–1298.
[35] A. Okcan, M. Riedewald, Processing theta-joins using MapReduce, in: Proc. of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 11, June 2011, pp. 949–960.
[36] X. Zhang, L. Chen, M. Wang, Efficient multi-way theta-join processing using MapReduce, in: Proc. of the 38th International Conference on Very Large Data Bases, VLDB 12, August 2012, pp. 1184–1195.
[37] I. Elghandour, A. Aboulnaga, ReStore: reusing results of MapReduce jobs, Proc. VLDB Endow. 5 (6) (2012) 586–597.
[38] F. Zhang, M.F. Sakr, Cluster-size scaling and MapReduce execution times, in: Proceedings of The International Conference on Cloud Computing and Science, CloudCom 2013, Bristol, UK, December 2013.



Fan Zhang is currently a postdoctoral associate with the Kavli Institute for Astrophysics and Space Research at Massachusetts Institute of Technology. He is also a sponsored researcher in Tsinghua University, Beijing, China. He has been appointed as a visiting associate professor in the Shenzhen Institute of advanced technology, Chinese Academy of Science since Jan 2014. He received his Ph.D. in Department of Control Science and Engineering, Tsinghua University in Jan. 2012. From 2011 to 2013 he was a research scientist at Cloud Computing Laboratory, Carnegie Mellon University. An IEEE Senior Member, he received an Honorarium Research Funding Award from the University of Chicago and Argonne National Laboratory (2013), a Meritorious Service Award (2013) from IEEE Transactions on Service Computing, two IBM Ph.D. Fellowship Awards (2010 and 2011). His research interests include big-data scientific computing applications, simulation-based optimization approaches, cloud computing, and novel programming models for streaming data applications on elastic cloud platforms.

Qutaibah M. Malluhi joined Qatar University in September 2005. He is the Director of the KINDI Lab for Computing Research. He served as the head of Computer Science and Engineering Department at Qatar University between 2005 and 2012. Before joining Qatar University he was a professor of Computer Science at Jackson State University where he served as a faculty member between 1994 and 2005. During 1995 and 1996, he was a research faculty at Lawrence Berkeley National Laboratory, Berkeley California. He was the co-founder and CTO of Data Reliability Inc. between 2001 and 2005. He was a Co-founder of Qloud (Qatar Cloud Computing Center); a collaboration between IBM, Qatar University, Carnegie Mellon University-Qatar, and Texas A&M University, Qatar (2009). He was also the Co-Founder and Executive Advisor for the Qatar University Wireless Innovation Center at the Qatar Science and Technology Park. His research area is in the fields of parallel and distributed high performance computing, Security & privacy and cloud computing.



Tamer M. Elsayed received the B.Sc. and M.Sc. degrees in Computer Science from Alexandria University in Egypt, and Ph.D. degree in Computer Science from the University of Maryland, College Park (UMD) in the United States in 2009. His main research interest is in information retrieval with an emphasis on web search and large-scale text analysis. He spent one year as a post-doctoral researcher at the Cloud Computing Center at UMD, where he participated in the design, development, and evaluation of an open-source retrieval engine called Ivory. In summer 2010, he joined the Advanced Systems Lab at King Abdullah University of Science and Technology (KAUST) as a post-doctoral fellow in the Division of Mathematics and Computer Science, where he worked on two research projects: asynchronous iterations support for MapReduce, and real-time search in Twitter. He joined Microsoft Advanced Technology Lab at Cairo in summer 2011 as a researcher before joining Qatar University in Fall 2012 as an assistant professor.

Samee U. Khan is an associate professor at the North Dakota State University. He received his Ph.D. from the University of Texas at Arlington in 2007. His research interests include optimization, robustness, and security of: cloud, grid, cluster and big data computing, social networks, wired and wireless networks, power systems, smart grids, and optical networks. His work has appeared in over 225 publications with two receiving best paper awards. He is a Fellow of the IET and a Fellow of the BCS.

Keqin Li is a SUNY distinguished professor of Computer Science and an Intellectual Ventures endowed visiting chair professor at Tsinghua University, China. His research interests are mainly in design and analysis of algorithms, parallel and distributed computing, and computer networking. He has over 255 research publications and has received several Best Paper Awards for his research work. He is currently on the editorial boards of *IEEE Transactions on Cloud Computing* and *IEEE Transactions on Computers*.

Albert Y. Zomaya is currently the Chair Professor of High Performance Computing and Networking and Australian Research Council Professorial Fellow in the School of Information Technologies, The University of Sydney. He is also the Director of the Centre for Distributed and High Performance Computing, which was established, in late 2009. He is the author/co-author of seven books, more than 400 papers, and the editor of nine books and 11 conference proceedings. He is the Editor-in-Chief of the IEEE Transactions on Computers and serves as an associate editor for 19 leading journals, such as, the IEEE Transactions on Parallel and Distributed Systems and Journal of Parallel and Distributed Computing. He is the recipient of the Meritorious Service Award (in 2000) and the Golden Core Recognition (in 2006), both from the IEEE Computer Society. Also, he received the IEEE Technical Committee on Parallel Processing Outstanding Service Award and the IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing, both in 2011. He is a Chartered Engineer, a Fellow of AAAS, IEEE, and IET (UK).