IntegrityMR: Exploring Result Integrity Assurance Solutions for Big Data Computing Applications

Yongzhi Wang [1], Jinpeng Wei [2], Mudhakar Srivatsa [3], Yucong Duan [4], Wencai Du [5]

[1] School of Computer Science and Technology, Xidian University,
2 South Taibai Road, Xi'an, Shaanxi, China
The Key Laboratory of Grain Information Processing and Control (Henan University of Technology),
Ministry of Education, Zhengzhou, Henan, China
yzwang@xidian.edu.cn

[2] School of Computer and Information Sciences, Florida International University,
11020 SW 8th Street, Miami, Florida, USA
weijp@cis.fiu.edu

[3] Network Technologies Department, IBM T.J. Watson Research Center,
1101 Kitchawan Road, Yorktown Heights, New York, USA
msrivats@us.ibm.com

[4,5] Information Science and Technology College, Hainan University,
No. 58, Renmin Avenue, Haikou, Hainan, China
[4] duanyucong@hotmail.com; [5] wencai@hainu.edu.cn

Abstract

Large-scale adoption of MapReduce applications on public clouds is hindered by the lack of trust on the participating virtual machines deployed on the public cloud. In this paper, we propose IntegrityMR, a multi-public clouds architecture-based solution, which performs the MapReduce-based result integrity check techniques at two alternative layers: the task layer and the application layer. Our experimental results show that solutions in both layers offer a high result integrity but non-negligible performance overheads.

Keywords: Big Data; MapReduce; Integrity Assurance; Cloud Computing

1. Introduction

Big data applications have drawn more attentions in recent years when MapReduce[1] and cloud computing techniques are getting mature. However, when the public cloud venders offer big data applications as public services, new challenges appear when retrospection is performed from the security perspective. MapReduce, the fundamental infrastructure of such a service, when deployed on the public cloud, suffers from the result integrity vulnerability. Given the distributed architecture of MapReduce, merely one malicious participant can render the overall computation result useless. This is because the cloud vendor is not responsible for the integrity of computations inside each virtual machine that runs MapReduce tasks. Specifically, due to the openness of public cloud architecture, customers have the freedom to choose virtual machine image provided by anybody, including the malicious provider. Sven et al. [21] pointed out a security vulnerability that Amazon EC2 suffers from: any member of the EC2 community can create and upload Amazon Machine Images (AMIs), which can be used by any EC2 user. If the AMIs are malicious and are widely used, it could flood the whole EC2 community with malicious applications, including MapReduce.

Our goal in this paper is to increase the result correctness of big data computing applications, known as the *result integrity*. To achieve our goal, we propose a novel architecture that combines the benefits of private clouds and public clouds. Our solution, named *IntegrityMR*, overlays the MapReduce framework on top of hybrid cloud. The master and a small number of workers called *verifiers* are deployed on the private cloud, while other workers are deployed on multiple public clouds. The workers on public clouds finish the majority of work. While the master and verifiers on the trusted private cloud control the correctness. The key rationale of our solution is to retain control "at home", while delegating the more resource-intensive computations to the public cloud. Such hybrid cloud architecture is enlightened by the previous work *Cross-Cloud MapReduce (CCMR)*[22]. However, we extend the idea into multiple public clouds environment. Since IntegrityMR assigns tasks to multiple public clouds, it raises a bar for the attackers who have to construct collusive malicious workers across multiple public clouds that can collude with each other to commit stealthier cheat.

We explore the design space of result integrity checking in two alternative layers of the MapReduce application software stack: the *MapReduce task layer* (we call it *task layer* in the following sections for brevity.) and the *application layer*. At the task layer, we resort to the techniques proposed in our previous work[18,22], and extend the techniques to multiple public clouds environment. We build a prototype system that support most big data applications. At the application layer, we make a case study on Pig Latin[8], a popular MapReduce based big data management application. We propose a technique to transform Pig Latin scripts to introduce *invariant* to map tasks and the *invariant check* to the succeeding reduce tasks.

To the best of our knowledge, this is the first paper proposing multiple public clouds architecture for MapReduce computation. Moreover, this is the first paper proposing to introduce invariant to the MapReduce job to guarantee result integrity. Specifically, our contributions are as follows.

- We propose a new hybrid cloud MapReduce architecture, IntegrityMR, which gains control at the private cloud and utilizes the computation capability from multiple public clouds.
- We explore task layer result integrity check technique by extending the techniques proposed in our previous works[18,22] to the multiple public clouds environment.
- We study the Pig Latin, and propose an application layer result integrity checking technique based on a script transformation technique.
- We make a prototype implementation of IntegrityMR on Apache Hadoop[7] and Pig Latin, and perform experiments on commercial public cloud (Amazon EC[22] and Microsoft Azure[3]) and local cluster to test the efficacy of both layer solutions.

The rest of this paper is organized as follows. Section 2 declares the system assumptions and attacker model. Section 3 presents the design, implementation, and evaluation of task layer result integrity checking approach. Section 4 describes the application layer result integrity checking approach using the Apache Pig as a case study. Section 5 discusses related work, and Section 6 concludes the paper.

2. Background and System Assumptions

2.1. System Assumptions

IntegrityMR overlays MapReduce on multiple clouds: one private cloud and multiple public clouds. We assume the private cloud is trusted since it is deployed in the user's organization. Hence the master and the verifiers that are deployed on the private cloud are trusted. We assume the public clouds are not trusted. Hence the workers that are deployed on the public clouds are not trusted. Since the integrity of the *Distributed File System (DFS)* can be guaranteed with the storage integrity assurance techniques[5,6], we assume DFS is trusted. Finally, we assume the infrastructures provided by the cloud provider, such as virtualized hardware and network, are trusted, although the virtual machine instances can be compromised. Overall, from the perspective of MapReduce, the only untrusted entities in the IntegrityMR environment are the workers deployed on public clouds.

2.2. Attacker Model

We model the attacker as a "powerful adversary" that controls a set of malicious nodes in each public cloud. It receives and shares information collected from malicious nodes and instructs a selected subset of malicious nodes to return incorrect result at the coordinated time in order to introduce as many errors as possible while not being detected. In other words, if two malicious workers are assigned to execute the same task, they can *collude* with each other. We call such malicious workers *collusive workers*.

3. Task Layer Result Integrity Check

We explore the task layer integrity check technique in this section.

3.1. System Design

IntegrityMR redefines the architecture of MapReduce: the master and verifiers, the trusted slave workers are deployed on the trusted private cloud within the customer's organization. The remaining slave workers are deployed on multiple public clouds. The verifiers are used to re-compute tasks to arbitrate inconsistent results and detect non-collusive workers, or to verify consistent results in a probabilistic manner to detect collusive workers. The master is responsible for assigning tasks and checking the consistency of the task results. Since the DFS is trusted, we deploy DFS across the multiple public clouds. The system architecture is shown in Fig. 1. We adapt the techniques proposed in our previous work[22] to meet the requirement of IntegrityMR. In the previous work[22], the authors propose CCMR, a single private cloud and single public cloud architecture. CCMR employs random replication,

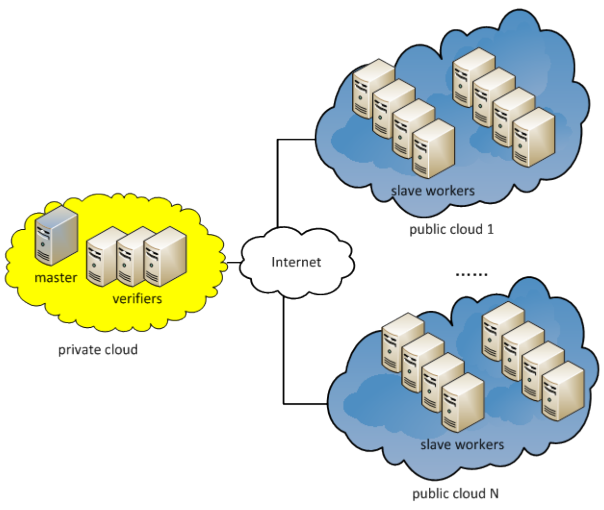

Fig. 1. The Architecture of task layer result integrity check

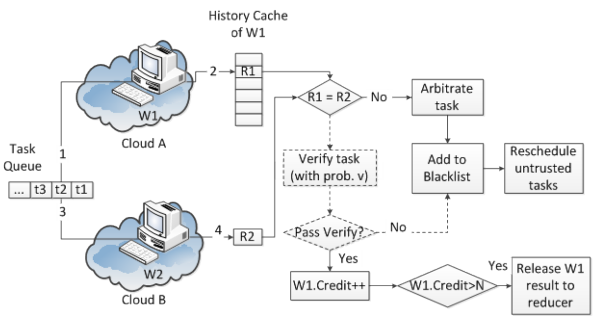

Fig. 2. The Control flow of task layer result integrity check

random verification and credit based management techniques on such MapReduce. We extend these techniques to the multiple public clouds architecture. In task assignment, instead of picking a worker from the single public cloud, IntegrityMR can randomly choose a public cloud and pick a worker from the chosen cloud. However, complete randomized task assignment would bear significant performance loss due to the existence of shuffle phase, where the mappers send their intermediate results to the reducer. If the mappers and the reducers are not in the same cloud, the data transmission would slowdown the overall computation. We solve this problem by having the master assign the original map tasks and reduce tasks to the same cloud, and the replicated map tasks and reduce tasks to another cloud. Since the reducer only accepts map results from the same cloud, shuffle would only happen inside a cloud.

The control flow of IntegrityMR is depicted in Fig. 2. Cloud A and cloud B are two clouds randomly picked from the available public clouds in the IntegrityMR environment. W1 and W2 are two slave workers randomly picked from cloud A and B respectively. The "Arbitrate task" and "Verify task" steps are executed by the verifier. Other components in the figure, including the Task Queue and the History Caches for the worker W1, are all maintained in the master. Whenever the Task Queue is not empty, the master will pick one task (e.g., t2) from the Task Queue and assign it (step 1) to two workers randomly picked from any two clouds among all the public clouds (W1 and W2 in cloud A and B, respectively). According to the *hold-and-test* strategy proposed in CCMR, the task is first assigned to W1. Only when the result R1 (hash value) is returned (step 2), the replicated task is assigned to W2 (step 3). By doing this, if the first worker W1 is a malicious worker, it cannot determine whether it is safe to cheat because it does not know whether W2 assigned in the future is a collusive worker or not. W1 stores the actual task result in its local storage and return the hash value of result R1 to the master, which is stored in the history cache of W1. When the result R2 is returned by W2 (step 4), the master will compare R1 and R2 to detect the malicious workers. If R1 and R2 are consistent, the master increments the credit of the first worker W1; otherwise, the task is arbitrated by the master (i.e., by asking a verifier to re-compute it) to identify which worker is malicious. If R1 and R2 are consistent, the consistent result is still verified with certain probability v (*verification probability*). If the verified result is different from R1/R2, both W1 and W2 are determined to be malicious workers. If the verified result is the same as R1/R2, the master increments the credit of worker W1. If the credit of W1 exceeds a certain value N (*credit threshold*), the task results buffered in W1's local storage are accepted by the master in *one result batch*. Meanwhile, its history cache is cleared and its credit is reset to 0.

A detected malicious worker will be added to the blacklist. And the tasks buffered in its history cache will be rescheduled. (i.e., putting the tasks back to the Task Queue)

3.2. Security Analysis

IntegrityMR has two lines of defense against the attacker. The multiple public clouds architecture raises the bar for the attackers. Since each task is replicated and assigned to two (or more) workers from different cloud service providers, successfully breaking in multiple public clouds is already non-trivial challenge for the attacker. Figuring out which virtual instance corresponds to a specific MapReduce job in each cloud and compromising them to construct collusion is even more difficult for the attackers.

Even if the first line of defense is breached, IntegrityMR can still protect the accuracy of computations by the techniques such as hold-and-test, verification and credit based trust management. Even though IntegrityMR includes multiple public clouds, the task assignment design is similar to the map phase integrity assurance design of CCMR[22]. Therefore, we can straightforwardly adapt the theoretical analysis result (Theorem 1) of CCMR. Given the space limit, we skip the theoretical accuracy and overhead analysis of IntegrityMR, and refer readers to the original paper of CCMR.

3.3. Experiment Environment

We implement the IntegrityMR based on the Apache Hadoop and deploy it on the environment consisting of one private cloud and two public clouds.

3. 3. 1 Environment Configuration

Our experiment environment consists of the following entities: a Linux server (2.93 GHz, 8-core Intel Xeon CPU and 16 GB of RAM) is deployed on the private cloud, running both the master and the verifier. 6 slave workers are running on Microsoft Azure extra small instances (Windows Server 2008 32-bit, 1 core@ 1GHz, 768MB Memory). Another 6 slave workers are deployed on the Amazon EC2 small instances (Amazon Linux AMI 32-bit, 1 ECU, 1 core, 1.7GB Memory). Since each Azure instance runs Windows system, we install Cygwin on each Azure instance so that Hadoop required SSHD service can work on it. Since the cloud provider assigns each virtual instance a unique URL, IntegrityMR uses such URL to identify different worker on public cloud. The topological information such as which worker is deployed on which cloud is statically configured in Hadoop configuration file (*mapred-site.xml*). In addition, the mapred-site.xml of IntegrityMR also configures the verification probability v and credit threshold N.

3. 3. 2 Collusive Worker Implementation

To test the effectiveness of IntegrityMR, we implement the collusive workers according to the analysis in CCMR: Since hold-and-test is applied, the collusive worker can only work as follows: when it receives a task, it first queries the adversary whether the same task (replicated one) has been executed previously by another malicious worker. If yes, the adversary will instruct the current malicious worker return the same result as the previous one. Otherwise, the current worker has to decide whether to cheat and send the decision back to the adversary. Our implementation assumes that the worker cheats with probability p when the adversary fails to give an instruction. We therefore define p as the *cheat probability*.

3.4. Experiment Result

Using the environment described in Section 3. 3. 1, we run a set of experiments to evaluate the effectiveness of IntegrityMR in terms of accuracy, overhead and performance overhead. Our test applications include not only example application such as Hadoop Word Count but also popular big data analytical application such as Mahout[23]. Specifically, our experiment tests the following Mahout applications: Mahout Bayes Classification, Canopy Clustering, K-means Clustering, Fuzzy K-means Clustering, and Dirlchlet Process Clustering. And all of these applications can run successfully under the IntegrityMR solution, showing a wide range of support for the big data applications.

3. 4. 1 Accuracy and Overhead

We measure both the computation accuracy and the overhead of the IntegrityMR in the task layer. Since in IntegrityMR, the map and reduce design share the same technique, our measurement only considers the map tasks. The reduce task measurement should have a similar result. We define the following metrics to measure the accuracy and overhead.

Error rate: The percentage of incorrect map task results accepted by the master in one job execution.

Worker overhead: The percentage of extra number of map tasks executed on the workers on public cloud in one job execution.

Verifier overhead: The percentage of map tasks executed by the verifiers on the private cloud in one job execution.

We assume that each execution of the same task consumes the same amount of resource (e.g., CPU time, memory, disk space, etc). Therefore, the worker overhead and the verifier overhead below represent the overhead of IntegrityMR across all resources. Comparatively, the performance overhead reported in Section 3. 4. 2 covers only the end-to-end execution time of the entire job.

We use Hadoop word count application to test the accuracy and overhead. The word count job computes the occurrences of each word in a batch of text input. In our experiment, each word count job consists of 100 map tasks and one reduce task. We introduce several collusive workers whose behavior is described in Section 3. 3. 2. In each job, we adjust the malicious node fraction n, cheat probability of malicious worker p, and credit threshold N. We set the verification probability v as constant value of 0.15.

We introduce the same number of malicious nodes in each public cloud. We vary the number of malicious workers from 1 to 3. Therefore, the values of n can be 0.15, 0.3 or 0.5. We pick four different environment configurations with different value of n and p, and test the effectiveness of IntegrityMR with different value of credit threshold N (1, 3, 5, 7, and 9).

Fig. 3 shows the error rate with different value of threshold under different environment configurations. In the figure, we learn that when N is increased from 1 to 9, the error rate in different environments decreases under all configurations. We also observe that when n is a constant (e.g., n=0.3) and N is small (e.g., N=1), a higher value of p (e.g., p=1.0) would bring a higher error rate (e.g., 12%) than an environment with a smaller value of p (e.g., p=0.5, 7% of error rate). However, with the increase of N, the error rate with a higher value of p decreases faster than the one with a smaller value of p. (e.g., when N=9, p=1.0, the error rate is 0%. When N=9, p =0.5, the error rate is 1%). Such changing trend is consistent with theorem 1 in CCMR paper[22].

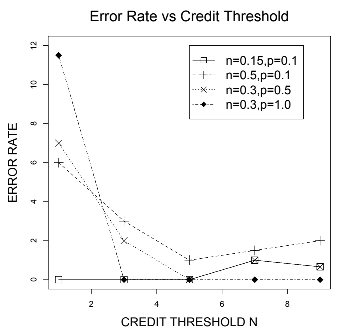

Fig. 3. The error rate of 100-map-task job

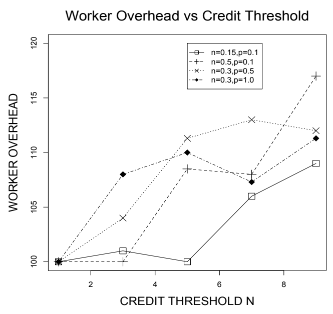

Fig. 4. The worker overhead of 100-map-task job

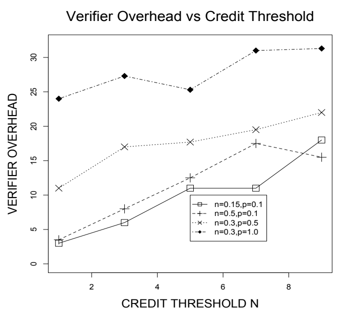

Fig. 5. The verifier overhead of 100-map-task job

Fig. 4 shows the worker overhead under different environment configurations. Under each environment configuration, the overhead increases with the increase of credit threshold N. Overall, the worker overhead ranges from 100% to 120%. Since each task is replicated, the 100% of overhead should be attributed to the replication. The remaining overhead should be attributed to the task reschedule due to the detection of malicious worker. Intuitively, a higher credit threshold means more task results are buffered in each worker, thus incurring more task reschedules if the worker is determined as malicious. We have to point out that the overhead will not grow unchecked with the increase of N. According to the analysis in CCMR, when N is big enough, the overhead will achieve its upper bound.

Fig. 5 shows that the verifier overhead ranges from 0% to 30% in different environments. Similarly, according to the analysis in CCMR, the verifier overhead will also achieve its upper bound when N is big enough.

3. 4. 2 End-to-end Performance Overhead

Our performance experiment does not contain malicious slave workers since the customers often feel it worthy to pay extra running time to detect errors. However, they are reluctant to pay more if the system does not contain error. We use Mahout 20 newsgroup example[11] to evaluate the performance of IntegrityMR. It classifies 20,000 news-group documents into 20 categories using naive Bayes classification. The classification algorithm is implemented in MapReduce.

We compare the execution time of such a job under three different environment settings shown as Table 1.

Table 1 Environment composition of performance test

Name	**Environment Composition**	**Cloud**	**Map Reduce**
Private-EC2-Azure	Private cloud with a Linux server, EC2 cloud with 6 small instances, Azure cloud with 6 extra small instances	Cross Cloud	Integrity MR
Private-Azure	Private cloud with a Linux server, Azure cloud with 6 extra small instances.	Cross Cloud	Map Reduce
Azure-only	Azure cloud with 6 extra small instances	Inside Cloud	Map Reduce

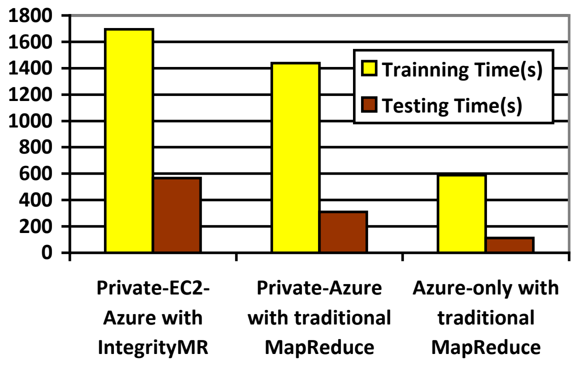

Fig. 6. Running time of Mahout 20 newsgroup classification

The Private-EC2-Azure environment is using IntegrityMR. The other two are using the traditional MapReduce. In terms of the cloud environment, the Azure-only platform uses a single cloud, while the other two use multiple clouds (cross-cloud environment).

In this set of experiments, we fix the value of N as 5, and set value of v as 0.15. We run the same job 5 times on each environment. The running time is shown in Fig. 6.

In the Azure-only environment where the traditional MapReduce is applied and the homogeneous cloud environment is used for communication inside of the cloud, the running time is the shortest (588s for the training job and 112s for the testing job). However, in the Private-Azure environment, where IntegrityMR is not yet used but the heterogeneous environment requires the cross-cloud communication, the running time increases to 1439s and 310s respectively. The increase percentages from the homogenous environment to the heterogeneous environment are 145% and 177%, respectively. When IntegrityMR is applied to the Private-EC2-Azure environment, the running times increase to 1694s and 567s, respectively. Compared with the Private-Azure environment, the increase of running time in the Private-EC2-Azure is 18% for the training time and 82% for the testing time.

It is necessary to point out the limitations of our implementation and experiments. Our implementation directly uses existing DFS implementation, which indistinguishably distributes storage on two public clouds. The synchronization among DFS nodes on two public clouds becomes a bottleneck and reduces the performance. By redesigning DFS to reduce such synchronizations would improve the performance. Our experimental setup only employs two public clouds. It is interesting to observe the result in a more general experimental environment, which introduces more public clouds. The improvement of the above limitations will be our future work.

4. Application Layer Result Integrity Check: The Pig Case Study

The accuracy of task layer integrity assurance is probabilistically determined by the credit threshold N. The small value of error rate is guaranteed when N is big enough. For example, in Fig. 3, when N is set to 1, the error rate can be as high as 12%. Besides, the overhead of task layer integrity checking is non-negligible. We propose to transform the MapReduce application to introduce invariants to the map tasks. By checking invariants during the job execution, we can indirectly infer whether the participating nodes are cheating or not.

Unfortunately, different MapReduce applications have different characteristics. Thus it is difficult to define a universal technique for the invariant insertion that is suitable for every application. Therefore, we narrow down our exploration to the major classes of MapReduce applications. One particularly interesting and widely used class is big data management application. Apache Pig[8] is one of such application widely used in both academia and industry for applications such as log analysis and data management. We choose Apache Pig as a case study to show that by transforming the Pig script, we can construct invariant in the map tasks and introduce the invariant check in the reduce tasks with minimum effort.

4.1. Background

Pig Latin is a scripting language designed to mimic the declarative style of SQL. The accompanying system, Pig, can compile Pig Latin script into physical plans that are executed over Hadoop[9].

In Pig Latin, a user specifies a sequence of steps via a Pig Latin script, where each step specifies only a single, high-level data transformation. The data organization in Pig consists of four data types: *atom*, atomic value such as string or number; *tuple*, a sequence of fields, equivalent to data record in traditional database; *bag*, a collection of tuples; *map*, a collection of data items where each item has a key with the type of atom and a value with the type of data bag. Pig Latin has ample and flexible keywords and operators that can meet most data manipulation requirements. Here we pick some relational operators related with our discussion and list them in Table 2. The full version of grammar manual can be found in the language menu[10].

The execution of Pig Latin script is a series of transformations of execution plans. Pig parses a Pig Latin script and translates it into a *logical plan*. Based on the logical plan, it translates it into a *physical plan*. And finally, it translates the physical plan to a

MapReduce job plan. All the plan translations are finished on the master. After that, the master assigns the tasks in the job plan to the workers.

Table 2 Selected operators in Pig Latin

Command	Explanation
LOAD	Load data from the file system. Return a data bag, each tuple is in a format as specified.
FOREACH GENERATE	Projection and aggregate each tuple in the bag, remove unspecified field and aggregate field data (which is usually built by the previous GROUP command) as specified.
FILTER BY	Drop tuples in the bag that does not satisfy the condition.
GROUP	Equivalent of SQL GROUP BY command. It will return tuples. Each tuple represents a distinct group. In each tuple, the first field is the group value, the second is a bag with all the input tuples in that group.
COGROUP	Group multiple data sets with common field. Suppose N data sets 1,2,…N are COGROUPed, it will return a bag, each tuple in the bag represents distinct group. In each tuple, the first field is the group value, followed by N bags. In the N bags, the ith bag contains tuples from the ith data set, but belonging to that group.
DUMP/STORE	Display the result/Store the result to the file system.

4.2. Invariant construction and check

We propose an invariant constructing and checking method for applications written in Pig Latin. Our method is to transform the original Pig Latin script into another equivalent script. By equivalent, we mean the two scripts will generate the same results. However, the job plans corresponding to the two scripts are different. In the transformed script, each original map plan will be substituted by two map plans. The two map plans will operate on some "overlapped" input data. In other words, a portion input of the two substituting map plans will be the same. As a result, when the two map plans are executed, their results should agree on the part corresponding to the overlapped input. This is the invariant the map task output should obey. Meanwhile, the original reduce plan is transformed to introduce the invariant check. The reduce task will not only check if the invariant is violated, but also restore the output data

```
-- Script 1: GROUP data in houred.txt by hour
raw_data = LOAD './houred.txt' USING PigStorage('\t')
                 AS (user, hour, query);
result = GROUP raw_data BY hour;
dump result;
```

Fig. 7. Script1: Group data in houred.txt by hour

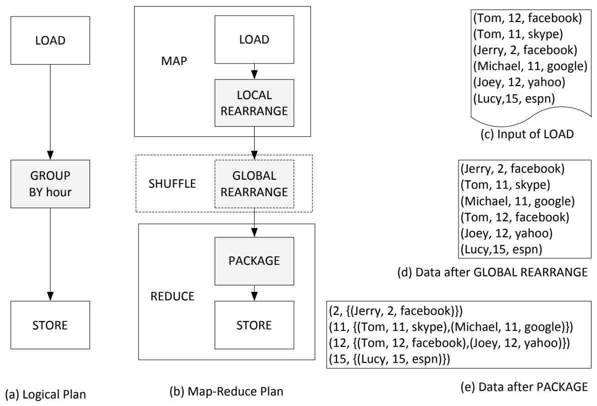

Fig. 8. Execution process of Script 1

so that the reduce task result will be the same as the original script result. If any invariant violation is detected during the reduce task execution, the reduce task will throw an exception indicating that some map task outputs contain errors.

Since we rely on the reduce task to check the invariant, we assume that the reduce task is trusted. To achieve high assurance of the reduce task, IntegrityMR can assign the reduce task to the verifiers on the private cloud. Alternatively, IntegrityMR can apply the task layer checking techniques in section 3 to each reduce task.

We create overlapped input data for the two new map plans by inserting an FILTER statement to each map plan, and having the conditions of the two FILTER

```
-- Script 2: invariant check is enforced
register ./tutorial.jar;
raw_data = LOAD './houred.txt' USING PigStorage('\t')
              AS (user, hour, query);
part1 = FILTER raw_data BY hour>=12;
part2 = FILTER raw_data BY hour<=12;
result = COGRUP part1 BY hour, part2 BY hour;
group_result=FOREACH result GENERATE
              group, org.apache.pig.tutorial.CheckInvariant($1,$2);
```

Fig. 9. Script 2: Script with invariant check

statements overlapped. For the invariant check, we develop a function and add an FOREACH statement to the script to invoke such a function for each record. Pig will automatically generate reduce plan to execute such FOREACH statement.

As a concrete illustration, we show how our method transforms a Pig Latin script (Script 1) in Fig. 7, which contains the relation operator GROUP, into another form (Script 2 in Fig. 9) to enforce invariant check. We use a text file hour.txt in Fig. 8 (c) as a running example of input file, which contains records about which users at which hour access what websites. The information is recorded in the fields “user”, “hour”, and “query” of a table. Our example Pig application aggregates the data in houred.txt by the field “hour” so that we can know the web access record for each specific hour.

As we can see from Fig. 7, Script 1 uses the GROUP command to implement the job. We show the logical and MapReduce plans of Script 1 in Fig. 8 (Since the physical plan is irrelevant to our discussion, we omit it to save space). The transformation from Script 1 to the logical plan is straightforward: each statement in Script 1 corresponds to one step in the logical plan. However, in the MapReduce plan, the GROUP statement is broken down into 3 steps: local rearrange, global rearrange, and package. The MapReduce plan also indicates whether each step should be processed in the map phase or the reduce phase: In our example, the local rearrange is executed by the mapper, and the package is executed by the reducer. The global rearrange is automatically carried out in the shuffle phase. Suppose the data contained in houred.txt is as Fig. 8 (c), after the local rearrange and the global rearrange, the data is transformed as shown in Fig. 8 (d). In the package step, the data is aggregated into different groups according to the field “hour”. Therefore, the final output is shown as in Fig. 8 (e).

In order to introduce invariant checks, we transform Script 1 into Script 2 (Fig. 9) that will generate an equivalent result. In Script 2, the raw_data is first split into two parts by applying two FILTER operations with different conditions (hour>=12 and. hour<=12, respectively). Since the conditions of the two FILTERs are overlapped at hour=12, part1 and part2 both contain the records with “hour” equal to 12 (highlighted records in Fig. 10 (c) and Fig. 10 (d)).

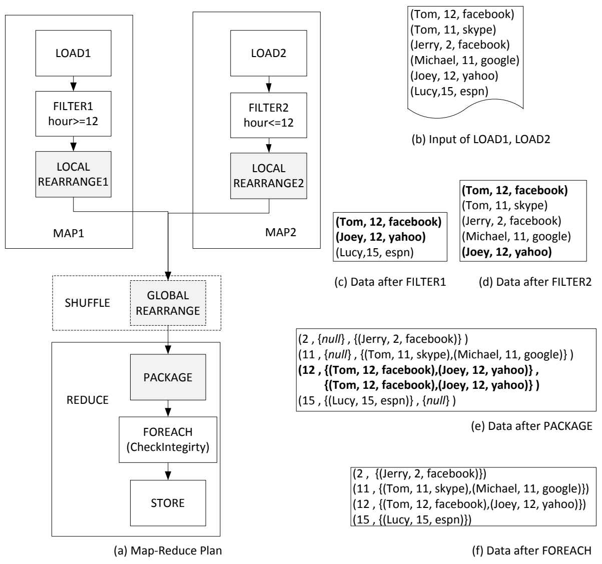

Fig. 10. Execution process of Script 2

Since part1 and part2 overlapped in the records with hour=12, the invariant of the map task output is that the records with “hour” as 12 should be the same. Then by applying COGROUP on part1 and part2 in Script 2, Pig will generate the result as in Fig. 10 (e). The COGROUP takes three steps to finish: Local rearrange, global rearrange and package. (Gray boxes in Fig. 10 (a).) After the package step of COGROUP, the result map should have duplicate bags when the key is 12. After that, each COGROUP result bag is processed by the CheckInvariant function in the FOREACH statement. The CheckInvariant function processes each record in Fig. 10 (e). If the key is 12, it will check if the bags inside the record are the same. If yes, it will delete one duplicated bag. If not, it will throw an exception. If the key is not 12, it will remove the empty bag. By doing this, the data is changed to Fig. 10 (f), which is the same as the original script result.

4.3. Security Analysis

In this section, we give an informal argument about how effective our method can defeat malicious mappers. The invariant property injected to the script involves the application domain knowledge. Since the worker on the public clouds only works on the MapReduce layer, the attacker needs to translate the MapReduce layer semantic into Pig script layer semantic and infer the invariant, which is a very challenging job. Furthermore, if the invariant check (reduce task) is performed on the trusted worker, the malicious workers have no way to access the invariant check logic. Therefore, they can only guess the checking logic from the transformed map tasks. Finally, if the map tasks passed to the worker is obfuscated byte code, the attacker has to perform online reverse engineering, which is even difficult for the attacker.

4.4. Performance measurement

We implement a prototype system based on Hadoop and test the performance slowdown between Script 1 and Script 2. We launch our experiment on a Linux Server with 2.93 GHz, 8-core Intel Xeon CPU and 16 GB of RAM. We deploy 3 virtual machines (512MB of RAM and 40GB of disk each) on VMware Workstation 7.11 to construct a MapReduce environment. Each machine runs on Debian 5.0.6 "lenny". Out of the 3 nodes, one is running as both a master and a trusted worker; the other two are running as untrusted workers. We compare the execution time of Scrip 1 and Scrip 2 with different input size. The result is shown in Fig. 11. We can see that when the input size is small (e.g., 31MB), the slowdown is negligible. When the input data size increases towards 372MB, the slowdown also increases to about 35%. The reason for such a slowdown is that the number of map tasks is doubled and each reduce task has to check the invariant on the overlapping part. However, since in Script 2, each map task only processes partial data, the slowdown is moderate.

It is necessary to point out that our system implementation and experiments has their limitations. Firstly, the transformation of Pig script is performed manually. Designing and implementing a system that automates such process would increase the practicality of our idea. Secondly, our experiments lack scalability test. It is interesting to observe the performance change with more worker nodes and bigger input data set. Improving the above limitation will be our future work.

5. Related Work

Result integrity assurance of distributed system has been discussed for decades. Several existing techniques such as replication, sampling, and verification solution

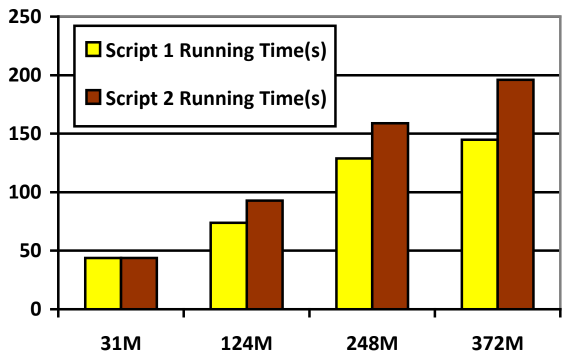

Fig. 11. Performance comparison between Script 1 and Script 2

have been proposed to address integrity issue in various distributed environments such as P2P Systems and Grid Computing[12-17].

Result integrity assurance of MapReduce is becoming popular in recent years. Wei et al.[4] proposed an integrity assurance framework SecureMR to enforce the *commitment protocol* and the *verification protocol*. By using non-deterministic duplication, it is effective of defeating malicious worker only when the malicious worker ratio is very small. Wang et al. proposed two MapReduce integrity assurance frameworks: VIAF[18] and CCMR[22]. The latter tried to solve the problem in a practical cloud environment with hybrid clouds. Moreover, CCMR[22] proposed different design to the map and the reduce phase given the different characteristic of different phases.

Compared with VIAF and CCMR, this paper proposes a new architecture which employs multiple public clouds to further enhance the security. Moreover, to the best of our knowledge, this paper is the first paper proposing the idea of application layer MapReduce integrity assurance.

6. Conclusion and Future Work

We present the design, implementation, and evaluation of IntegrityMR, an integrity assurance solution for big data applications. It overlays MapReduce on top of hybrid clouds which consists of one trusted private cloud and multiple public clouds. In order to perform result integrity check, we have explored the design space in two layers of the MapReduce software stack: the task layer and the application layer. Our experimental result in the task layer approach shows high integrity (98% with a credit threshold of 5) with non-negligible performance overhead (18% to 82% extra running time compared to original MapReduce). Our experimental result in the application layer approach shows improved performance compared with the task layer approach (less than 35% extra running time compared with the original MapReduce).

Our future work lies on the following two directions. In the task layer check, we will improve system performance by reducing cross-cloud communication and alleviate the DFS bottleneck. In the application layer check, we will work on the automating of the Pig script transformation.

Acknowledgement

This paper is supported in part by the Open Fund of the Chinese Key Laboratory of the Grain Information Processing and Control (under the Grant No. KFJJ-2015-202) and NSFC No. 61363007.

References

1. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Communications of the ACM, 51 (1): 107-113, 2008.
2. “Amazon Elastic Compute Cloud (Amazon EC2)” http://aws. amazon.com/ec2/
3. “Windows Azure Compute”, https://www.windowsazure. com /en-us/home/features/compute
4. Wei Wei, Juan Du, Ting Yu, Xiaohui Gu, “SecureMR: A Service Integrity Assurance Framework for MapReduce”, in Proceedings of the 2009 Annual Computer Applications Conference.
5. Kevin D. Bowers, Ari Juels, and Alina Oprea, “HAIL: a high-availability and integrity layer for cloud storage”, in Proceedings of the 16th ACM conference on Computer and communications security (CCS '09).
6. Raluca Ada Popa et al., “Enabling security in cloud storage SLAs with Cloud Proof”, In Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX ATC'11).
7. “What is Apache Hadoop”, http://hadoop.apache.org/
8. “Apache Pig”, http://pig.apache.org/
9. C. Olston et al. “Pig Latin: A not-so-foreign language for data processing”. In SIGMOD, 2008.
10. “Pig Latin Basics”, http://pig.apache.org/docs/r0.10.0/ basic.html
11. “20 newsgroups classification example”, https:// cwiki.apache.org/ confluence/display/MAHOUT/Twenty+Newsgroups
12. S. Zhao, V. Lo, and C. Gauthier Dickey, “Result verification and trust based scheduling in peer-to-peer grids,” in P2P ’05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing. Washington, DC, USA.
13. W. Du, J. Jia, M. Mangal, and M. Murugesan, “Uncheatable grid computing,” In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS’04), Washington, DC.
14. P. Golle and S. Stubblebine, “Secure distributed computing in a commercial environment,” in 5th International Conference Financial Cryptography (FC). Springer- Verlag, 2001, pp. 289–304.
15. P. Golle and I. Mironov, “Uncheatable distributed computations,” in CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology. London, UK: Springer-Verlag, 2001, pp. 425–440.
16. Petros Maniatis, David S. H. Rosenthal, Mema Roussopoulos, Mary Baker, TJ Giuli, and Yanto Muliadi. “Preserving peer replicas by rate-limited sampled voting”. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03).
17. Nikolaos Michalakis, Robert Soul, and Robert Grimm. “Ensuring content integrity for untrusted peer-to-peer content distribution networks”. In Proceedings of the 4th USENIX conference on Networked systems design & implementation (NSDI'07).
18. Yongzhi Wang, Jinpeng Wei, “VIAF: Verification-based Integrity Assurance Framework for MapReduce”, in the 4thIEEE International Conference on Cloud Computing (CLOUD 2011).
19. “Amazon Elastic MapReduce (Amazon AMR)”, http://aws.amazon.com/elasticmapreduce/
20. “Cloud Security: Amazon’s EC2 serves up 'certified pre-owned' server images” http://dvlabs.tippingpoint.com /blog/2011/04/11/ cloud-security-amazons-ec2-serves-up-certified-pre-owned-server- images
21. Sven Bugiel et al., “AmazonIA: when elasticity snaps back”, in the 18th ACM conference on Computer and Communications Security.
22. Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa, “Result Integrity Check for MapReduce Computation on Hybrid Clouds” in the 6th IEEE International Conference on Cloud Computing (CLOUD2013).
23. “What is Apache Mahout”, http://mahout.apache.org/