A Reliability Benchmark for Big Data Systems on JointCloud

Yingying Zheng[12], Lijie Xu[1], Wei Wang[12], Wei Zhou[3], Ying Ding[4]
[1]Institute of Software, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences, [3]KSYUN
[4]Institute of space optoelectronic technology, Changchun University of Science and Technology

***Abstract*—JointCloud provides a large-scale, flexible, and elastic computing resource platform. Big data systems such as MapReduce and Spark are widely deployed on this platform for big data processing. How to choose a cloud platform in accordance with the need of customers is a problem. Current performance benchmarking suites can choose suitable cloud platforms for customers. However, they do not consider the reliability of applications running atop big data systems. These systems have high scalability, but the applications running atop them often generate runtime errors, such as out of memory errors, IO exceptions, and task timeouts. For users, they want to know whether the developed applications have potential application faults. For system designers and managers, they want to know whether the deployed/updated systems have potential system faults. In addition, current benchmarks for big data system are also only designed for performance testing. To fill this gap, we propose a reliability benchmark, which contains representative applications, an abnormal data generator, and a configuration combination generator. Different from performance benchmarks, this benchmark (1) generates abnormal test data according to the application characteristics, and (2) reduces the configuration combination space based on configuration features. Currently, we implemented this benchmark on Spark system. In our preliminary test, we found three types of errors (i.e., out of memory error, timeout and wrong results) in five SQL, Machine Learning, and Graph applications.**

I. Introduction

JointCloud, as a new generation of cloud computing platform, provides joint cloud services and complex calculations. Its pay-as-you-go computing model provides a scalable platform for big data systems. Due to high scalability, big data systems such as MapReduce [1], Spark [2], and Flink [3] are now widely deployed on cloud platforms for big data processing. Because of the large number of cloud providers, there is a problem that how to choose a cloud platform in accordance with the need of customers. Previous studies [4, 5] provide performance benchmarking suites to choose a suitable cloud for customer. There is not a benchmark for choosing cloud for reliability of applications. However, the applications deployed on big data systems often suffer from runtime errors, such as out of memory errors [6], IO exceptions [7], and task timeouts [8]. These errors can directly lead to application failures and cannot be tolerated by current fault-tolerant mechanisms.

A big data application can be denoted as (*input data, configurations, user code*). The input data is usually stored as data blocks on distributed file system. Configurations include system-specific configurations (e.g., *input block size*, *partition number*) and application-specific configurations (e.g., *K-means* application's cluster *k*). User code refers to the user-defined functions, such as *map*(), *reduce*(), and *join*(), which process the input or intermediate data.

Previous empirical studies [6, 7, 9, 10] have summarized the root causes of applications' runtime errors: (1) *application faults*, including improper configurations, abnormal data, and user code defects. Improper configurations refer to large *input data block size*, small *partition number*, unbalanced *partition function*, etc. Abnormal data refers to exceptional input/intermediate/output data, such as skewed data and high dimension data. User code defects contain memory leak, high time/space complexity, etc. (2) *system faults*, including hardware faults and software faults. Hardware faults refer to CPU, memory, network and disk failures. Software faults include logic-specific bugs, data races, etc.

For users, they want to know whether the developed applications have potential application faults. For system designers and managers, they want to know whether the deployed/updated systems have potential system faults. Testing is a promising approach, but current benchmarks [11, 12, 13, 17] for big data systems are designed for performance testing. Since these benchmarks use normal input data and fixed configurations, they cannot be directly used to detect potential faults.

In this paper, we propose a reliability benchmark for big data systems. To detect the potential application/system faults, this benchmark generates abnormal input data, and combines both system-specific and application-specific configurations to test the applications. Different from performance benchmarks, this benchmark (1) generates abnormal input data according to the application characteristics, and (2) reduces the configuration combination space based on configuration features.

We implemented this benchmark on Spark system. This benchmark currently contains 10 representative applications, an abnormal data generator, a configuration combination generator, and a test report generator. In our preliminary test, we found five errors: (1) out of memory error in a SQL operation where a small table inner joins a large table with skewed data; (2) wrong results in a SQL operation where a table participates in multiple join operations but not renamed; (3) out of memory error in *RandomForest* with high dimension data; (4) out of memory and timeout errors in *LogisticRegression* with high dimension, and abnormal distribution data; (5) out of memory error in PageRank application with large and sparse data.

In summary, our main contributions are as follows:

- A reliability benchmark is designed for big data systems, which generates abnormal test data according to the application's characteristics.
- A greedy configuration combination method is designed to reduce the configuration combination space through analyzing the configuration independence and correlation.
- We found three types of errors (i.e., out of memory error, timeout, and wrong results) in five applications.

II. Benchmark Design and Implementation

The reliability benchmark mainly contains four parts, as shown in Fig. 1:

1) Representative applications selection: It selects widely-used SQL, Machine Learing, and Graph applications.

2) Abnormal input data generation: It summarizes applications' computational characteristics and generate abnormal input data according to the characteristics.

3) Configuration combination test: It combines system-/application-specific configurations and reduce the combination space to test the applications.

4) Test report generation: It analyzes testing results and reports the errors/faults .

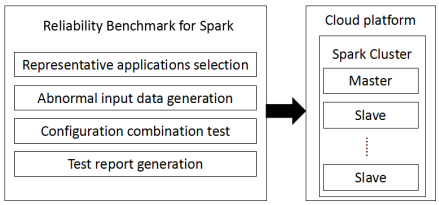

Fig. 1. The modules of the reliability benchmark

A. Representative applications selection

We select the representative applications based on the following criteria: (1) the application represents a basic data operation or a widely-used algorithm; (2) the application has a standard or well-tested implementation. At present, we selected 10 applications from previous benchmark [11, 12, 13, 17] or Spark's libraries such as Spark SQL, MLlib and GraphX library in Spark. Table I illustrates the type and computational characteristics of each selected applications.

1) Scan: A data filter query like SELECT * FROM *TableA* WHERE *columnValue* > *x*.

2) Aggregate: A data aggregation query like SELECT *columnA*, sum (*columnB*) AS *total* FROM *TableA* GROUP BY *columnA* ORDER BY *total*.

3) Join: A data join query like SELECT * FROM *TableA* INNER JOIN *TableB* ON *A.columnA* = *B.columnA*.

4) Mix: A query contains data filter, aggregation and join.

5) Logistic Regression: An iterative classification algorithm used to predict continuous or classified data. This algorithm uses a stochastic gradient-descent or L-BFGS algorithm to train the classification model. The model parameters are calculated, updated, and propagated in each iteration.

6) K-means: an unsupervised clustering algorithm which iteratively computes the K centers.

7) Decision Tree: a supervised classification algorithm which builds a tree to classify the data.

8) Random Forest: Different from Decision Tree, it builds multiple trees and combines them to classify the data. It uses random sampling to train each tree.

9) PageRank: An algorithm used by Google Search to rank websites in their search engine results.

10) Triangle Count: It counts the number of different triangles in a directed or undirected graph.

TABLE I. Representative Applications

Type	***Application***	***Computational Characteristics***	***Abnormal Data Characteristics***
SQL query	Scan	Filter operation	LD, SKD
	Aggregate	Aggregated operation	
	Join	Associated operation	
	Mix	Filter,Aggregated and Associated operation	
Machine Learning	Logistic Regression	classification algorithm, Iterative computation	LD, SD, HD, AD
	K-means	Clustering Algorithm, Iterative calculation	
	Decision Tree	Classification/Regression, Breadth-first tree	LD , HD, AD
	Random Forest	Classification/Regression, Breadth-first tree	
Graph	PageRank	Iterative calculation	LD, SD, AD
	TriangleCount	Iterative calculation	

B. Abnormal input data generation

Based on the summarized computational characteristics, this benchmark generates abnormal data for each application as shown in Table I. Abnormal data characteristics include: *large data volume* (LD), *skewed data* (SKD), *sparse data* (SD), *high dimension data* (HD), and *abnormal distribution data* (AD). How to select the abnormal data characteristics for different types of applications is summarized below.

1) SQL Query: The *Scan*, *Aggregate*, *Join* applications deal with key/value pairs. The computation complexity of the filter, aggreated, and associated operations are related to the key distribution, so this benchmark selects skewed data (i.e., generating uneven key distribution) as the abnormal input data. *Join* application is also related to the order of the operations.

2) Machine Learning: Machine Learning applications such as *Logistic Regression* and *K-means* take matrix-like features as input data, so the related data characteristics are: (1) matrix total size, (2) matrix dimension, (3) distribution of each matrix column, and (4) matrix sparsity. Other tree-based applications such as *Decision Tree* and *Random Forest* hold breadth-first trees in memory and use random sampling to train the trees. When the data dimension is high, the resource utilization will be high too. In addition, the random sampling method will affect the stability of the computing results.

3) Graph: PageRank and TriangleCount applications use vertex-centric partition. In each interation, each vertex needs to send its computation results to its adjacent vertices. So, the computation complexity of these applications are related to the edge distribution (i.e., the degree distribution of vertices). As a result, this benchmark generates skewed graph (i.e., some vertex has too many adjacent vertices) as the input data.

Fig. 2 illustrates the process of generating abnormal input data for the *Random Forest* application. The computational characteristics of *Random Forest* application are breadth-first tree and random sampling. Then, we select the corresponding data features, namely large-scale, high-dimensional, and abnormal distribution, to generate the abnormal input data.

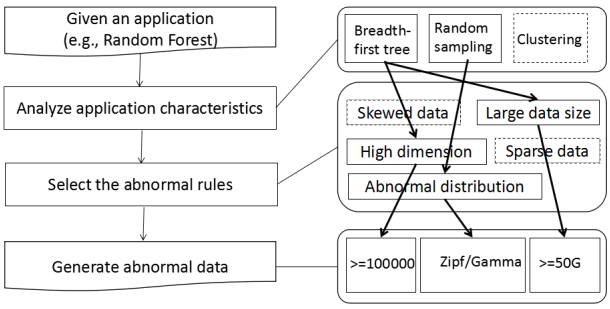

Fig. 2. Generate abnormal data for the Random Forest application

C. Configuration combination test

After generating the input data, the next task is to combine system-specific and application-specific configurations (e.g., as shown in Table II) to test the applications.

TABLE II. CONFIGURATIONS OF THE RANDOMFOREST APPLICATION

Type	*Configuration*	*Description*
System-specific configurations	Input split number	Data parallelism
	Partition number	Task parallelism
Application-specific configurations	maxBins	Maximum number of bins used for splitting features
	numClasses	Number of classes for classification
	numTrees	Number of trees in the random forest
	maxDepth	Maximum depth of the tree

The main problem is that the configuration combination space is too large. Suppose that an application has n configurations, where the i-th configuration has m_i optional values. So, the combination space is $O(m_1*m_2*\cdots*m_n)$. However, if the configurations satisfy the following two assumptions, the combination space can be reduced to $O(n)$.

1. The configurations are independent of each other.
2. The m_i values of configuration i are positively/negatively correlated with the applications' performance (e.g., execution time or resource usage).

Based on the above two assumptions, the application's performance will become worst (may trigger runtime errors) when the configurations take boundary values. Accordingly, we designed a greedy algorithm (Algorithm 1) to combine the configurations.

For example in Fig. 3 a), the application has three configurations. At first, this algorithm chooses the low boundary value of each configuration (i.e., 2-1-1). Then it changes the first configuration combination to be the high boundary value 100. Now the configuration is 100-1-2. If the application's resource utilization of (2-1-1) is less than that of (100-1-2), the algorithm will fix 100 as the first configuration. Next, the algorithm repeats this selection on the other two configurations as shown in Fig. 3 b), c) and d). Finally, the application may generate runtime errors under the worst configuration combination (i.e., 100-10-2) in Fig. 3 d).

However, if the configurations do not satisfy the given two assumptions, this benchmark uses binary search to select the worst value in each configuration. The average computing complexity is $O(\log m_1*\log m_2*\ldots*\log m_n)$.

Algorithm 1: Greedy configuration combination test

1. Give the range of each configuration.
2. Select a combination of each threshold value of each configuration, then test, and record the resource occupancy.
3. Change the value of a configuration to another threshold, then test, and record the resource occupancy.
4. Compare the resource usage in the last two combinations of configurations, and preserve the critical value of poor performance.
5. Return to step 2, and repeat until the exception or ends of test. If an exception was found, the configuration was found which can cause failures. If no exception was found, the configuration with worst resource usage or worst performance was found.
6. End of test.

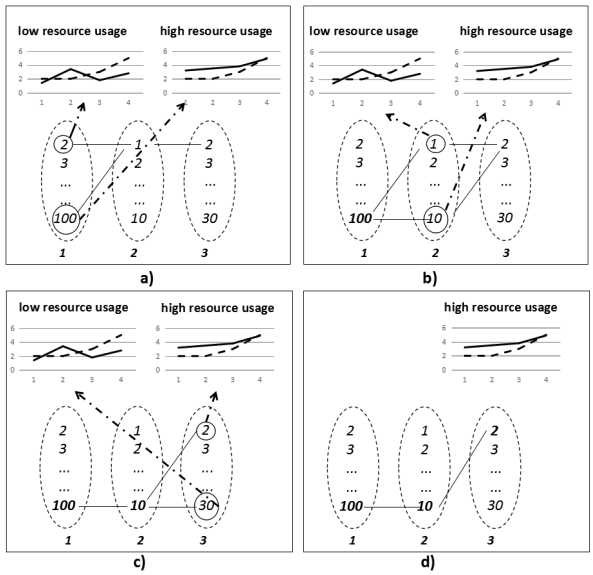

Fig. 3. The process of reducing configuration combination space

D. Test report generation

After generating the abnormal data and running configuration combination test, a task report generator is performed to analyze the application's runtime information and generate test reports. The test reports mainly include: 1) what the runtime error is, 2) what abnormal data causes the runtime error, 3) what the worst configurations are.

TABLE III. PRELIMINARY TEST RESULTS

Application	*Input Data*	*Configurations*	*Errors*
Join	10GB, skewed data	Small table inner join big table	OOM
Mix	10GB, skewed data	A table participates in multiple join operations but not rename it	Wrong results
RandomForest	1 million instances, 1000-dimensional, Gamma-Poisson distribution	numTrees = 100, maxDepth = 30, dimensions = 1000	OOM
LogisticRegression	1.05GB sparse data with 1000 dimensions	4 executor (2 cores, 8G), split=134.13MB, partition number = 8	OOM, Timeout
PageRank	10G data, 1 million vertices, 20 million edge	4 executor (2 cores, 8G), convergence accuracy = 0.001	OOM

III. PRELIMINARY RESULTS

A. Experimental setup

We performed this reliability benchmark on a 10-node cluster (including 1 master node and 9 slave nodes) using Spark-2.0 on Ubuntu-11.04 Operation System. Each node has 4 CPU, 16GB RAM and 2*1TB Disks. We tested each application 5 times and use the mean value.

For SQL applications, the input table schemas (shown in Table IV) are as same as that used in Pavlo *et al*. [15]. However, the input data of all the applications are generated by abnormal input data generator.

TABLE IV. TABLE SCHEMAS

Table name	*Column name*	*Data type*
Rankings	pageURL	VARCHAR
	pageRank	INT
	avgDuration	INT
UserVisits	sourceIP	VARCHAR
	destURL	VARCHAR
	visitDate	DATE
	adRevenue	FLOAT
	userAgent	VARCHAR
	countryCode	VARCHAR
	languageCode	VARCHAR
	searchWord	VARCHAR
	duration	INT

B. Results

The preliminary results are shown in Table III. We found three types of errors (i.e., out of memory (OOM), timeout and wrong results) in five applications.

C. Case studies

1) SQL join

When testing the Join query in Spark SQL, this benchmark generates both normal data and abnormal data (skewed data) for each table shown in Table IV. Since the Join operation is a binary operation, the join order can be changed. So, the join query has two sub-queries as shown in Table V. *BigSmallJoin* denotes *Uservisits* (large table) inner join *Rankings* (small table), while *SmallBigJoin* denotes *Rankings* (small table) inner join Uservisits (large table). Table IV shows the results of the two *Join* operations.

Out of memory error occurs in the second *SmallBigJoin*, where a small table inner joins a large table with skewed data. The execution time of the parallel tasks in *BigSmallJoin* and *SmallBigJoin* applications are shown in Fig. 4.

When a given data set is skewed, the number of processed records on a certain task increases significantly. The reason is that when the same key has too many values, these values will be pushed to the same task in shuffle phase. In this situation, the execution time of this task is far longer than that of the other tasks. By analyzing the inner join implementation in Spark, we found that: when two tables inner join each other, the first table is considered as a driven table, and the second table is considered as a buffer table. It will traverse each record in the drive table, look for the corresponding matching records in the buffer table, and put records into the matching table. So when we consider a large table as a buffer table, the matching records will be huge. If there is a large table with a seriously skewed data, the matching table will occupy much more memory, and out of memory error will occur when we query the relevant key.

TABLE V. TEST RESULTS OF JOIN QUERY

SQL Type	*Data type*	*Execution Time*
BigSmallJoin	Normal data（large table)	51s
	Skewed data（small table）	59s
SmallBigJoin	Normal data（small table）	56s
	Skewed data（large table）	Failed

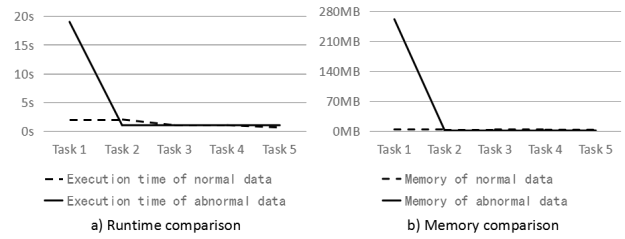

Fig. 4. Comparison of normal and skew data

1) Random Forest Application

The configurations of the *Random Forest* application are shown in Table II. The generated abnormal data is 23.7GB with 104 dimensions. The data distribution is Gaussian distribution. The test results are shown in Table VI. Configurations in group A are the initial values. Group B changes the configuration *numTrees* from 2 to 100. After that, the configuration combination test found that the time and GC time increased significantly. Therefore, the configuration combination algorithm keeps the configuration *numTrees* to be 100 in group C. The next test is to change *maxDepth* to be 100. Out of memory error occurs in group C. If we continue testing using the configurations in group D, out of memory error will also occur. So, the worst configuration combination is 100-5-32-10. However, for the configurations in group C and D, the out of

memory errors will disappear if the data distribution is changed to the uniform distribution. It indicates that the application has potential faults while processing the data with Gaussian distribution.

TABLE VI. TEST RESULTS OF RANDOMFOREST

Configurations	*A*	*B*	*C*	*D*
numTrees	2	100	100	100
maxDepth	5	5	100	5
maxBins	5	5	5	32
Partition num	10	10	10	10
Running time	6.4min	41min	OOM	OOM

IV. RELATED WORK

The reliability of big data applications/systems has emerged as a critical problem for both academia and industry. Many researchers have performed empirical studies on big data application/system failures. However, the current benchmarks are not designed for reliability testing.

Failure study on big data applications/systems: Li *et al*. [9] studied 250 failures in SCOPE jobs in Microsoft big data platform, and found 84.5% failures are caused by defects in data processing. They also found 3 OOM errors that are caused by accumulating large data (e.g, all input rows) in memory. Xu *et al*. [6] studied 123 OOM errors in real-world Hadoop/Spark applications and found three causes of out of memory errors: improper configurations, abnormal dataflow and memory-consuming user code. Kavulya *et al*. [7] analyzed 4100 failed Hadoop jobs, and found 36% failures are array indexing errors and 23% failures are IOExceptions. Zhou *et al*. [16] studied the quality issues of big data platform in Microsoft. They found 36% issues are caused by system side defects and 2 issues (1%) are memory issues. Gunawi *et al*. [10] studied 3655 development and deployment issues in cloud systems such as Hadoop MapReduce, HDFS, and HBase. They found 87% issues are software faults, while 13% issues are hardware faults. They also reported 1 OOM error in HBase (users submit queries on large data sets) and 1 OOM error in Hadoop File System (users create thousands of small files in parallel). These studies help us design the abnormal data generator and configuration generator.

Big data benchmarks: Pavlo [15] designed a big SQL benchmark to compare the performance between MapReduce and relational databases. Berkeley AMPLab developed a SQL benchmark [12] to compare the performance among Spark, Hive, Impala, etc. HiBench [13] is designed to test the performance of Hadoop and Spark. BigDataBench [17] includes 14 real-world data sets, and 34 big data workloads. These benchmarks use normal data and fixed configurations to test the performance of big data systems.

V. CONCLUSION AND FUTURE WORK

Big data applications deployed on the cloud platform frequently suffer from runtime errors. However, current benchmarks are designed for performance testing and cannot be directly used for detecting potential faults. In this paper, we design a reliability benchmark for big data systems and implement it on Spark. This benchmark first generates abnormal input data according to the application characteristics, and then uses greedy algorithm to combine system-/application-specific configurations for testing. Preliminary results show that this benchmark can detect application faults.

In the future, we will build more applications into the benchmark and implement this benchmark on more systems such as Flink.

VI. ACKNOWLEDGEMENTS

This work is supported by the National Key Research and Development Program of China (2016YFB1000103) and Youth Innovation Promotion Association, CAS (No. 2015088).

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 137–150.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[3] "Apache Flink." [Online]. Available: https://flink.apache.org/.

[4] A. Li, et al. "CloudCmp: comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (SIGCOMM).*,2010.

[5] Lenk, Alexander, et al. "What are you paying for? performance benchmarking for infrastructure-as-a-service offerings." *Cloud Computing (CLOUD), 2011 IEEE International Conference on. IEEE,* 2011.

[6] L. Xu, W. Dou, F. Zhu, C. Gao, J. Liu, H. Zhong, and J. Wei, "Experience report: A characteristic study on out of memory errors in distributed data-parallel applications," in *26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 518–529.

[7] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010.

[8] "Spark reduce operation taking too long." [On- line]. Available: http://stackoverflow.com/questions/33558593/ spark-reduce-operation-taking-too-long.

[9] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, "A characteristic study on failures of production distributed data-parallel programs," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 963–972.

[10] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 7:1– 7:14.

[11] "Spark Performance Tests." [Online]. Available: https://github.com/databricks/spark-perf.

[12] "Spark SQL Benchmark." [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/.

[13] "HiBench: the bigdata micro benchmark suite." [Online]. Available: https://github.com/intel-hadoop/HiBench.

[14] M. Armbrust, et al. "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[15] Pavlo, Andrew, et al. "A comparison of approaches to large-scale data analysis." in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

[16] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin, "An empirical study on quality issues of production big data platform," in *ICSE*, 2015.

[17] L. Wang, et al, "Bigdatabench: A big data benchmark suite from internet services," in *HPCA*, 2014.