An Improvement of a Checkpoint-based Distributed Testing Technique on a Big Data Environment

Bhuridech Sudsee, Chanwit Kaewkasi

School of Computer Engineering

Suranaree University of Technology, Nakhon Ratchasrima, Thailand, 30000

m5741861@g.sut.ac.th, chanwit@sut.ac.th

***Abstract*— The advancement of storage technologies and the fast-growing number of generated data have made the world moved into the Big Data era. In this past, we had many data mining tools but they are inadequate to process Data-Intensive Scalable Computing workloads. The Apache Spark framework is a popular tool designed for Big Data processing. It leverages in-memory processing techniques that make Spark up to 100 times faster than Hadoop. Testing this kind of Big Data program is time consuming. Unfortunately, developers lack a proper testing framework, which cloud help assure quality of their data-intensive processing programs while saving development time and storage usages.**

We propose *Distributed Test Checkpointing (DTC) for Apache Spark.* DTC applies unit testing to the Big Data software development life cycle and reduce time spent for each testing loop with checkpoint. By using checkpoint technique, DTC keeps quality of Big Data processing software while keeps an inexpensive testing cost by overriding original Spark mechanism so that developers no pain to learn how to use DTC. Moreover, DTC has no addition abstraction layers. Developers can upgrade to a new version of Spark seamlessly. From the experimental results, we found that in the subsequence rounds of unit testing, DTC dramatically speed the testing time up to 450-500% faster. In case of storage, DTC can cut unnecessary data off and make the storage 19.7 times saver than the original checkpoint of Spark. DTC can be used either in case of JVM termination or testing with random values.

***Keyword*— Distributed Checkpointing; Apache Spark; Big Data Testing; Software Testing;**

I. INTRODUCTION

THE increasing and diversity of electronic devices, sensors, IoT devices and the fast-growing numbers of Internet users have been generating tremendous amount of data recently. They are not only the large amount of data but their structures are also complex as well. This complexity makes the traditional data mining tools inadequate to manage today's data [1].

The MapReduce [2] programming model has induced the development of many frameworks such as Apache Hadoop [4], Map-reduce-merge [5] and Apache Spark [6], which aim to process data intensive tasks. Developers only need to rewrite their programming logic in the form of *map* and *reduce* functions in order to process data on a MapReduce framework. These functions will be automatically managed by the framework's default configuration. This mechanism makes the MapReduce framework easy to use. At its simplest form, a MapReduce program usually starts by a *map* function creating key/value pairs from the input. These intermediate key/value pairs are then passed to a *reduce* function to produce the final results. The MapReduce model is parallel by nature. It is designed to allow developers to run MapReduce programs for high performance computing jobs using a commodity cluster, built from low-cost hardwares. With this kind of the cluster architecture, we can handle massive amount of data and process them on numerous cluster nodes without a single point of failure [3].

Although the MapReduce model is easy to use for software development, but it is quite tricky to test software written by the MapReduce model. Software testing is a vital part of the development process. Testing is usually 25-50% of the overall cost [8]. We found that the current mechanism is not enough to assure quality for Big Data processing programs. Unit testing is a software testing technique which properly leads to better levels of quality. However, tools like Scalatest[9] or jUnit[10] have their own limitations to use with a MapReduce framework like Spark. For example, SparkContext and SparkSession objects must be instantiated only once for each running Java Virtual Machine (JVM) to avoid unexpected testing results [12]. Spark-testing-base [11] also does not have a testing mechanism for Spark. Without modification, it cannot work on a Spark cluster because if its inability to distribute class files across worker nodes. There aforementioned techniques are not suitable for Spark simply because they are not designed to test programs that distributelly process large amount of data.

Test-driven development (TDD) is a software development technique that helps developers to focus on

---

Manuscript received December 27th, 2017. This work was supported by Suranaree University of Technology, and a follow-up of the invited journal to the accepted & presented paper of the 20th International Conference on Advanced Communication Technology (ICACT2018),

Bhuridech Sudsee is with School of Computer Engineering, Suranaree University of Technology, Nakhon Ratchasrima, Thailand (corresponding author phone: +66-44-22-4422; e-mail: m5741861@g.sut.ac.th).

Chanwit Kaewkasi is with School of Computer Engineering, Suranaree University of Technology, Nakhon Ratchasrima, Thailand (e-mail: chanwit@sut.ac.th).

writing a specific test at a time. It additionally allows code improvement while preserving correctness according to the specification. TDD workflow consists of the following steps, (1) writing a minimum test (2) writing codes to just make the test passed, and (3) refactoring to remove unnecessary codes while still making the current test passed [13]. We call these steps a TDD workflow herein this paper. Applying TDD to data intensive programs is difficult due to the nature of workloads, which need to process on a cluster. So, developers require a special tool to help shorten each loop of the TDD workflow.

Spark has *cache*, *persist* and *checkpoint* methods to help mitigate job failure. These mechanisms however do not help software testing process much. The main reason is that a cluster state cached or persisted by them does not survive across runs of JVMs. A cluster state saved by the *checkpoint* method does survive on disk but unfortunately it cannot be retrieved back by a newly started JVM [14, 15].

In this paper, we present Distributed Test Checkpointing (DTC), a technique that leverages the checkpoint technique to enhance software testing for data intensive jobs. With DTC, developers can increase productivity when testing their software on a distributed cluster repeatedly. DTC applied a hash function on each data partition of a Resilient Distributed Datasets (RDD) [18] to use an identifier. Modification of an RDD or a Dataset can be traced by the hashed number. The testcase that uses the RDD is also hashed at the bytecode level. Combining these techniques, DTC is found to reduce testing time and storage required by checkpointing significantly compared to the original Spark's checkpointing technique.

The remaining of this paper is organized as followed. Section II discusses related works, including Apache Spark. Section III presents the design and internal mechanism of DTC. Section IV presents the system architecture of the cluster used by our experiments, and the experimental results. This paper then ends with conclusion and future works in Section V.

II. Background and Related work

A. Apache Spark

Spark is a data intensive processing framework focusing on in-memory data processing [6], which is implemented in the form of Resilient Distributed Dataset (RDD) [18]. RDD is designed to take care of the data flow and handle the processing mechanism. An RDD could be created using one of the following methods (1) reading data from file (2) parallelizing collection in the driver program (3) transforming from another RDD (4) and by transforming back from a persisted RDD [6]. An RDD comprises with two kinds of command, *transformations* and *actions*. A transformation command transforms an RDD to another RDD. These commands are *map*, *filter* and *groupByKey*, for example. Another set of commands are actions, which are *collect* and *count*, for example. An RDD keeps all previous transformation inside itself. This direct acyclic graph of transformation is known as *lineage*. The beginning of the real computation occurs only when an action is called. This is the lazy evaluation nature of Spark.

A mechanism for failure recovery that helps an RDD to resume the processing without re-computation from scratch are methods such as *cache*, *persist* and *checkpoint*. The *cache* method uses persistency at MEMORY_ONLY, while the *persist* method has several levels of persistency. The checkpoint method, in contrast, uses the technique which save data onto a reliable storage, such as HDFS, Amazon S3 or Ceph. An RDD is usually cached or persisted during its computation to avoid re-computation previous steps [15].

The checkpoint technique is also applicable for Spark Streaming because it truncates the internal lineage, so the RDD does not need to knowledge of its parent. However, this mechanism is not designed for software testing. The re-computation is still required to start from the beginning when the testcase is re-run. The rerunning of the testcase destroys a Block Manager inside an Executor. This Block Manage is responsible for keeping cached and persisted data. The new Driver program and the testcase therefore is not able to access the location of checkpoints.

In addition, Spark has introduced the Dataframe API in 1.3 and Dataset in 1.6. Both abstractions can be used interchangeably because Dataset[Row] is the type safer version of DataFrame. A dataset is also convertible to an RDD. In the case of DTC proposed in this paper, we read and write data directly without triggering any computation of related RDDs.

B. Debugging framework for Spark

A technique used to improve quality of the software is debugging. Developers usually debug to observe certain set of variables they are interested. However, in the Data-intensive Scalable Computing (DISC), the debugging process is difficult as data are computed distributedly on a cluster.

BigDebug [7] is a tool designed to helps Spark's developers deal with debugging a Big Data program. There is a downside that the tool requires user's interaction during the debugging process. Those interactions make the debugging more difficult than those of normal programs because the Big Data programs are distributed by nature. Moreover, a BigDebug program cannot tackle the problem when the RDD being debug requires changes. The whole debugging process needs to start over in that case. In case of the developer changing codes on-the-fly, the RDD will become in-consistent as some partitions of the RDD has been processed by the old version of codes, while other partitions will be processed by the new codes. BigDebug support Spark up to 1.2.1 as the time writing.

C. Checkpoint implementation for Spark

Researchers have been employed the checkpoint of Spark in many ways to improve its efficiency, as follows.

Flint [26] was created atop the original checkpoint technique of Spark. It aims at applying checkpoint and store their data on transient instances to reduce the VM usage cost. A transient instance in a kind of low-cost computing unit, which can be recalled anytime by its cloud provider. Flint solves this problem by writing an RDD's partitions to an HDFS, which is operated on on-demand instances. We found that this implementation lacks a mechanism to prevent re-calculation when JVM is terminated. In addition,

their checkpoint will be saved automatically so developers need to prepare a huge amount of space in order to prevent the full of storage, which can lead to the failure of the whole system.

TR-Spark [27] implements the similar approach as Flint. The difference is that TR-Spark allows fined-granularity checkpoints at task-level. By leveraging this level of checkpoints, the storage usage cloud be reduced in comparison to checkpoint the whole RDD. However, TR-Spark makes it difficult to use as developers need to collect the information of VM failure to let it know the failure probability. TR-Spark does not deal with changes of the Driver program.

Automatic Spark Checkpointing (ASC) [25] was designed to help analyze the trade-off between RDD checkpointing and its restore. ASC performs this computation by estimating them from an RDD lineage. Nevertheless, this technique does not support checkpoint across JVM termination. It also lacks the ability to recognize the similarity or identity of an RDD.

Spark-flow [24] aims to mitigate the effect of JVM termination for checkpoint restoration. It makes use of Distributed Collection (DC), a library similar to the Dataset API. DC is able to analyze an RDD at the bytecode level with ASM. It can identify the location of checkpoint calls, inside an anonymous function. It also uses the MD5 hash function to help detect changes at the bytecode level. However, DC has some downside as the following. First, when calling checkpoint on a DC, the data is re-read again after checkpointing. Second, when restoring from checkpoint, the action *count* will be triggered, so the re-computation kicks in. Finally, computation is mainly done on the Driver machine, so the mechanism is actually not distributed. This often causes Out-of-Memory exception inside the Driver program and it stops working.

```
1  val data = sc.parallelize(Array(1,2,3,4,5))
2  val distData = data.map(x => (x,1))
3  distData.dtCheckpoint()
4  distData.count()
5  distData.collect()
```

Fig. 1. Example of a dtCheckpoint call on an RDD

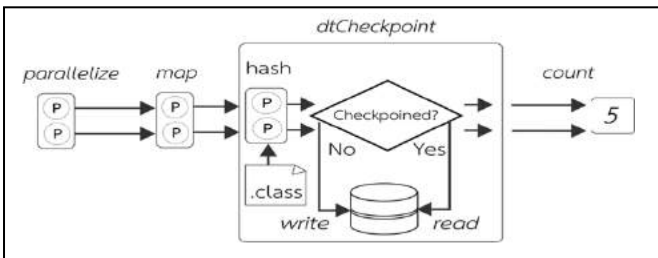

Fig. 2. The dtCheckpointing mechanism inside DTC

III. Design and Implementation

Spark stores the RDD transformations in the form of a lineage graph a.k.a. the logical execution plan. When an action is triggered for a certain RDD, its job will be submitted to the DAG Scheduler to transform the RDD's lineage into a directed acyclic graph, whose a *vertex* is an RDD partition and *edge* is a transformation. After that the staging process will be kicked in. This staging process will be started from the final action going backwards to the beginning of the RDD. However, in the real execution, the process will be performed from the beginning of the RDD forwardly to the final action. After the staging, the system obtains a set of Stages and Tasks.

A checkpoint of an RDD however must be done before the first action is performed. From the source code in the Fig. 1, when a program starts to process an array of integer 1 to 5, the array will be passed as a parameter of method *parallelize* of class SparkContext. This result in a ParallelCollectionRDD stored in variable *data.* At line 2, each element from the *data* RDD is mapped with 1 using the *map* method as a key/value pair. The result is a MapPartitionsRDD stored in variable *distData*. At line 3, method *dtCheckpoint* is invoked. Please note that the original Spark and DTC both use the lazy evaluation mechanism, this means that the checkpoint method only marks at a certain point over the DAG, where checkpoints will happen there. At line 4, command *distData.count()* is the first action. When this first action is triggered, the checkpoint is not yet created. The computation then is started from the beginning of the RDD to the mark point. After that, the checkpoint is stored at the first upper directory level as a hash value generated by the mechanism of DTC. At the line no 5, method *distData.collect()* is invoked as the second action. The system will then check backwards from the action to the beginning of the RDD. This time the system will find a checkpoint already existed because there is a directory whose name matches with the hash. When the DAG Scheduler starts to transform the lineage, it uses the data directly from the checkpoint without re-computation. Please also note that action *count()* and *collect()* belong to the different jobs. The result computed by *count()* will not be included as an input for *collect(),* despite their order of execution.

In Scala, it allows us to implement a new feature for a class by creating an *Implicit Class* then mixes it in to the existing classes, like RDD or Dataset. The DTC mechanisms proposed in this paper are implemented using that technique. With DTC as an Implicit Class, developers could still use all existing properties and behavior of an RDD, while having an additional method from DTC. Developers are also able to upgrade the Spark framework to the newer versions without rewriting this mechanism. DTC is more suitable for testing than Spark-flow, which has many abstraction layers. These abstraction makes it difficult to enhance capability of Spark-flow.

A. DtCheckpointing

This mechanism works when the method *dtCheckpoint* of an RDD or a DataSet is called. This call marks an RDD and also starts the Hashing RDD mechanism to obtain a directory path from hash transformation. If there is no directory matched the hash value, it means that the system never created that checkpoint. After the creation of the directory content of the RDD will be stored inside of it. But if the directory exists, the system will read the content as the data of the RDD. In Fig. 2, when an RDD is created using the *parallelize* method and is transformed with *map* followed by an invocation of *dtCheckpoint*. The sub-system

DtCheckpointing kicks in to mark points in the RDD for later storing when action *count* is called.

We usually perform the test on a Spark Cluster with SBT, which is an interactive build tool to help develop software with Java or Scala. SBT allows us to write a build file using Scala-based Domain Specific Language. It manages a program dependency with Apache Ivy. With DTC, we modify test commands of the SBT namely *test*, *test-only*, and *test-quick* to support not only the local execution but also in the real working cluster. We solve the problem of *ClassNotFoundException* and *NoClassDefFoundError* by making a fat jar via custom SBT task. So, we introduce *testOnCluster* for testing every testcase, *testOnlyOnCluster* to test a specific testcase, and *testQuickOnCluster* to test a certain testcase which may be failed from last time, or never tested or need re-computation. Our modification to SBT allows the new mode of testing on the real cluster.

B. Hashing an RDD

Hash function is a one-way function which can be used to check data modification. Eve one bit of data is changed this function notices that modification. In this paper, we will compare MD5, SHA-1 and SHA-256 because these algorithms have various speed of hash and resource usage.

This technique of the DTC framework is able to track the change of an RDD because the generated transformations. So we can use this mechanism to detect modification of any transformation back to the original RDD. When an action is triggered, the DTC framework detects all RDD dependencies and prepares a clean bytecode available by the CleanF property of the RDD, following by preparing other Java bytecode's files which related to the dependencies. In preparation stage, DTC uses ASM, a tool to manage a Java bytecode [17], which Scala internally uses it for the compilation mechanism. With a ASM, the DTC's hashing an RDD mechanism can access Java class file at runtime and de-serialize them for reverse engineering propose. DTC needs to remove some brittle information such as LINENUMBER or serialVersionUID from a class file. With this information filtered out, we can detect changes of an RDD or DataSet even when the line numbers have been changed.

The result of class file analysis in preparation stage, after unnecessary dependencies was eliminated, these dependencies will compute hash number and input data, which the origin of an RDD will compute hash number also. The computation is distributed computing with Spark's accumulator in the first level hash number computation will compute hash number of input data for every partition, and then collect and reorder result because unpredictable computation time. After that, the DTC will compute hash number of sorted hash number again. Fig. 3, illustrates the steps of hashing mechanism please note that the computation of input data is an option that can specify with *dtCheckpoint(true)*.

```
SET hash_array = empty array of string
IF (HASH_INPUT_DATA = true) THEN
  READ each data partition from (RDD or DataSet)
  COMPUTE hash of each data partition
  APPEND hashes to hash_array
ENDIF
```

Fig. 3. Pseudo codes of the mechanism of Hashing an RDD

IV. Experiments

A. Cluster configuration

The experiments presented in this paper have been conducted on a Spark cluster consisted of 10 nodes. Each node is an Intel Core i5-4570 Quad-core with 4 GB of RAM. The drive node is an Intel Xeon E5-2650V3 Deca-core with 8GB of RAM. We use Apache Spark 2.0 for the experiments along with Ceph as the distributed file system over these 10 nodes. The Ceph storage is 10 TB. The system architecture is illustrated in Fig. 4.

TABLE I
COMPUTATION PROGRAMS AND INPUT DATA OF EXPERIMENTAL

Program	Input dataset
Wordcount	31 GB of Wikipedia
Triangle Counting	875,713 vertices and 5,105,039 edges
PageRank	875,713 vertices and 5,105,039 edges
Pi Estimation	10^9 times

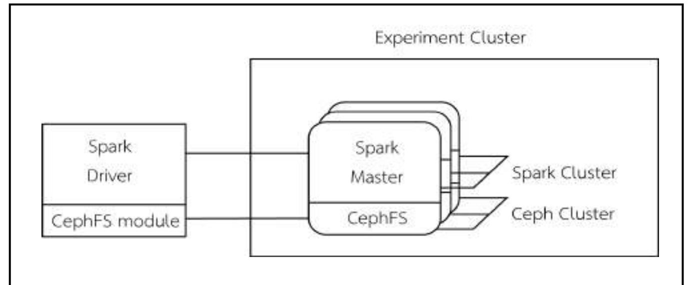

Fig. 4. The cluster architecture used by the experiments

B. Methodology

For the experiments, we use a MapReduce program Wordcount on 31 GB data dump of Wikipedia, Triangle Counting with Google Web Graph [28], PageRank with Google Web Graph and the last one is Pi Estimation with one billion times. Each program with its input dataset is shown in Table I. The Wordcount Program splits sentences into array of words and counts them using both RDD and Dataset (or DC in case of Spark-flow) with different checkpoint mechanisms. We tested each checkpoint mechanism 10 times continuously and measured both in space and time perspectives. Moreover, we tested 5 additional with JVM termination. Then we started the JVM again to test the recovery process of checkpoints.

Table II shows the comparison of checkpoint mechanism properties. If we do not use checkpoint, the system does not have the fault tolerance property. If we use the original Spark, it is not suitable for testing because its checkpoint mechanism does not work well in the test environment. In case of Spark-flow it does not work on the cluster environment out-of-the-box. DTC, on the other hand, is designed to address these problems in the testing

TABLE II
FEATURE COMPARISON BETWEEN CONFIGURATIONS

Method	Failure tolerance	More abstraction layer	Prevent re-calculation from beginning	Suitable for Testing	Cluster
No-Checkpoint	No	No	No	No	Yes
Spark Original	Yes	No	Yes	Not Suitable	Yes
Spark-flow	Yes	Yes	Yes	Yes	No
DTC	Yes	No	Yes	Yes	Yes

TABLE III
THE COMBINATION OF ALL EXPERIMENTAL CONFIGURATIONS

Configuration	Type			Checkpoint Data Format				Hash Algorithm		
	RDD	DataSet	DC	Java	Kryo	Avro	Parquet	MD5	SHA1	SHA256
No-checkpoint	√	√	-	-	-	-	-	-	-	-
Spark Original	√	√	-	√	-	-	-	-	-	-
Spark-flow	-	-	√	-	-	-	√	√	-	-
DTC	√	√	-	√	√	√	√	√	√	√

environment. So, DTC provides the better environment to support unit testing.

Table II shows a brief differentiation of comparison method that we will experiment. That meant, if we have no checkpoint it will lack failure tolerance, the Spark original checkpoint insufficient to testing. The Spark-flow push developer in more abstraction layer by create a higher level of a DataSet and it not work on cluster naturally. In Table III, we show the combination of all experimental configurations. Accordingly, the DTC introduce to rectify that plain.

We compared with MapReduce Wordcount algorithms on Wikipedia 31 GB with separating each word from each other with white space. And then, we filtered only word occurred more than 10 million times, after that asserted with the most word occurred. We consecutively repeated these steps 10 cases and performed testing on 5 cases then stopped the JVM. After that we re-run these 5 cases again on both RDD and DataSet.

Next, we compared with Triangle Counting Program which gathers the number of vertices whose has two adjacent vertices with an edge between them. And then perform PageRank Program to ranks members onto the graph. Input of these programs came from Google Web Graph. with 875,713 vertices and 5,105,039 edges, testing on 5 cases then stop the JVM, after that re-run these 5 cases again on RDD.

Finally, we compared the Pi Estimation program by using Monte Carlo algorithm shows in (1) [29].

$$\mathbb{P}(drop\ within\ circle) = \frac{Area\ of\ the\ unit\ circle}{Area\ of\ the\ square} = \frac{\iint_{\{x^2+y^2\leq 1\}} 1\,dx\,dy}{\iint_{\{-1\leq x,y\leq 1\}} 1\,dx\,dy} = \frac{\pi}{4} \quad (1)$$

The algorithm randomly generated two values which represent to coordinate x and y of unit circle (so both x and y are between -1 to 1). After that, trying to addition between square magnitude of x and square magnitude of y and if that result less than or equal to 1 will be count as fall in the unit circle. That number will use to represent $\pi/4$, so that we can multiply by 4 to roughly results Pi number. We tested 5 cases then stop the JVM, after that we re-run these 5 cases again on RDD.

C. *Experimental results (consecutively 10 cases)*

From the experiments, we start discussing in the case of no hashing input data, denoted *not-hashinput* by running consecutively 10 cases. In this case the input will not be verified by hashing functions before the program starts. We assume that development and during the tests. The experimental results are show in Fig. 5. At the first run, DTC and the `original-checkpoint` mechanism are all slow with insignificant difference. The `DTC-Java-SHA1` is slowest. It uses 636 seconds slightly different from `original-checkpoint`. The `no-checkpoint` configuration does not have this startup overhead, so it run at 136 seconds on average. For the first run, All DTC and the `original-checkpoint` are 4.7 times or slower than the `no-checkpoint` mechanism. However, all DTC configurations are significantly faster in the subsequence runs.

TABLE IV
CHECKPOINT'S STORAGE USAGE OF AN RDD

Storage usage	Size	Unit
No-checkpoint	0	MB
Spark original checkpoint	9.870	MB
DTC-Java-with-hash	0.987	MB
DTC-Java-without-hash	0.987	MB
DTC-Kryo-with-hash	*0.501*	*MB*
DTC-Kryo-without-hash	*0.501*	*MB*

TABLE V
CHECKPOINT'S STORAGE USAGE OF DATASET

Storage usage	Size	Unit
No-checkpoint	0	MB
Spark original checkpoint	9.860	MB
DTC-Avro-with-hash	*0.987*	*MB*
DTC-Avro-without-hash	*0.987*	*MB*
DTC-Parquet-with-hash	0.993	MB
DTC-Parquet-without-hash	0.993	MB
Spark-flow	9.930	MB

Fig. 6 shows the comparison between cases of applying hash functions over input data to allow the system to detect

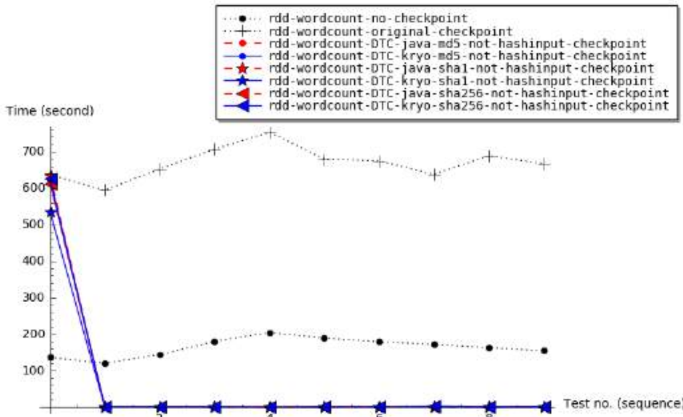

Fig. 5. Comparison of checkpoint time of RDDs without hashing inputs using the Wordcount program. (10 cases consecutively)

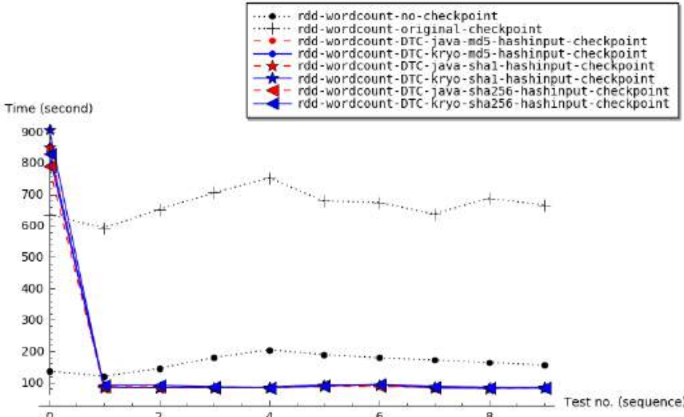

Fig. 6. Comparison of checkpoint time of RDDs with hashing inputs using the Wordcount program. (10 cases consecutively)

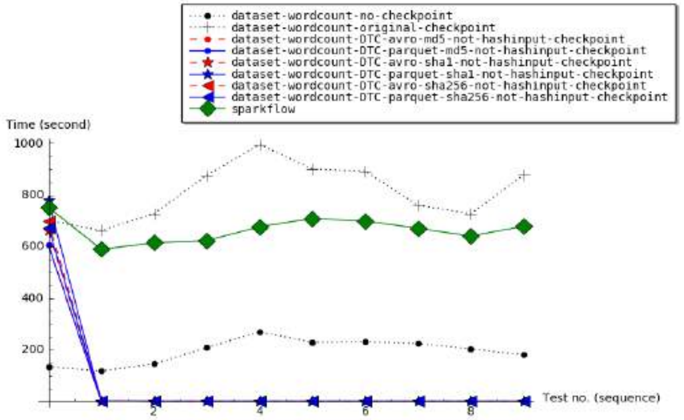

Fig. 7. Comparison of checkpoint time of DataSet,including Spark-flow without hashing inputs using the Wordcount program (10 cases consecutively).

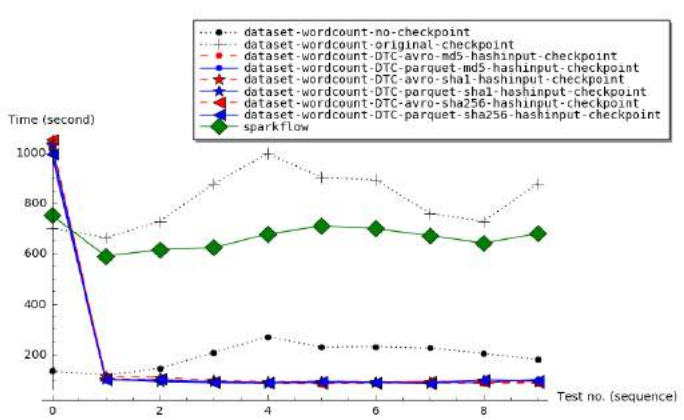

Fig. 8. Comparison of checkpoint time of DataSet,including Spark-flow with hashing inputs using the Wordcount program (10 cases consecutively).

changes of the input. It shows that DTC mechanisms are slower than `no-checkpoint` and `original-checkpoint` only in the first run. In the subsequence runs, DTC mechanisms make the test s faster than those run by `no-checkpoint` and `original-checkpoint`. We found that `DTC-Kryo-SHA1` is slowest in the first run. It uses 908 seconds on average, while `no-checkpoint` uses 136 seconds and `original-checkpoint` use 636 seconds. In the subsequence runs, DTC mechanism uses around 85 seconds on average. It is significantly faster that both `no-checkpoint` and `original-checkpoint`, which is 60%

In the first run with hash input, the fastest DTC mechanism is `DTC-Java-SHA256` it is 480% slower than no-checkpoint and 24% slower than `original-checkpoint`. In the subsequence runs, this mechanism is 40% faster than `no-checkpoint` and 590% faster than `original-checkpoint`. Other cases are in similar trends.

In case of DataSet, we found the similar trends as the case of RDD. During the first run DTC mechanisms are slowest, and significantly faster in subsequence runs. Fig. 7 and Fig. 8 show the comparison between checkpoint mechanisms for the DataSet without hashing input and with hashing input, respectively. We also include `Spark-flow` in these experiments. We found that `Spark-flow` uses 752 seconds at the first run, while `DTC-Parquet-MD5` uses 606 seconds, so DTC is 24% faster than `Spark-flow`. In case of hash input data, DTC is 40% slower than `Spark-flow` for the first run. However, in the subsequence runs, DTC dramatically reduces time spending, according aforementioned trends.

The mechanism of checkpoint usually requires use of storage. The storage usage comparison is then presented in Table IV. According to the table, DTC with Java serializer uses the storage only one-tenth of those used by the original Spark checkpoint. In case of DTC with Kryo, it uses storage only 5% of the `original-checkpoint`.

This storage usages are similar for DataSet. According to Table IV, DTC with Avro format uses only 10% of the original storage. In case of DTC with Parquet format, it uses only 11% of the original storage. Comparison of these results with `Spark-flow`, we are roughly at the same ration.

DTC is designed to allow re-usability of RDDs and DataSets. It can traverse and detect change of the dependency of each RDD or a DataSet. From the experiments, we have found that DTC has a larger overhead than the mechanism of the Original Spark only when a testcases are in first run. When the testcases are in the later runs, DTC makes them 5-6 times faster than running by the Original Spark and `Spark-flow`. Moreover, DTC uses disk space 8-9 times less than both implementations as shown in Table IV and Table V.

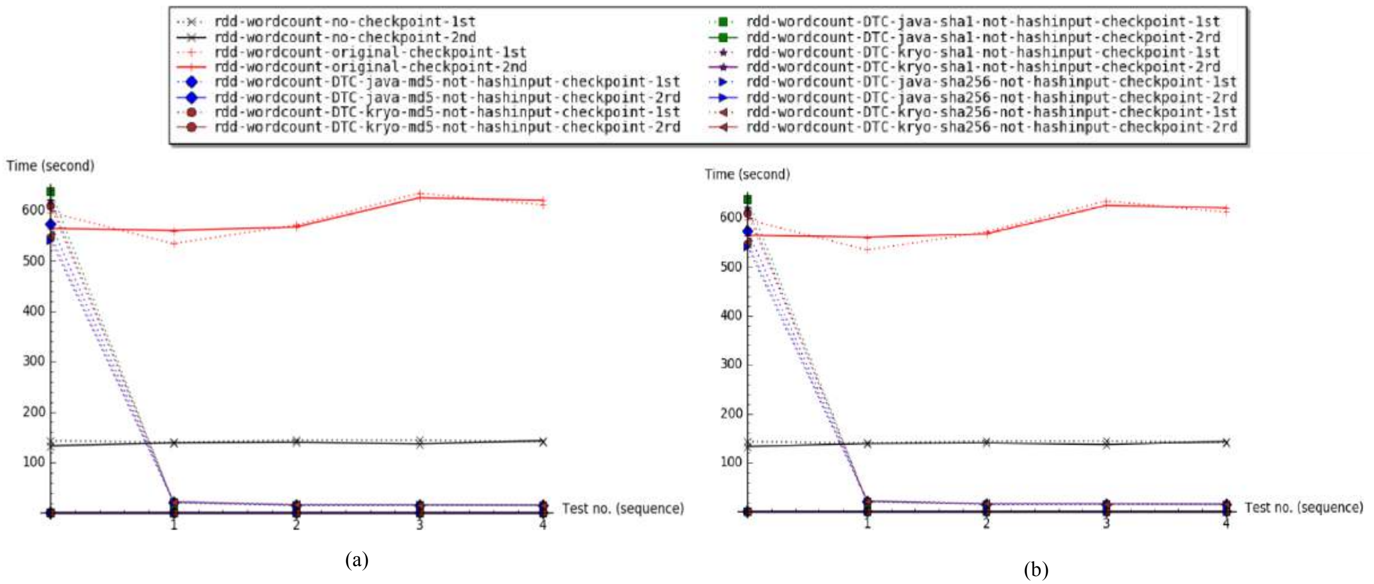

Fig. 9. Comparison of checkpoint time of RDDs using the Wordcount program (5 cases with JVM termination) while (a) without hashing inputs and (b) with hashing inputs.

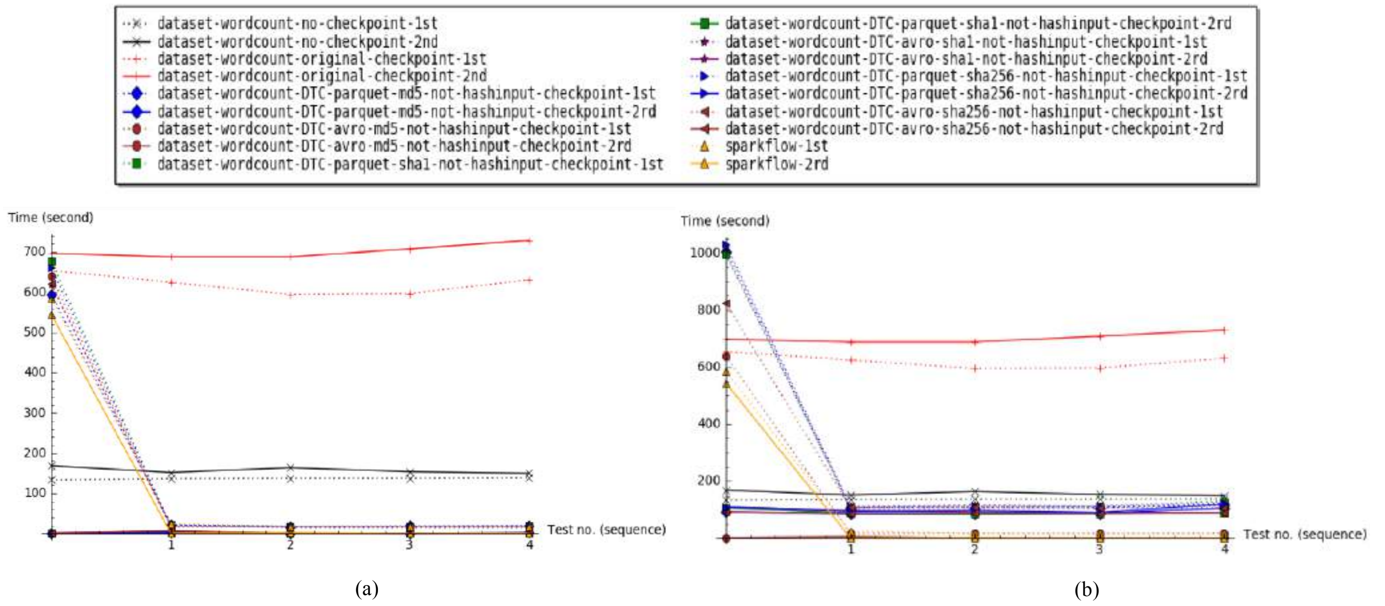

Fig. 10. Comparison of checkpoint time of DataSet using the Wordcount program (5 cases with JVM termination) while (a) without hashing inputs and (b) with hashing inputs.

D. Experimental results (5 cases with JVM termination)

In this section, we discuss the experimental results in case of running 5 cases consecutively, then stopping the JVM, after that the experimental cases were re-run again. Its behavior on different frameworks were observed.

Firstly, we discuss the result of the Wordcount program on RDD. We found that `DTC-Java-SHA256` used 542 seconds at the first run in case of running if before stopping JVM, so DTC is 9% faster than `original-checkpoint` which uses 596 seconds. After stopping JVM or closing the program then re-running the test cases, DTC with all settings used only few seconds to recover checkpoint, while other frameworks used hundreds of second, as showed in Fig 9. In Fig 9, the dashed line is the first running before JVM terminating and the solid line is the second running after restarting the JVM.

In the case of DataSet shown you in Fig 10, the dashed line presents the first run of 5 cases. We found that the `original-checkpoint` used 654 seconds, while `Spark-flow` used 585 seconds. So, `Spark-flow` is 11% faster than the original one. But DTC with the `DTC-Parquet-MD5` configuration, it used 595 seconds, 9% faster than `original-checkpoint`. However, in the second run of 5 cases after restarting the JVM, as the solid line, the results show that the `original-checkpoint` used 697 seconds and `Spark-flow` used 545 seconds, while DTC with any configuration used just few seconds.

Fig. 11 shows the results comparing between frameworks using Triange Counting Program, In the case of not applying hashing to the input data, we showed that in Fig 11 (a), `no-checkpoint`, `original-checkpoint` and DTC used almost the same amount of time for the first runs.

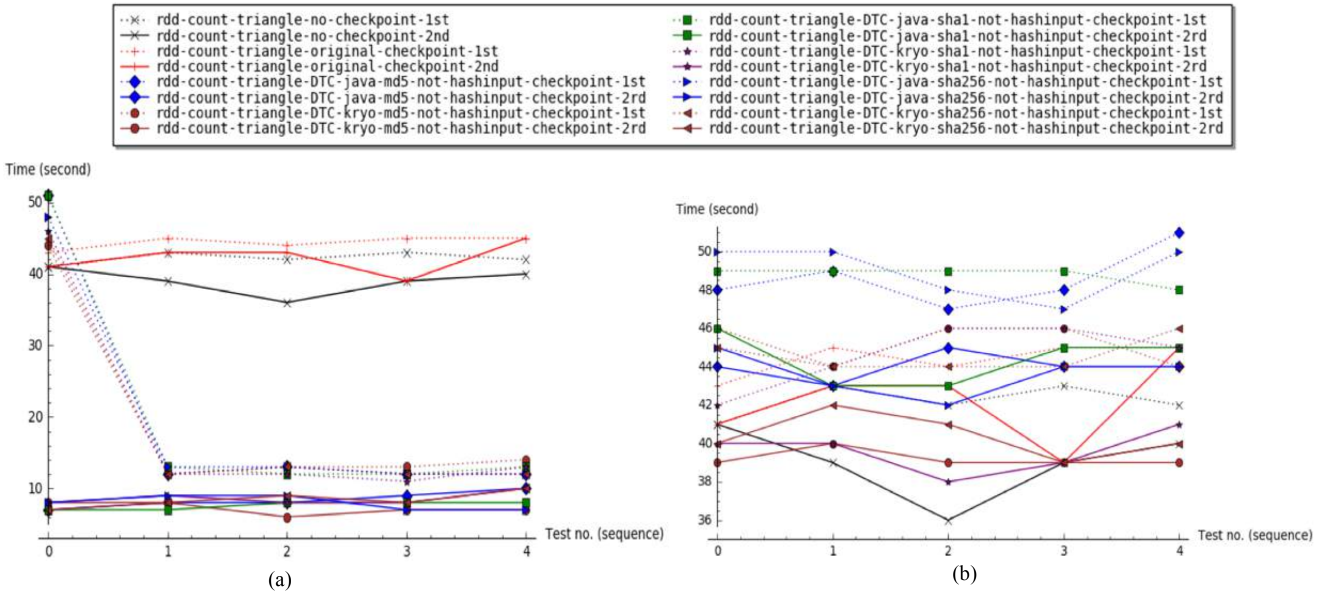

Fig. 11. Comparison of checkpoint time of RDDs using the Triangle Counting program (5 cases with JVM termination) while (a) without hashing inputs and (b) with hashing inputs.

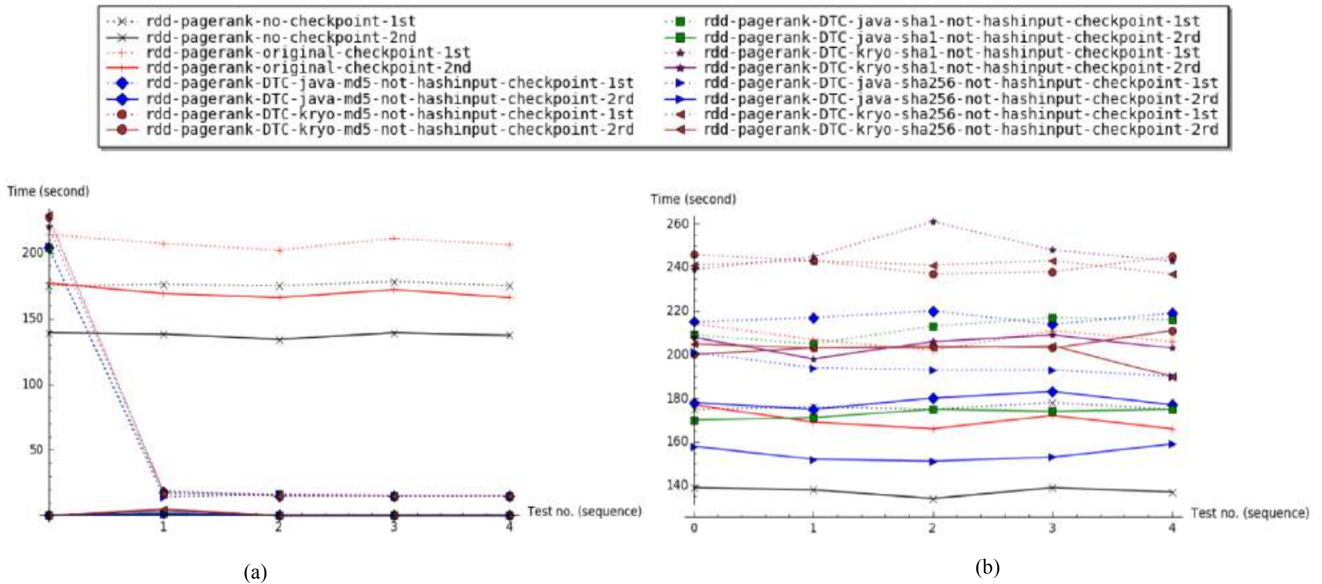

Fig. 12. Comparison of checkpoint time of RDDs using PageRank Program (5 cases with JVM termination) while (a) without hashing inputs and (b) with hashing inputs.

For the second runs after restarting the JVM, we found the same trend as we were discussing earlier. DTC with all configurations could reduce time for testing to just a few seconds. Due to inputs were in the form of graph (vertices and edges) as shown in Fig 11 (b), the underlying mechanism of the Spark Framework tries to perform operations efficiently by casting the partition of the input to class *ShippableVertexPartition*. In the research work reported in this paper, DTC does not import to support to read this kind of data type. Fig 11 (b) shows that DTC with all configurations could not help reduce time much. All frameworks use the same amount of time processing the data.

In Fig 12 shows the experimental results obtained from running the PageRank program. PageRank is a program that processes graphs. It used the same set of inputs as the previous experimental, Triangle Counting. In Fig 12 (a), it shows the results in the case of not applying hashing to the input data. We found that in the first testcase of the first run, the results of DTC with Java serialization, with either MD5 or SHA1 as the hash function, used 204 seconds, while the `original-checkpoint` used 214 seconds. In this comparison, DTC could speed up by 4%. For the rest of testcases, times spent by DTC is cut down to just a few seconds. In Fig 12 (b), we also found the same problem as of the Triangle Counting program. This was the result of hashing input.

Finally, we discuss the results of the Pi Estimation program. In Fig. 13, we showed tenor of comparing frameworks. For the first testcase of the first run, we found

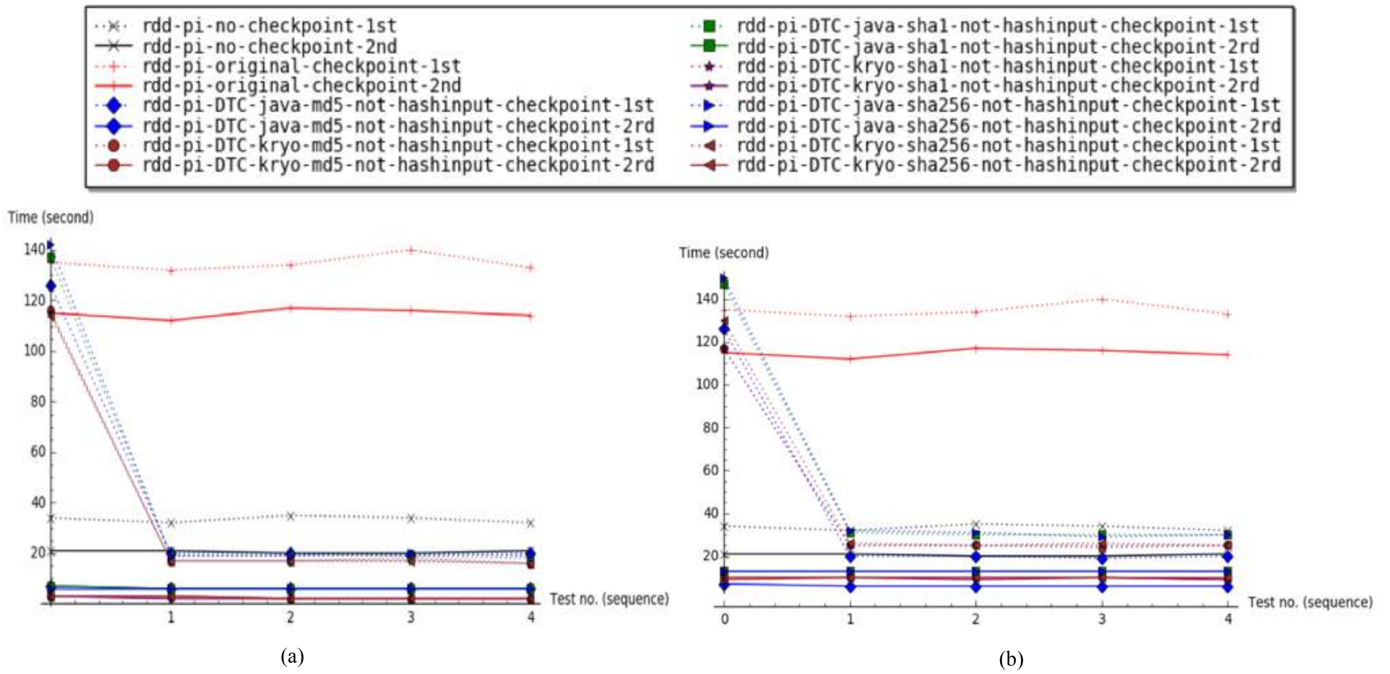

Fig. 13. Comparison of checkpoint time of RDDs using Pi Estimation Program (5 cases with JVM termination) while (a) without hashing inputs and (b) with hashing inputs.

that without hashing inputs, the `DTC-Kryo-SHA256` used 114 seconds, while the `original-checkpoint` used 135 seconds as shown in Fig 13 (a) DTC was 18% faster in this case. In the consequent testcases, DTC could cut the running time significantly.

In case of hashing inputs, we found the same trend as shown in Fig 13 (b) as the previous results. DTC used processing time almost the same as `original-checkpoint` at the first testcase then dramatically speed up by using only a few seconds for testing each testcase. Moreover, the DTC framework can be detected in case of random values, so that spark developers can reproduce the input which causes software is issues.

V. CONCLUSIONS AND FUTURE WORK

The experimental results have obviously shown that DTC is suitable for improving productivity for unit testing in Big Data applications in terms of time consumption and storage usage. We can perform testing for Big Data either on a local or a cluster. DTC could trace change in testcases with random values. Unfortunately, we found that DTC could work well in case of graph algorithms such as Triangle Counting or PageRank due to spark framework cast partition of an input to *ShippableVertexPartition*. So that one of limitation the DTC is input datatype. We are researching in potential mechanisms which can be used for increasing speed of testing and reducing storage usages such as *cache* and *persist*. The JVM configurations are ones of tuning parameter we are focusing. These subjects are being studied.

REFERENCES

[1] W. Fan and A. Bifet, “Mining big data: current status, and forecast to the future,” in *ACM SIGKDD Explorations Newsletter*, 2012, vol. 14, pp. 1–5.

[2] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Communications of the ACM - 50th anniversary issue: 1958 - 2008*, vol. 51, no. 1, pp. 107–113, 2008.

[3] B. Mark and B. Rajkumar, “Cluster Computing: The Commodity Supercomputer,” in *Software-Practice and Experience*, 1999, vol. 29(6), pp. 551–576.

[4] “Welcome to Apache™ Hadoop®!” [Online]. Available: https://hadoop.apache.org/. [Accessed: 06-May-2017].

[5] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, “Map-reduce-merge: Simplified Relational Data Processing on Large Clusters,” in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2007, pp. 1029–1040.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” in *Proceedings of the second USENIX Conference on Hot Topics in Cloud Computing*, 2010, pp. 10–10.

[7] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim, “BigDebug: Interactive Debugger for Big Data Analytics in Apache Spark,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2016, pp. 1033–1

[8] A. Spillner, T. Linz, and H. Schaefer, Software testing foundations: a study guide for the certified tester exam. Rocky Nook, Inc., 2014.

[9] “ScalaTest.” [Online]. Available: http://www.scalatest.org/. [Accessed: 06-May-2017].

[10] “JUnit 5.” [Online]. Available: http://junit.org/junit5/. [Accessed: 06-May-2017].

[11] “holdenk/spark-testing-base,” *GitHub*. [Online]. Available: https://github.com/holdenk/spark-testing-base. [Accessed: 06-May-2017].

[12] “[SPARK-2243] Support multiple SparkContexts in the same JVM - ASF JIRA.” [Online]. Available: https://issues.apache.org/jira/browse/SPARK-2243. [Accessed: 06-May-2017].

[13] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[14] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*. O’Reilly Media, Inc., 2015.

[15] “JerryLead/SparkInternals,” *GitHub*. [Online]. Available: https://github.com/JerryLead/SparkInternals. [Accessed: 07-May-2017].

[16] H. Karau and R. Warren, High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media, Incorporated, 2017.
[17] E. Kuleshov, "Using the ASM framework to implement common Java bytecode transformation patterns," *Aspect-Oriented Software Development*, 2007.
[18] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.
[19] "A Universally Unique IDentifier (UUID) URN Namespace," *A Universally Unique IDentifier (UUID) URN Namespace*. [Online]. Available: https://www.ietf.org/rfc/rfc4122.txt. [Accessed: 07-May-2017].
[20] S. Saxena, *Getting Started with SBT for Scala*. Packt Publishing, 2013.
[21] "Home | Apache Ivy ™." [Online]. Available: https://ant.apache.org/ivy/. [Accessed: 07-May-2017].
[22] "sbt/sbt-assembly," *GitHub*. [Online]. Available: https://github.com/sbt/sbt-assembly. [Accessed: 07-May-2017].
[23] "The Daily Build - write simple SBT task." [Online]. Available: http://blog.bstpierre.org/writing-simple-sbt-task. [Accessed: 07-May-2017].
[24] "bloomberg/spark-flow," *GitHub*. [Online]. Available: https://github.com/bloomberg/spark-flow. [Accessed: 08-May-2017].
[25] W. Zhu, H. Chen, and F. Hu, "ASC: Improving spark driver performance with automatic spark checkpoint," in *Advanced Communication Technology (ICACT), 2016 18th International Conference on*, 2016, pp. 607–611.
[26] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, "Flint: batch-interactive data-intensive processing on transient servers," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, p. 6.
[27] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "TR-Spark: Transient Computing for Big Data Analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, New York, NY, USA, 2016, pp. 484–496.
[28] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 2009
[29] A. M. Johansen, L. Evers, "Monte Carlo Methods", University of Bristol, Department of Mathematics

Bhuridech Sudsee received B.Eng. in Computer Engineering from Suranaree University of Technology and B.Sc. in Information Technology from Sukhothai Thammathirat Open University, both in Thailand. Currently, he is studying a Master degree in Computer Engineering. His fields of research interests are high-performance computing, distributed computing, data storage, Big Data processing and MapReduce frameworks.

Chanwit Kaewkasi received his PhD in Computer Science from the University of Manchester, United Kingdom in 2010. He is currently an Assistant Professor at School of Computer Engineering, Suranaree University of Technology, Thailand. Dr. Kaewkasi is actively researching in the areas of Low-Power Clusters, Cloud Computing, Big Data and Software Container Technologies.