Experience Report: A Characteristic Study on Out of Memory Errors in Distributed Data-Parallel Applications

Lijie Xu[†‡], Wensheng Dou[†*], Feng Zhu[†‡], Chushu Gao[†], Jie Liu[†], Hua Zhong[†], Jun Wei[†]
[†]State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences
[‡]University of Chinese Academy of Sciences
{xulijie09, wsdou, zhufeng10, gaochushu, ljie, zhongh, wj}@otcaix.iscas.ac.cn

***Abstract*—Out of memory (OOM) errors occur frequently in data-intensive applications that run atop distributed data-parallel frameworks, such as MapReduce and Spark. In these applications, the memory space is shared by the framework and user code. Since the framework hides the details of distributed execution, it is challenging for users to pinpoint the root causes and fix these OOM errors.**

This paper presents a comprehensive characteristic study on 123 real-world OOM errors in Hadoop and Spark applications. Our major findings include: (1) 12% errors are caused by the large data buffered/cached in the framework, which indicates that it is hard for users to configure the right memory quota to balance the memory usage of the framework and user code. (2) 37% errors are caused by the unexpected large runtime data, such as large data partition, hotspot key, and large key/value record. (3) Most errors (64%) are caused by memory-consuming user code, which carelessly processes unexpected large data or generates large in-memory computing results. Among them, 13% errors are also caused by the unexpected large runtime data. (4) There are three common fix patterns (used in 34% errors), namely changing the memory/dataflow-related configurations, dividing runtime data, and optimizing user code logic. Our findings inspire us to propose potential solutions to avoid the OOM errors: (1) providing dynamic memory management mechanisms to balance the memory usage of the framework and user code at runtime; (2) providing users with memory+disk data structures, since accumulating large computing results in in-memory data structures is a common cause (15% errors).

***Keywords*—MapReduce, out of memory, characteristic study**

I. Introduction

In recent years, MapReduce [1] and MapReduce-like distributed frameworks, such as Dryad [2] and Spark [3], have emerged as the representative data-parallel frameworks for large-scale data processing. Their open-source implementation, Apache Hadoop and Apache Spark, are now widely used in academia and industry to develop data-intensive applications, such as Web indexing, click-log mining, machine learning, and graph analysis.

These data-parallel frameworks provide users with simple programming models and hide the details of parallel/distributed execution. This design helps users focus on the data processing logic, but complicates the error diagnosis when users' data-parallel applications (jobs) generate runtime errors. Out of memory (OOM) is a serious and common runtime error in these data-parallel applications. OOM errors can directly lead to the job failure, and cannot be tolerated by frameworks' fault-tolerant mechanisms, such as re-executing the failed map/reduce tasks.

```
FATAL org.apache.hadoop.mapred.Child: Error running child
   : java.lang.OutOfMemoryError: Java heap space
at java.util.Arrays.copyOf(Arrays.java:2882)
at java.lang.AbstractStringBuilder.expandCapacity(
   AbstractStringBuilder.java:100)
...
at cloud9.ComputeCooccurrenceMatrixStripes$MyReducer.
   reduce(ComputeCooccurrenceMatrixStripes.java:136)
at org.apache.hadoop.mapred.Child.main(Child.java:404)
```

Listing 1: OOM stack trace in a reduce task in a Hadoop job

Different from other runtime errors like node crashes, OOM errors are caused by excessive memory usage. While running a data-parallel application, the framework buffers/caches the intermediate data in memory for better performance, and user code (e.g., *map*()) also stores intermediate computing results in memory. Once the size of in-memory data and computing results exceeds the memory limit of the (map/reduce) task, an OOM error will occur in the task. Fig. 1 shows two OOM errors in two tasks in a data-parallel application. At the time of OOM, the framework only throws an OOM stack trace as Listing 1, which cannot directly reflect the root cause.

Unfortunately, there is little work that studies OOM errors in data-parallel applications. Li *et al.* [4] studied 250 failures and fixes in SCOPE jobs, which are SQL-like applications running atop Dryad in Microsoft. They found that most failures are caused by undefined columns, wrong schemas, incorrect row format, and illegal arguments. Kavulya *et al.* [5] analyzed the performance problems and failures in Hadoop jobs from the M45 cluster administrated by Yahoo!. They only reported that the failures are Array indexing errors and IOException errors, without considering the OOM errors. Gunawi *et al.* [6] studied 3655 development and deployment issues in cloud systems, such as Hadoop and HBase [7]. They focused on the hardware/software faults in these systems, without analyzing the OOM errors in data-parallel applications.

To help users understand, fix, and avoid these serious OOM errors, we conduct a characteristic study on 123 real-world OOM errors collected from open forums such as StackOverflow.com and Hadoop/Spark mailing list [8], [9]. These errors occur in Hadoop/Spark applications, including raw MapReduce/Spark code and code generated by high-level languages/libraries, such as Apache Pig [10], Hive [11], and MLlib [12]. For each OOM error, we manually review the

* Corresponding author

users' error descriptions (the data, configurations, user code, etc.) and the experts' answers (the analysis of the root causes and fix suggestions). In order to understand the root causes, verify the fixes, and identify frameworks' memory management weakness, we also reproduced 43 errors. Specifically, our study intends to answer the following research questions:

- **RQ 1:** What are the root causes of OOM errors in distributed data-parallel applications? Are there any common patterns?
- **RQ 2:** How do users fix OOM errors? Are there any common fix patterns?
- **RQ 3:** Are there any potential solutions to improve the data-parallel frameworks' fault tolerance or facilitate the OOM error diagnosis?

The root causes of 123 OOM errors have been identified by the experts (66), users themselves (45), or us (12) in our reproduced errors. We also found that 42 OOM errors have fix suggestions, and 25 out of them have been fixed by users based on the fix suggestions. The 123 errors can be found in our technical report [13]. Our major findings are as follows:

- Although users can configure the buffer size and cache threshold to limit the framework's memory usage, there are still many OOM errors (12%) caused by the large data stored in the framework. This result implies that it is hard for users to configure the right memory quota that balances the runtime memory usage of the framework and user code.
- Abnormal dataflow is another common cause of the OOM errors (37%), which can lead to the unexpected large runtime data, such as large data partition, hotspot key, and large single key/value record. This result suggests that current data-parallel mechanisms do not properly consider the runtime data property (e.g., key distribution) and cannot limit the input data size of user code.
- Most OOM errors (64%) are caused by memory-consuming user code, which carelessly processes unexpected large data or generates large computing results in memory. This result indicates that it is hard for users to design memory-efficient code and predict its memory usage, without knowing the runtime data volume.
- The common fix patterns (proposed in 34% OOM errors and 20% OOM errors are fixed) are adjusting memory/dataflow-related configurations, dividing runtime data, and optimizing user code logic. We are also surprised to see that there are tricky fix patterns, such as redesigning the key and skipping the abnormal data. This result indicates that there is not a unified method to fix the OOM errors. The general fix guide is to limit the data storage, the runtime data, or the memory usage of user code.
- Current data-parallel frameworks provide limited support for OOM error diagnosis and tolerance. There are several potential solutions to improve frameworks' fault tolerance and error diagnosis, such as providing statistical dataflow information, enabling dynamic memory management, and providing memory+disk data structures.

In summary, our main contributions are as follows:

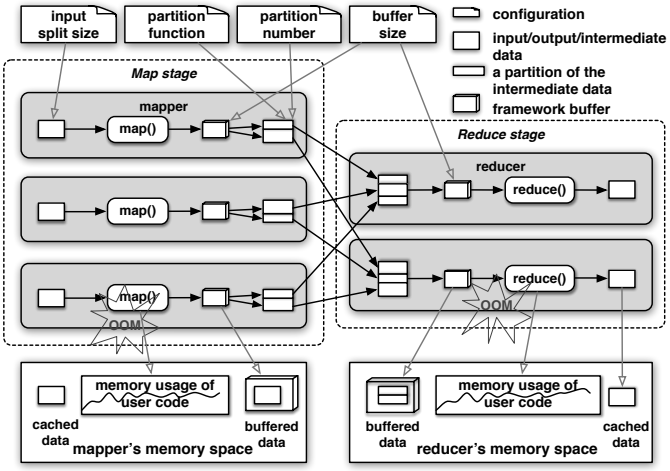

Fig. 1: Two examples of OOM errors in a Hadoop/Spark job

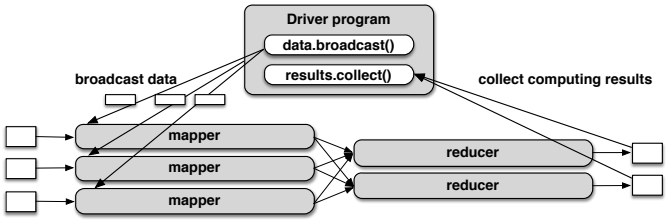

Fig. 2: The driver program in Spark

- We present the first comprehensive study on the OOM errors in distributed data-parallel applications, which can help users understand the root causes and is also useful for further research work.
- We summarize the common fix patterns for most root causes, which can help users fix the OOM errors in practice.
- We provide potential solutions to improve the frameworks' fault tolerance and facilitate the OOM error diagnosis.

The rest of the paper is organized as follows. Section II introduces the basics of data-parallel applications. Section III describes our study methodology. Section IV presents our study results on the root causes of OOM errors, which answers RQ1. Section V presents our study results on fix patterns, which answers RQ2. Section VI describes the potential solutions to avoid the OOM errors, which answers RQ3. Section VII discusses the generality of our study. Section VIII lists the related work and section IX concludes this paper.

II. Background

A distributed data-parallel application can be represented as ⟨*input dataset*, *configurations*, *user code*⟩. The input dataset is usually split into data blocks (e.g., 3 input splits in Fig. 1), and stored on the distributed file system. Before submitting an application to the data-parallel framework, users need to specify user code and the application's configurations.

A. Programming model and user code

MapReduce and MapReduce-like data-parallel frameworks, such as Dryad [2] and Spark [3], share the same programming

model as follows:

$$Map\ Stage: \ map(k_1, v_1) \Rightarrow list(k_2, v_2)$$
$$Reduce\ Stage: \ reduce(k_2, list(v_2)) \Rightarrow list(v_3)$$

In the map stage, $map(k,v)$ reads $\langle k,v \rangle$ records one by one from an input split, processes each record and output new $\langle k,v \rangle$ records. In the reduce stage, the framework groups the $\langle k,v \rangle$ records into $\langle k, list(v) \rangle$ by the key k, and then launches $reduce(k, list(v))$ to process each $\langle k, list(v) \rangle$ group. Hadoop natively supports this programming model, while Dryad and Spark provide general and user-friendly operators, such as *map*(), *flatMap*(), *groupByKey*(), *reduceByKey*(), *coGroup*(), and *join*(), which are built on top of *map*() and *reduce*(). Users can also write applications using high-level languages such as SQL-like Pig script [10], which can automatically generate binary *map*() and *reduce*(). For optimization, users can define a *mini reduce*() named *combine*(). We regard *combine*() as *reduce*() since they usually share the same code for aggregation.

Apart from *map*() and *reduce*(), users need to write a driver program (shown in Fig. 2) to submit an application to Spark. The driver program can also (1) generate and broadcast data to each task; (2) collect the tasks' outputs. So, in this paper, we regard user code as *map*(), *reduce*(), and the driver program.

B. Dataflow

A distributed data-parallel application consists of one or multiple MapReduce jobs. As shown in Fig. 1, a job will go through a map stage and a reduce stage (a Dryad/Spark job can go through multiple map and reduce stages connected as a directed acyclic graph). Each stage contains multiple map/reduce tasks (i.e., mappers/reducers). For parallelism, the mappers' outputs are partitioned and each partition is shuffled to a corresponding reducer by the framework. Dataflow refers to the data that flows among mappers and reducers.

The major difference between MapReduce and Dryad/Spark is that Dryad/Spark supports *pipeline*. In the pipeline, map/reduce tasks can continuously execute multiple user-defined functions (e.g., run another *map*() after a *map*()) without storing the intermediate results (e.g., results of the first *map*()) into the disk. In Spark, users can also explicitly tell the framework to cache reusable intermediate results in memory (e.g., outputs of *reduce*() used for the next job) using *cache*().

C. Configurations

The application's configurations consist of two parts: (1) Memory-related configurations affect the memory usage directly. For example, *memory limit* defines the memory space (heap size) of map/reduce tasks and *buffer size* defines the size of framework buffers. (2) Dataflow-related configurations affect the volume of data that flow among mappers and reducers. For instance, *partition function* defines how to partition the $\langle k,v \rangle$ records outputted by *map*(), while the *partition number* defines how many partitions will be generated and how many reducers will be launched.

III. Methodology

A. Subjects

We took real-world data-parallel applications that run atop Apache Hadoop and Apache Spark as our study subjects. Since there are not any special bug repositories for OOM errors (JIRA mainly covers the framework bugs), users usually post their OOM errors on the open forums (e.g., StackOverflow.com and Hadoop/Spark mailing list). We totally found 1151 issues by searching keywords such as "Hadoop out of memory" and "Spark OOM" in StackOverflow.com, Hadoop mailing list [8], Spark user/dev mailing list [9], developers' blogs, and two MapReduce books [14], [15]. We manually reviewed each issue and only selected the issues that satisfy: (1) The issue is a Hadoop/Spark OOM error, since 786 issues are not OOM errors (e.g., only contain partial keywords "Hadoop Memory"). (2) The OOM error occurs in the Hadoop/Spark applications, not other service components (e.g., the scheduler and resource manager). In total, 276 OOM errors are selected. These errors occur in diverse Hadoop/Spark applications, such as raw MapReduce/Spark code, Apache Pig [10], Apache Hive [11], Apache Mahout [16], Cloud[9] [17] (a Hadoop toolkit for text processing), GraphX [18] and MLlib [12]. Based on the approach in Section *B*, we identified the root causes of 123 OOM errors (listed in Table I). The root causes of the other 153 OOM errors are unknown. Therefore, our study only performs on these 123 OOM errors (a.k.a. failures).

B. Root cause and fix pattern identification

For each OOM error, we manually reviewed the user's error description and the professional answers given by experts (e.g., Hadoop/Spark committers from cloudera.com, experienced developers from ebay.com, and book authors). Out of the 276 OOM errors, the root causes of 123 errors have been identified in the following three scenarios: (1) The experts identified the root causes and users have accepted the experts' professional answers. (2) Users identified the root causes themselves. They have explained the causes (e.g., abnormal data, abnormal configurations, and abnormal code logic) in their error descriptions and just asked how to fix the errors. (3) We identified the causes by reproducing the errors in our cluster and manually analyzing the root causes.

Similar to the root causes, we collected the fix patterns from 42 OOM errors, where the experts provided fix methods or users reported the successful fix methods (25 errors). Then, we merged the similar fix methods together and got 11 fix patterns.

C. OOM error reproduction

To fully understand the root causes and fix patterns of OOM errors, we have reproduced 43 OOM errors (35%), which have detailed data characteristics, reproducible user code, and OOM stack traces. Since we did not have the same dataset as the users', we used the public dataset (Wikipedia) and synthetic dataset (random text and a well-known benchmark [19]) instead. The experiments were conducted on a 11-node cluster using Hadoop-1.2 and Spark-1.2. Each node has

TABLE I: DISTRIBUTION OF OUR STUDIED OOM ERRORS

Framework	Sources	Raw code	Pig	Hive	Mahout	Cloud9	GraphX	MLlib	Total	Reproduced
Hadoop	StackOverflow.com	20	4	2	4	0	0	0	30	16
	Hadoop mailing list	5	5	1	0	1	0	0	12	6
	Developers' blogs	2	1	0	0	0	0	0	3	2
	MapReduce books	8	3	0	0	0	0	0	11	2
Spark	Spark mailing list	16	0	0	0	0	1	2	19	3
	StackOverflow.com	42	0	0	0	0	1	5	48	14
Total		**93**	**13**	**3**	**4**	**1**	**2**	**7**	**123**	**43**

16GB RAM, and the memory limit of each mapper/reducer is configured to 1GB. We made sure that the thrown stack trace of each OOM error is the same as that reported by the users.

D. Threats to validity

1) Representativeness of applications: Our study only covers the applications that run atop open-source frameworks (i.e., Hadoop and Spark). Although many companies (e.g., Facebook and Yahoo!) use these two frameworks, some other companies have built their own (e.g., Dryad in Microsoft). We have not studied the applications on these private frameworks.

2) Pattern completeness: Since the root causes of 153 OOM errors are unknown, there may be some new OOM cause patterns or fix patterns. Moreover, the users' error descriptions and the experts' professional answers may only cover the major root cause when an OOM error has multiple root causes.

3) Bias in our judgment: Although we tried our best to understand, identify, and classify the root causes, there may be still inaccurate identification or classification. For example, if there are multiple professional answers, we only select the ones that are accepted by users. However, the other answers may also be right in some cases.

IV. CAUSE PATTERNS OF OOM ERRORS

Although the root causes of the 123 OOM errors are diverse, we can classify them into 3 categories and 10 cause patterns according to their relationship with the data storage, runtime dataflow, and user code. Table II illustrates the concrete cause patterns and the corresponding number of errors in Hadoop and Spark applications. Most OOM errors (64%) are caused by memory-consuming user code. The second largest cause is abnormal dataflow (37% errors). Note that 13% errors are caused by both memory-consuming user code and abnormal dataflow. The left 12% errors are caused by the large data stored in the framework. We next go through each OOM cause pattern and interpret how they lead to OOM errors.

A. Large data stored in the framework

This category has two data storage related cause patterns. In the first pattern, the framework buffers large intermediate data in memory. In the second pattern, users purposely cache large data in the framework for reuse.

1) Large data buffered by the framework: To lower disk I/O, data-parallel frameworks usually allocate in-memory buffers to temporarily store the intermediate data (the outputs of *map*() or input data of *reduce*() in Fig. 1). There are two types of buffers: (1) *fixed buffer*. The buffer itself occupies a large memory space such as Hadoop's *map buffer* (a large byte[]). (2) *virtual buffer*. This is a threshold that limits how much memory space can be used to buffer the intermediate data. Both Hadoop and Spark have a virtual buffer named *shuffle buffer*. When users configure large buffer size, large data will be stored in memory and OOM errors may occur.

This pattern has 8 OOM errors (6%). Four errors are caused by the large map buffer (i.e., *io.sort.mb*) in Hadoop. For example, a user configures a 300MB fixed buffer, but the mapper's memory is only 200MB (e01)[1]. Four errors are caused by the large virtual buffer in Hadoop and Spark. For example, a user sets the virtual buffer to be 70% of the reducer's memory, which is too large (should be 30% in this case) and leads to the OOM error (e02).

2) Large data cached in the framework for reuse: Different from the buffered data, cached data refers to the data that are purposely cached in the framework by users for reuse. In some applications, especially iterative machine learning and graph applications, such as PageRank and K-means clustering, the input data (e.g., the graph and training data) and the intermediate data (e.g., weight parameters) will be reused across multiple jobs. Hadoop reuses data between MapReduce jobs through writing/reading distributed file system, while Spark provides users with an interface *cache*() to cache the data in memory. Large cached data can directly cause OOM errors or reduce the available memory space for data processing.

This pattern has 7 OOM errors (6%). Three errors are caused by caching large reusable data (RDD in Spark). For example, the application throws an OOM error, while trying to cache the eventual large RDD (e03). Three errors are caused by continuously caching data into memory. For example, as the iteration goes on in a machine learning application named *SVDPlusPlus*, more and more graphs are cached and the OOM error eventually happens (e04). The last error is caused by caching the large data broadcasted from the driver (e05).

Finding 1: Although users can configure the buffer size and cache threshold to limit the framework's memory usage, there are still many OOM errors (12%) caused by the large data stored in the framework.
Implication: It is not easy for users to configure the right memory quota that balances the runtime memory usage of the framework and user code.

[1](eXX) denotes the OOM error with ID=eXX in Table IV in the Appendix.

TABLE II: CAUSE PATTERNS OF THE OOM ERRORS

Category	Cause patterns	Pattern description	Hadoop	Spark	Total	Ratio
Large data stored in the framework	Large data buffered by the framework	Large intermediate data are temporarily stored in the framework buffers	6	2	8	6%
	Large data cached in the framework for reuse	Users explicitly cache large data in the framework for reuse	0	7	7	6%
	Subtotal		**6**	**9**	**15**	**12%**
Abnormal dataflow	Improper data partition	Some partitions are extremely large	3	13	16	13%
	Hotspot key	Large $\langle k, list(v)\rangle$	15	8	23	18%
	Large single key/value record	Large $\langle k, v\rangle$	6	1	7	6%
	Subtotal		**24**	**22**	**46**	**37%**
Memory-consuming user code	Large external data loaded in the code	User code loads large external data	8	0	8	6%
	Large intermediate results	User code generates large computing results while processing a single record	4(3)	2	6(3)	5%
	Large accumulated results	User code accumulates large intermediate computing results in memory	30[13]	10[1]	40[14]	33%
	Large data generated in the driver	The driver generates large data	0	9	9	7%
	Large results collected by the driver	The driver collects tasks' large outputs	0	16	16	13%
	Subtotal		**42**	**37**	**79**	**64%**
Total			**72**	**68**	**123+17**	**113%**

Notations: 4(3) means that 3 out of the 4 OOM errors are also caused by *large single key/value record*. 30[13] means that 13 out of the 30 OOM errors are also caused by *hotspot key*. 113% means that 13% errors have two OOM cause patterns.

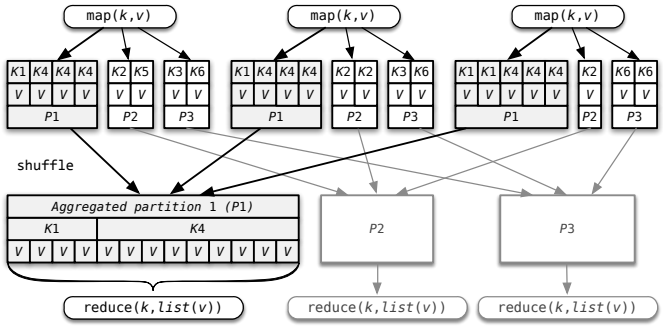

Fig. 3: An example of large data partition and hotspot key

B. Abnormal dataflow

Since the data is processed in a distributed fashion and data volume (e.g., key/value size and distribution) in each processing step is determined at runtime, the framework cannot avoid generating abnormal dataflow, such as large data partition, hotspot key, and large single record. The abnormal dataflow can directly cause OOM errors.

1) Improper data partition: Partition is a common technique used in data-parallel frameworks to achieve parallelism. In Hadoop/Spark, *map*() outputs its $\langle k, v\rangle$ records into different data partitions according to the k's partition *id*. For example in Fig. 3, $k1$ and $k4$ have the same partition *id* (suppose *id* = *hash*(*key*) % *partitionNumber*). Records in the same partition will be further processed by the same user code (*reduce*() or *combine*()). Two cases can cause improper data partition: (1) When the partition number is small, all the partitions would be large. (2) An unbalanced partition function makes some partitions become extremely larger than the others. Commonly used partition functions, such as *hash* and *range* partition, cannot avoid generating unbalanced partitions. Improper data partition can lead to large in-memory data. For example in Fig. 3, if $P1$ is much larger than $P2$ and $P3$, the reducer that processes $P1$ will need to shuffle and buffer more data in memory. More aggressively, $P1$ will be completely buffered in memory in Spark, if the *spill* configuration is set to be false. Improper data partition can also lead to large input data for the following user code to process. Since the memory usage of user code is usually related to the volume of input data, large input data can lead to excessive memory usage of user code. For example in Fig. 3, *reduce*() may run out of memory while processing the large partition $P1$.

This pattern has 16 OOM errors (13%). Seven errors are caused by small partition number. For example, after increasing the partition number to 1000, the user reports each partition holds far less data and the OOM error is solved (e06). Four errors are caused by the unbalanced partitions. For example, a user reports most data is skew to only 2 reducers and one of them gets more than half of the data (e07). In the left 5 errors, users report the data partition is very large without detailed descriptions about the partition number and function.

2) Hotspot key: A data partition is a coarse-grained record collection, in which the records still have different keys. Although these records will be processed by the same user code, they are first aggregated into different $\langle k, list(v)\rangle$ groups by the key. Then, user code (*reduce*() or *combine*()) will process the groups one by one. *Hotspot key* means that some groups contain much more records than the others. The partition number can affect the size of each partition, but it cannot affect the size of each group because the group size depends on how many records have the same key at runtime. For example in $P1$ in Fig. 3, if $\langle k4, list(v)\rangle$ is much larger than $\langle k1, list(v)\rangle$, the framework may run OOM while aggregating the $\langle k4, list(v)\rangle$. Furthermore, the following *reduce*() may run OOM while processing the large $\langle k4, list(v)\rangle$.

This pattern has 23 OOM errors (18%), all of which are caused by the huge values associated with one key. For

example, a user reports some keys only return 1 or 2 items but some other keys return 100,000 items (e08). Another user reports the key ⟨*custid, domain, level, device*⟩ is significantly skewed, and about 42% of the records have the same key (e09). Another 6 errors are caused by the hotspot key in Spark's aggregation operators such as *groupByKey*(), *reduceByKey*(), and *cogroup*(). For example, a user reports there are too many values for a specific key and these values cannot fit into memory (e10). The experts interpret that current *groupByKey*() requires all of the values for one key can fit in memory (e11).

3) Large single key/value record: Large key/value record means that a single $\langle k, v \rangle$ record is too large. Since user code needs to read the whole record into memory to process it, the large record itself can cause the OOM error. Large record can also cause user code to generate large intermediate results, which will be detailed in Section *C*. Since the record size is determined at runtime, dataflow-related configurations cannot control its size.

This pattern has 7 OOM errors (6%), all of which have the information that a single record is too large. For example, a user reports that the memory is 200MB, but the application generates a 350MB record (a single line full of character *a*) (e12). Another user reports that some records are 1MB but some are 100MB non-splittable blob (e13). More surprisingly, a user reports that the application is trying to send all 100GB data into memory for one key because the changed data format makes the terminating tokens/strings do not work (e14).

Finding 2: Abnormal dataflow is another common cause of the OOM errors (37%), which can lead to the unexpected large runtime data, such as large data partition, hotspot key, and large single key/value record.
Implication: Current data-parallel mechanisms do not properly consider the runtime data property (e.g., key distribution) and cannot limit input data size of user code.

C. Memory-consuming user code

Different from traditional programs, user code in data-parallel applications has an important *streaming-style* feature. In the streaming style, the $\langle k, v \rangle$ records are read, processed and outputted one by one. So, once an input record is processed, this record and its associated computing results will become useless and reclaimed, unless they are purposely cached in memory for future use. Based on this feature, we summarized two cause patterns: *large intermediate computing results* (generated for a single record) and *large accumulated results*. Another pattern is that user code loads large external data in memory. In addition, the driver program can trigger OOM errors while generating large data in memory or collecting large outputs of the tasks.

1) Large external data loaded in the user code: Different from the buffered/cached data managed by the framework, external data refers to the data that is directly loaded in user code. In some applications, user code needs to load external data from local file system, distributed file system, database, etc. For example, in order to look up whether the key of each input record exists in a dictionary, user code will load the whole dictionary into a *HashMap* before processing the records. Large external data can directly cause OOM errors.

This pattern has 8 OOM errors (6%). Three errors occur in Mahout applications, whose mappers try to load large trained models for classification (e15) and clustering (e16). One error occurs in a Hive application that tries to load a large external table (e17). One error occurs in a Pig script, whose UDF (User Defined Function) tries to load a big file (e18).

2) Large intermediate results: Intermediate results refer to the in-memory computing results that are generated while user code is processing a $\langle k, v \rangle$ record. This pattern has two sub-patterns: (1) the input record itself is very large, so the intermediate results may become large too. For example, if a record contains a 64MB sentence, its split words are also about 64MB. (2) Even a small input record may generate large intermediate results. For example, if the value of a record has two sets, Cartesian product of them is orders of magnitude larger than this input record.

This pattern has 6 OOM errors (5%). In 3 errors, user code generates large intermediate results due to the extremely large input record. In 1 error, *reduce*() generates very long output record in memory (e19). In 1 error, a machine learning application constructs large models in memory for matrix factorization (e20). The last error occurs in a text processing application (e21), which aims to lemmatize the words in a large document using a third-party library *StanfordLemmatizer* as follows. During processing each *line* of the document, *lematize*() allocates large dynamic programming data structures, which might be 3 times larger than the *line* (interpreted by the library author). The OOM error occurs in *lematize*() because the *line* under processing is extremely large.

```
public class Mapper {
  StanfordLemmatizer slem = new StanfordLemmatizer();
  public void map(Long key, Text value) {
    String line = value.toString();
    for(String word: slem.lemmatize(line))
      emit(word, 1);
  }
}
```

3) Large accumulated results: If the intermediate computing results generated at current input record are cached in memory for future use, they become accumulated computing results. So, more records are processed, more intermediate results may accumulate in memory. For example, to deduplicate the input records, *map*() may allocate a *Set* to keep each unique input record. If there are many distinct input records, the *Set* will become large too. For *reduce*(), it can generate large accumulated results during processing a large $\langle k, list(v) \rangle$ group, which could be a result of hotspot key.

This pattern has 40 OOM errors (33%). In 11 errors, users allocate in-memory data structures to accumulate the input records. For example, a user allocates an *ArrayList* to keep all the values for a key, which might contain 100 million values (e22). In other errors, users try to accumulate the intermediate results, such as the word's frequency of occurrence (e23) and the training weights (e24). User code accumulates the

intermediate results to find distinct tuples (e25), perform in-memory sort, compute median value, or take a cross product (e26). The following error occurs in a reducer, which is computing the word co-occurrence matrix of a large document (e27). In this error, *reduce*() allocates a HashMap-like data structure *HMapStIW* to keep each word's neighboring words. Since there are many words in the document and each word has many neighboring words, the accumulative operator *plus*() puts too many words into *map* and causes the OOM error (its OOM stack trace is Listing 1).

```
public class Reducer {
  void reduce(Text key, Iterable<HMapStIW> values) {
    Iterator<HMapStIW> iter = values.iterator();
    HMapStIW map = new HMapStIW();
    while (iter.hasNext()) {
      map.plus(iter.next());
    }
    emit(key, map);
  }
}
```

4) Large data generated/collected by the driver: Although the driver program does not directly process the $\langle k, v \rangle$ records, there are two OOM cause patterns in it: (1) The driver generates large data in memory. (2) The driver collects large computing results from the tasks to compute the final results.

The first pattern has 9 OOM errors (7%). The generated large data is used for broadcasting or local computation. For broadcasting, a driver generates a 1GB array (e28), and another one generates a 0.15GB variable (e29). For local computation, a driver generates an 8000×8000 dense matrix about 256MB (e30). Another driver generates 400 million doubles for computing the conditional probabilities (e31).

The second pattern has 16 OOM errors (13%). For example, a driver uses *graph.edges.collect*() to collect all the edges of a 4.5GB graph into memory (e32). Another driver collects the large computing results of *reduceByKey*(), which have 200 million words (e33). More interestingly, in an iterative application, the driver collects the tasks' outputs in each iteration, which causes the memory usage to increase gradually and finally triggers the OOM error (e34).

Finding 3: Most OOM errors (64%) are caused by memory-consuming user code, which carelessly processes unexpected large data or generates large computing results. **Implication:** It is hard for users to design memory-efficient code and predict the memory usage of user code, without knowing the runtime data volume.

V. Common Fix Patterns for OOM Errors

It is not easy to fix the OOM errors, since the causes may be related to data storage, runtime dataflow, and user code. As a result, users usually attempt to enlarge the memory limit to resolve the OOM errors. However, it is not a reliable solution since the enlarged memory space can be filled up again.

To investigate reliable solutions, we summarized the fix patterns from the users' fix practice, experts' fix suggestions, and our practice in fixing the reproduced OOM errors. In total, we found 11 fix patterns from 42 OOM errors. Eight patterns are conventional, which try to change memory/dataflow-related configurations, divide data into small pieces, and optimize user code logic. We are also surprised to see 3 tricky patterns, which try to redesign the key, skip the abnormal data, and change the application-related parameters. Table III summarizes the common fix patterns for each cause pattern (3 cause patterns lack fix patterns). In this table, the label *C* and *U* denote whether the pattern needs to modify the configurations (*C*) and user code (*U*). *Errors(n)* represents how many OOM errors belong to this fix pattern and *n* errors have been fixed. Next, we will detail each fix pattern and the related errors.

A. Data storage related fix patterns

There are two fix patterns for reducing the memory usage of data storage. To perform the two patterns, users only need to adjust the memory-related configurations.

1) Lower framework buffer size: Lowering framework buffer size means decreasing the map buffer size or decreasing the shuffle buffer size. Both of them can directly reduce the size of buffered data at the expense of more disk I/O.

In total, six OOM errors are fixed by this pattern. Three errors are fixed by lowering map buffer in Hadoop (e.g., changing *io.sort.mb* from 128MB to 64MB (e35)). Two errors are fixed by lowering shuffle buffer in Hadoop (e.g., changing *shuffle.input.buffer.percent* from 70% to 30%, but it leads to much longer (2-hour) execution time (e02)). The last error is fixed by removing the shuffle buffer in Spark (i.e., setting *spark.shuffle.memoryFraction* to 0), which results in 20% longer execution time (e36).

2) Lower the cache threshold: Cache threshold is a configuration that limits the memory space used for data cache. Lowering cache threshold directly reduces the size of cached data. However, the expense is that large data block cannot be cached in memory. To solve this problem, Spark provides a memory+disk cache mechanism, which has three data storage levels: memory-only, memory+disk, and disk-only.

Two OOM errors are fixed by this pattern. One error is fixed by changing cache threshold (*spark.storage.memoryFraction*) from 0.66 to 0.1 (e37). The other error is fixed by changing storage level from memory-only to disk-only, at the expense of a little slower execution speed (e20).

B. Dataflow related fix patterns

Since large data partition, hotspot key (i.e., large group), and large single record all can lead to the OOM errors, the proposed fix patterns are to divide the large runtime data (large partition, large group, and large record) into small pieces.

1) Change partition number/function: Data partition size is directly affected by the *partition number* and *partition function*. Increasing partition number is a simple way to reduce the data partition, though it is not 100% reliable. Another fix pattern is to try other partition functions, such as range partition, round-robin partition, or self-designed partition function to get more balanced partitions.

TABLE III: COMMON FIX PATTERNS

Category	Cause patterns	Fix patterns	C	U	Errors(n)
Data storage related fixes	Large data buffered by the framework	Lower framework buffer size	✓		6 (6)
	Large data cached in the framework	Lower cache threshold, or use disk-based cache	✓		2 (2)
Datflow related fixes	Improper data partition	Increase partition number, or change partition function	✓		12 (6)
	Hotspot key	Redesign the key (e.g., using composite key)		✓	3 (0)
	Large single key/value record	Split the large record into multiple small records		✓	4 (1)
User code related fixes	Large accumulated results	Change accumulative operator to streaming operators		✓	2 (2)
		Do the accumulative operation in several passes		✓	3 (1)
		Spill partial accumulated results into disk		✓	3 (1)
		Skip the abnormal data		✓	2 (2)
	Large results collected by the driver	Use tree aggregation instead of direct collect()		✓	3 (2)
		Adjust application's parameters	✓		2 (2)
Total					**42 (25)**

In 12 OOM errors, users are suggested to use this pattern. Six errors are fixed by increasing partition number (i.e., *reduce number* in Hadoop and *spark.default.parallelism* in Spark). For example, after increasing the partition number to 1000, a user reports that each partition holds far less data and the OOM problem is solved (e06). One error is partially fixed as the user reports that "my only solution has been to increase reducer counts, but that does not seem to be getting me anywhere reliable". The expert interprets that increasing partition number only works when the keys are evenly distributed (e38). In 1 error, the user attempts to adjust the partition function (choosing custom ranges for a range partitioner) to handle the unbalanced partitions (e39).

2) Redesign the key: Redesigning the key aims to spread the key distribution, so that the large (hotspot) group can be smaller. A simple method is to use composite key instead of a single key. For example, $map()$ can output a key tuple (k_1,k_2) instead of a single key k as follows.

$$map(k,v) \Rightarrow list((k_1,k_2),v)$$
$$reduce((k_1,k_2),list(v)) \Rightarrow list(v)$$

As a result, the original large $\langle k,list(v)\rangle$ group is divided into multiple smaller $\langle (k_1,k_2),list(v)\rangle$ groups.

In 3 OOM errors, users are suggested to use this pattern. In 1 error, the key *vehicle* has a huge amount of values. The expert suggests the user to use $(vehicle,day)$ as a key, which can drastically cut down the data associated to a single key (e40). In another error, the expert suggests the user to subdivide very large groups by appending a hashed value in a small range (e.g., 1-10) to large keys (e11). In the last error, the expert suggests the user to suffix 1 or 2 to the keys (e26).

3) Split the large single record: This fix pattern aims to split the large input/output record into small records.

In 4 OOM errors, users are suggested to use this pattern. In 1 error, the user is suggested to break up the 100MB+ record into smaller records in $map()$, so that the further $reduce()$ will receive smaller records (e41). In 1 error, the user is suggested to write the large record out as multiple strings rather than a single super giant string (e42). In 1 error, the user is suggested to use the *pos.maxlen* interface of the library to split long sentence (e21). The last error is fixed by partitioning the large record (posting of the inverted index) into manageable sized chunks and outputting several records for the same key (e43).

C. User code related fix patterns

The following fix patterns aim to fix the OOM errors in $map()$ and $reduce()$.

1) Change accumulation to streaming operators: Since accumulation is a common OOM cause, the experts suggest users to change the accumulative operators to one-pass streaming operators.

Two OOM errors are fixed by this pattern. Both errors occur in the in-memory sort. To sort the $list(v)$ in the $\langle k,list(v)\rangle$, users allocate a *List* in memory to keep each v. This accumulation leads to the OOM error. The experts suggest users to utilize framework's sort mechanism by adding v into the key as (k,v). So, $list(v)$ in the $\langle (k,v),list(v)\rangle$ will be sorted while the framework is sorting the key. As a result, user code just needs to read and directly output the sorted $list(v)$ one by one, which is a one-pass streaming operation (e44).

2) Do the accumulation in several passes: This pattern aims to divide the accumulative operators into several memory-efficient *lightweight* operators. Lightweight operator refers to the operator that performs on partial data or generates limited accumulated results.

In 3 OOM errors, users are suggested to use this pattern. In 1 error, the user is suggested to add a $combine()$ (i.e., a *mini reduce*()) operator to perform partial aggregation after $map()$ and before $reduce()$. As a result, $reduce()$ will receive smaller data (fewer unique values) (e45). In another error, the user is suggested to use $aggregateByKey()$ instead of $groupByKey()$, because the former can aggregate the records locally and reduce the shuffling data (e46). The last error (e47) is fixed, which intends to count the distinct values of column B (for each distinct value of column A) in a table. The user first performs $groupby(A)$ and then counts the distinct B in each group A. However, $count(distinct\ B)$ runs out of memory because a group has too many distinct values. The expert suggests the user to use *two groups*, which have two steps of lightweight operators as follows.

The first step performs $groupby(A,B)$ and directly outputs the keys (i.e., got distinct (A,B)). The second step performs

groupby(A) and directly outputs the values as the final results.

$$\begin{aligned} \textit{Original}: \ & \textit{groupby}(A) \Rightarrow \textit{count}(\textit{distinct } B) \\ \textit{Optimized}: \ & \textit{groupby}(A,B) \Rightarrow \textit{output}(A,B) \Rightarrow \quad (1) \\ & \textit{groupby}(A) \Rightarrow \textit{output}(B) \quad (2) \end{aligned}$$

3) Spill accumulated results into disk: This fix pattern does not aim to optimize the code logic, but just spills the accumulated results into disk before they become too large.

In 3 OOM errors, users are suggested to use this pattern. In 1 fixed error, *map*() emits the partial results after processing every *n* input records (e23). In one Hive error, the user is suggested to lower the *hive.map.aggr.hash.percentmemory* from 0.5 to 0.25, which means that the hash map used in map-side aggregation will be spilled into disk once it occupies 25% of the memory space (e48). Similarly, in one Pig error, the expert suggests the user to set *pig.cachedbag.memusage* to 0.1 or lower, which controls the memory threshold for spilling Pig's internal data (*i.e., bags*) (e25).

4) Skip the abnormal data: In this fix pattern, user code does not process (just skips) the extremely large record/group. We are surprised to see that this tricky pattern is used to fix the OOM error. The reason may be that it is hard to optimize the user code logic, especially when user code invokes a third-party library without available source code. This skip action is also acceptable for the applications that do not need precise results (e.g., perform statistical analysis on very large data).

Two OOM errors are fixed by this pattern. The first error occurs in a video recommendation application, while user code is processing the problematic data (the people who watched 100,000 videos). These people are skipped, because they are probably bots and the application only needs a hundred recent views to compute the person's preferences (e49). In the other error, the useless but hotspot keys are directly skipped (e38).

D. Driver program related fix patterns

The experts have proposed two fix patterns to reduce the driver's memory usage while it is collecting the tasks' outputs.

1) Use tree aggregation: As the name indicates, tree aggregation aggregates the tasks' outputs in a multi-level tree pattern. In the first layer, it aggregates the tasks' outputs two by two, resulting in half of the numbers of the outputs. And then, it continuously does the aggregation layer by layer. The final aggregation will be done in the driver but at that time, the numbers of data will be very small.

Two MLlib OOM errors are fixed by this pattern. The users use tree aggregation to avoid directly collecting too many vectors (e50) and too many gradients (e51) in the driver.

2) Adjust the application's parameters: To fix the OOM error, the users/experts even try to adjust the application's parameters. In some machine learning applications such as SVD and ALS, the size of the tasks' outputs has linear/quadratic relationship with the applications' parameters such as SVD's *k* and ALS's *rank*. Lowering these parameters means that the tasks' outputs will become smaller.

Two OOM errors are fixed by this pattern. The first error is fixed by lowering SVD's parameter *k* to 200 (e52). The second error is fixed by setting the ALS's *rank* value under 40 (e53).

> **Finding 4:** The commonly used OOM fix patterns are adjusting memory/dataflow-related configurations, dividing runtime data, and optimizing user code logic. We are also surprised to see that there are tricky fix patterns, such as redesigning the key and skipping the abnormal data.
> **Implication:** There is not a unified method to fix the OOM errors. The general fix guide is to limit the data storage, the runtime data, or the memory usage of user code.

VI. Current and Future Practice

Since the OOM error cannot be tolerated by current fault-tolerant techniques, it is critical to design new fault-tolerant mechanisms and OOM diagnosis tools.

Learnt from the errors and our practice in the OOM diagnosis, we found several potential solutions that can facilitate the cause diagnosis or improve the frameworks' fault tolerance.

A. Facilitate OOM cause diagnosis

1) Provide statistical dataflow information: Abnormal dataflow is a common OOM cause, but current frameworks provide limited runtime dataflow information. For example, Hadoop only counts the number/size of currently processed input/output records, while Spark requires users to manually count how many records have been processed.

If the frameworks can provide statistical dataflow information, OOM error diagnosis will become easier. The statistical information should contain the statistics (e.g., *min*, *average*, *median*, and *max* number/size) of the input/output records in each group and in each data partition. Based on this information, the frameworks can perform some anomaly detection on the dataflow, such as detecting the unbalancedness of the data partition using statistical methods. We have implemented this feature in our enhanced Hadoop-1.2, and this feature helps us identify the root causes of our reproduced OOM errors, including improper data partition (1 error), hotspot key (6 errors), and large single record (1 error).

B. Improve frameworks' fault tolerance

1) Enable dynamic memory management: From the cause patterns, we can see that 12% OOM errors are caused by the large data stored in the frameworks. This indicates that it is hard for users to configure the right memory quota to balance the memory usage of the framework and user code.

Dynamic memory management aims to automatically balance the memory usage of the framework and user code at runtime. Although it is hard to estimate the memory usage of user code, the framework can constantly monitor the memory usage of the buffered/cached data (*du*) and the total memory usage (*total*). The memory usage of user code can be roughly computed by *total-du*. When the total memory usage achieves a threshold, the framework can pause the user code, spill the cached/buffered data into disk, and then resume the user code. In contrast, the spilled cached data can be read back when the memory usage of user code is lower.

2) Provide memory+disk data structures: In-memory data structures are frequently used in user code for accumulating $\langle k, v \rangle$ records or intermediate results. However, they are error-prone and we found 18 OOM errors (15%) that occur in the data structures: *List*/*ArrayList* (9 errors), *Map*/*HashMap* (5 errors), *Set*/*HashSet* (3 errors), and *PriorityQueue* (1 error).

An ideal solution is to provide users with commonly used memory+disk data structures for accumulation/aggregation. The new data structures can have the same/similar APIs with Java Collections and C++ STL. The difference is that the new data structures can automatically swap between memory and disk depending on the available memory.

Spark has implemented a HashMap-like memory+disk data structure named *ExternalAppendOnlyMap*, in which the keys cannot be removed but the value for each key can be changed. However, this data structure cannot be directly used by users (internally used by the framework in some aggregative operators such as *reduceByKey*()). Other open-source projects, such as STXXL [20] and TPIE [21], have implemented disk-based containers and algorithms, which can process large volumes of data that only fit on disks. We can optimize them for aggregating records and incorporate them into the frameworks.

Finding 5: Current data-parallel frameworks provide limited support for OOM error diagnosis and tolerance.
Implication: There are several potential solutions to improve frameworks' fault tolerance and the error diagnosis, such as providing statistical dataflow information, enabling dynamic memory management, and providing memory+disk data structures for aggregation.

VII. Discussion

Although our study is conducted on Apache Hadoop and Apache Spark, most of our results can be generalized to other MapReduce-like frameworks such as Dryad [2], Naiad [22], Map-Reduce-Merge [23], and the under-developing Apache Flink [24]. These frameworks share the same programming model and data-parallel mechanisms.

Currently, we only studied the OOM errors in MapReduce-like applications. They are representative and have covered many data-intensive applications, such as text processing, SQL processing, machine learning, and graph processing. However, there are some non-MapReduce frameworks designed for special data-intensive applications. For example, Pregel [25] and Powergraph [26] are designed to process large graph, while Apache Storm [27] is designed for stream processing. We leave the investigation on OOM errors in these non-MapReduce applications as our future work.

We also found many OOM errors in Hadoop/Spark applications in Alibaba and Tencent. These OOM errors (e.g., [28] in Tencent) have the same causes and fix patterns with our studied errors (e.g., e47).

VIII. Related work

Failure study on big data applications Many researchers have studied the failures in big data applications/systems. Li *et al.* [4] studied 250 failures in SCOPE jobs and found the root causes are undefined columns, wrong schemas, incorrect row format, etc. They also found 3 OOM errors that are caused by accumulating large data (e.g, all input rows) in memory. The 3 errors can be classified to the *large accumulated results* pattern in our study. Kavulya *et al.* [5] analyzed 4100 failed Hadoop jobs, and found 36% failures are Array indexing errors and 23% failures are IOExceptions. Xiao *et al.* [29] studied the correctness of MapReduce programs. They summarized 5 patterns of non-commutative reduce functions, which will generate inconsistent results if re-executed. Zhou *et al.* [30] 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.* [6] studied 3655 development and deployment issues in cloud systems such as Hadoop and HBase [7]. They 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). Different from the above studies, our work focuses on analyzing the root causes and fixes of OOM errors.

Optimize frameworks' memory management Facade [31] proposes a compiler and runtime to separate data storage from data manipulation: data are stored in the unbounded off-heap, while heap objects are created as memory-bounded facades for function calls. InterruptibleTask [32] presents a new type of data-parallel tasks, which can be interrupted upon memory pressure (excessive GC effort or OOM errors) and execute interrupt handling logics specified by users. The tasks can reclaim part or all of its consumed memory when memory pressure comes, and re-activate when the pressure goes away.

Memory leak analysis Memory leaks can cause OOM errors, so researchers proposed many memory leak detectors, including static detectors [33], [34] and dynamic detectors [35], [36]. Memory leak means that users forget to release the useless objects. In our study, we have not found memory leaks. One reason is that user code is written by GC-based languages (Java/Scala) in our studied applications. The other reason is that it is hard for us to judge whether the large data/results in user code are necessarily or unnecessarily persisted.

IX. Conclusion

The paper presents the first comprehensive study on 123 OOM errors in distributed data-parallel applications. We found that the OOM root causes are memory-consuming user code, abnormal dataflow, and large buffered/cached data. We also summarized the common fix patterns for most OOM root causes. Our findings inspire us to propose potential solutions to improve frameworks' fault tolerance and facilitate the OOM cause diagnosis. We believe our results can help both users and the framework designers to handle OOM errors properly.

X. Acknowledgements

We thank the anonymous reviewers, Hucheng Zhou, Zhenyu Guo, and Sa Wang, for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (61202065, U1435220).

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. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 EuroSys Conference (EuroSys)*, 2007, pp. 59–72.

[3] 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 *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 15–28.

[4] 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.

[5] 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, pp. 94–103.

[6] 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, Seattle (SoCC)*, 2014, pp. 7:1–7:14.

[7] "Apache HBase." [Online]. Available: http://hbase.apache.org/

[8] "Hadoop mailing list." [Online]. Available: http://hadoop-common.472056.n3.nabble.com/Users-f17301.html

[9] "Spark mailing list." [Online]. Available: http://apache-spark-user-list.1001560.n3.nabble.com/

[10] "Apache Pig." [Online]. Available: http://pig.apache.org

[11] "Apache Hive." [Online]. Available: https://hive.apache.org/

[12] "MLlib: Apache Spark's scalable machine learning library." [Online]. Available: https://spark.apache.org/mllib/

[13] "Real-world OOM Errors in Distributed Data-parallel Applications." [Online]. Available: https://github.com/JerryLead/MyPaper/blob/master/OOM-Study.pdf

[14] D. Miner and A. Shook, *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 2012.

[15] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*, ser. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010. [Online]. Available: https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

[16] "Apache Mahout." [Online]. Available: https://mahout.apache.org

[17] "Cloud^9: A Hadoop toolkit for working with big data." [Online]. Available: http://lintool.github.io/Cloud9/

[18] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 599–613.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009, pp. 165–178.

[20] "STXXL: Standard Template Library for Extra Large Data Sets." [Online]. Available: http://stxxl.sourceforge.net/

[21] "TPIE - The Templated Portable I/O Environment." [Online]. Available: http://madalgo.au.dk/tpie/

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 439–455.

[23] H. Yang, A. Dasdan, R. Hsiao, and D. S. P. Jr., "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 1029–1040.

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

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.

[27] "Apache Storm." [Online]. Available: https://storm.apache.org/

[28] "The OOM error in Count(distinct) in Tencent." [Online]. Available: http://download.csdn.net/detail/happytofly/8637461

[29] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 44–53.

[30] 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 *37th International Conference on Software Engineering (ICSE)*, 2015.

[31] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "FACADE: A compiler and runtime for (almost) object-bounded big data applications," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 675–690.

[32] L. Fang, K. Nguyen, G. H. Xu, B. Demsky, and S. Lu, "Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs," in *ACM SIGOPS 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[33] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 480–491.

[34] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2005, pp. 115–125.

[35] M. Jump and K. S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007, pp. 31–38.

[36] G. H. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 151–160.

APPENDIX

TABLE IV: LINKS OF REAL-WORLD OOM ERRORS REFERRED IN THIS PAPER

ID	The real-world OOM errors referred in this paper	Version Info	URL
e01	Out of heap space errors on TTs	Hadoop 0.20.2	http://tinyurl.com/p9prayt
e02	pig join gets OutOfMemoryError in reducer when mapred.job.shuffle.input.buffer.percent=0.70	Pig	http://tinyurl.com/kb4zmot
e03	Kyro serialization slow and runs OOM	Spark	http://tinyurl.com/qbchkfp
e04	[Graphx] some problem about using SVDPlusPlus	GraphX	http://tinyurl.com/pb6857w
e05	Problems with broadcast large datastructure	Spark	http://tinyurl.com/osvc2dg
e06	Why does Spark RDD partition has 2GB limit for HDFS	Spark	http://tinyurl.com/qcugugs
e07	RDD Blocks skewing to just few executors	Spark	http://tinyurl.com/pzp4l3u
e08	Building Inverted Index exceed the Java Heap Size	Hadoop	http://tinyurl.com/ojf9npb
e09	memoryjava.lang.OutOfMemoryError related with number of reducer?	Hadoop	http://tinyurl.com/q6jomja
e10	OutOfMemory in “cogroup”	Spark	http://tinyurl.com/qaofbsc
e11	Understanding RDD.GroupBy OutOfMemory Exceptions	Spark 1.1	http://tinyurl.com/os6hbgo
e12	Hadoop Streaming Memory Usage	Hadoop	http://tinyurl.com/orfv3n3
e13	Hadoop Pipes: how to pass large data records to map/reduce tasks	Hadoop	http://tinyurl.com/phwdob4
e14	Hadoop Error: Java heap space	Hadoop 2.2	http://tinyurl.com/qy6wyj9
e15	OutOfMemory Error when running the wikipedia bayes example on mahout	Mahout	http://tinyurl.com/p3cj4ve
e16	Mahout on Elastic MapReduce: Java Heap Space	Mahout 0.6	http://tinyurl.com/na5wodj
e17	Hive: Whenever it fires a map reduce it gives me this error	Hive 0.10	http://tinyurl.com/p32aqfd
e18	OutOfMemoryError of PIG job (UDF loads big file)	Pig	http://tinyurl.com/ne6o6z3
e19	Writing a Hadoop Reducer which writes to a Stream	Hadoop	http://tinyurl.com/p46zupz
e20	MLLib ALS question	Spark 1.1	http://tinyurl.com/oa5eotk
e21	java.lang.OutOfMemoryError on running Hadoop job	Hadoop 0.18.0	http://tinyurl.com/odydwfx
e22	Why does the last reducer stop with java heap error during merge step	Hadoop	http://tinyurl.com/crbd6q8
e23	MapReduce Algorithm - in Map Combining	Hadoop	http://tinyurl.com/ohcue2r
e24	how to solve reducer memory problem?	Hadoop	http://tinyurl.com/okq74kp
e25	java.lang.OutOfMemoryError while running Pig Job	Pig	http://tinyurl.com/ovpo8th
e26	A join operation using Hadoop MapReduce	Hadoop	http://tinyurl.com/b5m72hv
e27	Set number Reducer per machines	Cloud9	http://tinyurl.com/m4fo6wr
e28	trouble with broadcast variables on pyspark	Spark	http://tinyurl.com/nktoyp4
e29	driver memory	Spark	http://tinyurl.com/oa6bn5f
e30	RowMatrix PCA out of heap space error	MLlib 1.1.0	http://tinyurl.com/p382x4k
e31	Running out of memory Naive Bayes	MLlib 1.0	http://tinyurl.com/nwzq4sr
e32	GraphX does not work with relatively big graphs	GraphX	http://tinyurl.com/qj4fst5
e33	something about rdd.collect	Spark	http://tinyurl.com/ok2zkjn
e34	How to efficiently join this two complicated rdds	Spark 0.9	http://tinyurl.com/ppa9suv
e35	CDH 4.1: Error running child: java.lang.OutOfMemoryError: Java heap space	Cloudera 4.1	http://tinyurl.com/oggnsg3
e36	org.apache.spark.shuffle.MetadataFetchFailedException: Missing an output location for shuffle 0	Spark	http://tinyurl.com/pu3nfzc
e37	[0.9.0] MEMORY_AND_DISK_SER not falling back to disk	Spark 0.9.0	http://tinyurl.com/nbajgc3
e38	Reducer’s Heap out of memory	Pig 0.8.1	http://tinyurl.com/mftlvvv
e39	OOM writing out sorted RDD	Spark	http://tinyurl.com/oe9dj78
e40	Lag function equivalent in an RDD	Spark	http://tinyurl.com/ng23nl6
e41	Hadoop Pipes: how to pass large data records to map/reduce tasks	Hadoop	http://tinyurl.com/phwdob4
e42	OOM with groupBy + saveAsTextFile	Spark 1.1.0	http://tinyurl.com/pa7ells
e43	Efficient Sharded Positional Indexer	Hadoop	http://tinyurl.com/pdbcp7y
e44	OutOfMemory during Plain Java MapReduce	Hadoop	http://tinyurl.com/otq9ucs
e45	Hadoop UniqValueCount Map and Aggregate Reducer for Large Dataset (1 billion records)	Hadoop	http://tinyurl.com/q6vglsu
e46	spark aggregatebykey with collection as zerovalue	Spark	http://tinyurl.com/qh6nacx
e47	ORDER ... LIMIT failing on large data	Pig	http://tinyurl.com/q3ef6ak
e48	Out of memory due to hash maps used in map-side aggregation	Hive	http://tinyurl.com/qyeg9zc
e49	Reducers fail with OutOfMemoryError while copying Map outputs	MapR 3.0.1	http://tinyurl.com/ozavd4q
e50	news20-binary classification with LogisticRegressionWithSGD	MLlib 1.0.0	http://tinyurl.com/p8kw3pd
e51	fail to run LBFS in 5G KDD data in spark 1.0.1?	MLlib 1.0.1	http://tinyurl.com/newu7d7
e52	java.lang.OutOfMemoryError while running SVD MLLib example	MLlib 1.1.0	http://tinyurl.com/pb8lg2b
e53	MLLib/ALS: java.lang.OutOfMemoryError: Java heap space	MLlib	http://tinyurl.com/oxmjugf