Characterizing and Diagnosing Out of Memory Errors in MapReduce Applications

Lijie Xu[a], Wensheng Dou*[a], Feng Zhu[c], Chushu Gao[a], Jie Liu[a], Jun Wei[a,b]

[a]*State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences*
[b]*University of Chinese Academy of Sciences*
[c]*Tencent Inc.*
{*xulijie, wsdou, zhufeng10, gaochushu, ljie, wj*}*@otcaix.iscas.ac.cn*

Abstract

Out of memory (OOM) errors are common and serious in MapReduce applications. Since MapReduce framework hides the details of distributed execution, it is challenging for users to pinpoint the OOM root causes. Current memory analyzers and memory leak detectors can only figure out *what* objects are (unnecessarily) persisted in memory but cannot figure out *where* the objects come from and *why* the objects become so large. Thus, they cannot identify the OOM root causes.

Our empirical study on 56 OOM errors in real-world MapReduce applications found that the OOM root causes are improper job configurations, data skew, and memory-consuming user code. To identify the root causes of OOM errors in MapReduce applications, we design a memory profiling tool *Mprof*. *Mprof* can automatically profile and quantify the correlation between a MapReduce application's runtime memory usage and its static information (input data, configurations, user code). *Mprof* achieves this through modeling and profiling the application's dataflow, the memory usage of user code, and performing correlation analysis on them. Based on this correlation, *Mprof* uses quantitative rules to trace OOM errors back to the problematic user code, data, and configurations.

We evaluated *Mprof* through diagnosing 28 real-world OOM errors in diverse MapReduce applications. Our evaluation shows that *Mprof* can accurately identify the root causes of 23 OOM errors, and partly identify the root causes of the other 5 OOM errors.

Keywords:
MapReduce; out of memory; memory profiler; error diagnosis

1. Introduction

As a representative big data framework, MapReduce [1] provides users with a simple programming model and hides the parallel/distributed execution. This design helps users focus on the data processing logic, but burdens them when the applications running atop this framework generate runtime errors. Since MapReduce applications process large data in memory, out of memory (OOM) errors are *common*. For example, in StackOverflow.com, users complained *why* their MapReduce applications run out of memory [2, 3, 4]. OOM errors are also *serious*, since they can directly lead to the application failures and cannot be tolerated by MapReduce framework's fault-tolerant mechanisms (i.e., OOM errors will occur again if re-executing the failed map/reduce tasks).

In general, a MapReduce application (job) can be represented as $\langle$*input data*, *configurations*, *user code*$\rangle$. The input data is usually stored as data blocks on the distributed file system. Before submitting an application to MapReduce framework, users need to specify the application's configurations and user code. (1) **Memory-related configurations**, such as *buffer size*, define the size of framework buffers, which temporarily store intermediate data in memory. (2) **Dataflow-related configurations**, such as *partition function/number*, affect the volume of data that flows in mappers or reducers. *Partition function* defines how to partition the output key/value records of mappers, while *partition number* defines how many partitions will be generated. (3) **User code**, which refers to the user-defined functions, such as *map*(), *reduce*(), and optional *combine*(). These user-defined functions generate in-memory computing results while processing the key/value records.

While running, a MapReduce application goes through a map stage and a reduce stage (shown in Figure 1). Each stage contains multiple map/reduce tasks (i.e., mappers and reducers), and each task runs as a process on a node. While running a MapReduce application, the framework buffers intermediate data in memory for better performance, and user code also stores intermediate computing results in memory. Once the required memory exceeds the memory limit of a map/reduce task, an OOM error will occur in the task. Figure 1 shows two OOM errors that occur in a map task and a reduce task. When an OOM error occurs, users can only figure out *what* functions/methods are running from the OOM stack trace. However, this error message cannot directly reflect the OOM root causes.

To understand the common root causes of OOM errors, we conducted a characteristic study on 56 real-world OOM errors in MapReduce applications collected from open forums, such as StackOverflow.com and Hadoop mailing list [5]. Our study found 3 types of common root causes with 7 cause patterns. (1) 15% errors are caused by *improper job configurations*, which can lead to large buffered data or improper data partition. (2)

*Corresponding author.

38% errors are caused by *data skew*, which can lead to unexpected large runtime data, including large $\langle k, list(v)\rangle$ group and large single $\langle k, v\rangle$ record. (3) 75% errors are caused by *memory-consuming user code*, which loads large external data in memory or generates large intermediate/accumulated results (28% errors are also caused by *data skew*).

Even with the summarized OOM common causes, it is still challenging to automatically identify the root causes of OOM errors in a running MapReduce application. (1) The application's configurations do not directly affect the memory usage of distributed map/reduce tasks. (2) User code can be written arbitrarily or automatically generated by high-level languages (e.g., SQL-like Pig script [6]), which makes us treat the user code as a black box. With memory analysis tools, such as Eclipse MAT [7], users can figure out *what* objects exist in memory but do not know *where* the objects come from and *why* they become so large. The static and dynamic memory leak detectors [8, 9, 10, 11] can identify memory leak (i.e., *which* objects are unnecessarily persisted in memory). However, OOM errors in big data applications are commonly caused by excessive memory usage, not memory leak.

In this paper, we propose a memory profiling tool *Mprof*. *Mprof* can automatically profile and quantify the correlation between a MapReduce application's runtime memory usage and its static information (input data, configurations, user code). Based on the correlation, *Mprof* uses quantitative rules to trace OOM errors back to the problematic user code, data, and configurations. Some but limited manual efforts are required in the cause identification such as linking the identified cause patterns with the code semantics.

Mprof mainly solves three problems: (1) How to figure out the correlation between an application's runtime memory usage and its static information? *Mprof* solves this problem through modeling and profiling the application's dataflow, memory usage of user code, and performing correlation analysis on them. (2) How to figure out the correlation between memory usage of black-box user code and its input data? We find that user objects (generated by user code) have different but fixed lifecycles, and objects with different lifecycles are related to different parts of the input data. Based on this observation, we design a *lifecycle-aware memory monitoring strategy* that can profile and quantify the correlation between different user objects and their related input data. (3) How to identify the root causes based on the correlation? Two types of quantitative rules are designed: *Rules for user code* can identify the problematic user code and the error-related runtime data. *Rules for dataflow* can identify the skewed data and improper configurations.

We implemented *Mprof* in the latest Hadoop-1.2, and evaluated it on 28 real-world OOM errors in diverse Hadoop MapReduce applications, including raw MapReduce code, Apache Pig [6], Apache Hive [12], Apache Mahout [13], and Cloud[9] [14]. Twenty of them are reproducible OOM errors from our empirical study. Since only these 20 errors have detailed data characteristics, user code, and OOM stack trace, we reproduced them. Eight of them are new reproducible OOM errors collected from the Mahout/Hive/Pig JIRAs and open forums, which are not used in our empirical study. The results show that *Mprof* can precisely identify the root causes of 23 OOM errors, and partly identify the root causes of the other 5 OOM errors (in which the first OOM root cause is identified, but the second root cause is missed). *Mprof* is available at github [15].

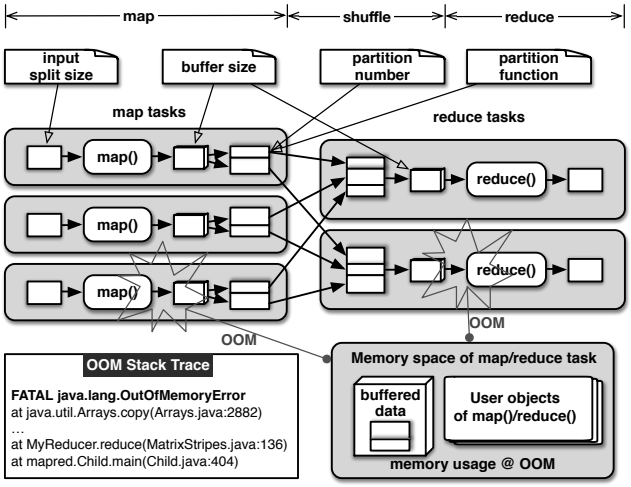

Figure 1: Two examples of OOM errors in a MapReduce application

The main contributions of this paper are as follows:

- An empirical study on 56 real-world OOM errors narrowed the root causes of OOM errors in MapReduce applications down to 3 kinds of common causes and 7 cause patterns.
- A memory profiling tool is designed to profile and quantify the correlation between a MapReduce application's runtime memory usage and its static information.
- Two types of quantitative rules (11 rules in total) are designed to diagnose OOM errors in MapReduce applications.
- An evaluation on 28 real-world OOM errors shows that *Mprof* can accurately identify the OOM root causes.

An earlier version of this work appeared at ISSRE 2015 [16]. In this paper, we significantly extend the earlier version in three aspects. (1) We designed and implemented a memory profiler that can automatically profile the memory usage of MapReduce applications. (2) We designed two types of quantitative rules in the profiler to identify the root causes of OOM errors. (3) We evaluated our profiler on 28 real-world MapReduce applications.

The rest of the paper is organized as follows. Section 2 introduces the background of MapReduce applications. Section 3 presents our empirical study results on the common root causes of OOM errors. Section 4 describes the design and implementation of *Mprof*. Section 5 describes the diagnosing procedure and diagnostic rules in *Mprof*. Section 6 presents the evaluation results. Section 7 discusses the limitation and generality of *Mprof*. Section 8 lists the related work and Section 9 concludes this paper.

2. Background

A MapReduce application can be generally represented as $\langle$*input dataset*, *configurations*, *user code*$\rangle$. Input dataset is split into data blocks (e.g., 3 input blocks in Figure 1) and stored on

the distributed file system (e.g., HDFS [17]). Before submitting an application to MapReduce framework, users need to write user code (e.g., $map()$) according to the programming model and specify the application's configurations. While running, a MapReduce application (job) is split into multiple map/reduce tasks, and each task runs as a separate process (a JVM instance). So, the memory usage of a MapReduce application denotes the usage of its map/reduce tasks. MapReduce's programming model, dataflow, and configurations are detailed below.

2.1. Programming model and user code

MapReduce programming model can be denoted as

$$\begin{aligned} \textit{Map stage}: \ & map(k_1, v_1) \Rightarrow list(k_2, v_2) \\ \textit{Reduce stage}: \ & reduce(k_2, list(v_2)) \Rightarrow list(k_3, v_3) \end{aligned}$$

In the map stage, $map(k,v)$ reads $\langle k_1, v_1 \rangle$ records one by one from an input block, processes each record, and outputs $\langle k_2, v_2 \rangle$ records. In the reduce stage, the framework groups the $\langle k_2, v_2 \rangle$ records into $\langle k_2, list(v_2) \rangle$ by the key k_2, and then launches $reduce(k, list(v))$ to process each $\langle k_2, list(v_2) \rangle$ group. For optimization, users can define a *mini reduce*() named *combine*(), which performs partial aggregation on map output records before *reduce*(). We regard *combine*() as *reduce*() since they usually share the same user code.

2.2. Dataflow

As shown in Figure 1, a typical MapReduce application contains multiple map/reduce tasks (i.e., mappers/reducers). Map tasks go through a map phase, while reduce tasks go through a shuffle and reduce phase. 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.

In the map phase, each mapper reads sequential $\langle k_1, v_1 \rangle$ records from an input split, performs $map()$ on each record, and outputs $\langle k_2, v_2 \rangle$ record with a partition *id* into a fixed buffer. Once the buffer is full, the buffered records will be sorted and merged onto the local disk. Records with the same partition *id* are stored in the same partition.

In the shuffle phase, each reducer fetches the corresponding partitions from the finished mappers, and temporarily stores them into a virtual buffer (x% of reducer's memory space where x is specified by users). Once the buffer is full, the buffered records are sorted and merged onto disk.

In the reduce phase, each reducer reads $\langle k_2, list(v_2) \rangle$ records from the merged records, performs *reduce*() on each record, and outputs $\langle k_3, v_3 \rangle$ records onto the distributed file system.

2.3. Configurations

A MapReduce 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.

3. Empirical Study on OOM Errors

To understand and summarize the common causes of OOM errors in MapReduce applications, we perform an empirical study on 56 real-world OOM errors[1].

We took real-world MapReduce applications that run atop Apache Hadoop as our study subjects. Since there are not any special bug repositories for OOM errors (JIRA mainly covers Hadoop framework bugs), users usually post their OOM errors on the open forums (e.g., StackOverflow.com and Hadoop mailing list). We in total found 632 issues by searching keywords such as "Hadoop out of memory" in StackOverflow.com, Hadoop mailing list [5], developers' blogs, and two MapReduce books [18, 19]. We manually reviewed each issue and only selected the issues that satisfy: (1) The issue is an OOM error. We excluded 459 issues that are not OOM errors (e.g., only contain partial keywords "Hadoop Memory"). (2) The OOM error occurs in a Hadoop application, not the framework's service components (e.g., the scheduler and resource manager). In total, 151 OOM errors are selected. These errors occur in diverse Hadoop applications, such as raw MapReduce code, Apache Pig [6], Apache Hive [12], Apache Mahout [13], Cloud9 [14] (a Hadoop toolkit for text processing).

For each OOM error, we manually reviewed the user's error description and the answers given by experts (e.g., Hadoop committers from cloudera.com and experienced developers from ebay.com). Out of the 151 OOM errors, the root causes of 56 errors (listed in Table 2) 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.

Table 2: Distribution of our studied OOM errors

Sources	Raw code	Pig	Hive	Mahout	Cloud9	Total
StackOverflow.com	20	4	2	4	0	30
Hadoop mailing list	5	5	1	0	1	12
Developers' blogs	2	1	0	0	0	3
MapReduce books	8	3	0	0	0	11
Total	**35**	**13**	**3**	**4**	**1**	**56**

Although the root causes of the 56 OOM errors are diverse, we can classify them into 3 categories and 7 cause patterns ac-

[1]The concrete OOM cases and the empirical study results are available at https://github.com/JerryLead/TR/blob/master/Hadoop-OOM-Study.pdf

Table 1: Cause patterns of OOM errors

Category	Cause patterns	Pattern description	Total	Ratio
Improper job configurations	Large framework buffer	Large intermediate data are temporarily stored in the framework buffer	6	10%
	Improper data partition	Some partitions are extremely large (small partition number, or unbalanced partition function)	3	5%
	Subtotal		**9**	**15%**
Data skew	Hotspot key	Large $\langle k, list(v)\rangle$ group	15	28%
	Large single key/value record	Large single $\langle k, v\rangle$ record	6	10%
	Subtotal		**21**	**38%**
Memory-consuming user code	Large external data	User code loads large external data	8	15%
	Large intermediate results	User code generates large intermediate computing results while processing a single $\langle k, v\rangle$ record	4(3)	7%
	Large accumulated results	User code accumulates large intermediate computing results in memory	30[13]	53%
	Subtotal		**42**	**75%**
Total			**56+16**	**128%**

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*. 128% means that 28% errors have two OOM cause patterns. Different with the original table in [16], this table is categorized according to the cause's relationship with configurations, runtime data, and user code.

cording to their relationship with the application's configurations, runtime data, and user code. Table 1 illustrates the concrete cause patterns and the corresponding number of errors in Hadoop applications. Most OOM errors (75%) are caused by memory-consuming user code. The second largest cause is data skew (38% errors). Note that 28% errors are caused by both memory-consuming user code and data skew. The left 15% errors are caused by improper job configurations. We next go through each OOM cause pattern and interpret how they lead to OOM errors.

3.1. Cause category: Improper job configurations

Users can adjust memory/dataflow-related configurations to optimize jobs' execution time and disk I/O. However, two types of improper configurations can lead to OOM errors.

3.1.1. Pattern 1: Large framework buffer

To lower disk I/O, data-parallel frameworks usually allocate in-memory buffers to temporarily store the intermediate data (output data of *map*() or input data of *reduce*()). 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. For example in shuffle phase, Hadoop allocates a virtual buffer named *shuffle buffer* to buffer the shuffled data. When users configure large buffer size, large intermediate data will be buffered in memory and OOM errors may occur.

This pattern has 6 OOM errors (10%). Four errors are caused by large map buffer (i.e., *io.sort.mb*). For example, a user configures a 300MB map buffer, but the mapper's memory space is only 200MB (e01)[2]. Two errors are caused by large shuffle buffer. For example, a user sets the shuffle buffer to be 70% of the reducer's memory space, which is too large (should be 30% in this case) and leads to the OOM error (e02).

[2](eXX) denotes the OOM error with ID=eXX in Table 10 in the Appendix.

3.1.2. Pattern 2: Improper data partition

Partition is a common technique used in data-parallel frameworks to achieve parallelism. In MapReduce applications, *map*() outputs its $\langle k, v\rangle$ records into different data partitions according to the k's partition *id*. For example in Figure 2, k_1, k_3, and k_5 exist in the same partition because they 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 intermediate data. For example in Figure 2, if the *aggregated partition* AP_1 is much larger than AP_2, the reducer that processes AP_1 will need to shuffle and buffer more data in memory. 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 OOM errors in user code. For example in Figure 2, if AP_1 is much larger than AP_2, *reduce*() may run out of memory while processing the large partition AP_1.

This pattern has 3 OOM errors (5%). Two errors are caused by small partition number. For example, a user reports that an OOM error constantly occurs in the reduce phase and his solution has been to increase the partition number (e03). The left one error is caused by the unbalanced partitions, where a very large number of items (key/value records) are sent to a single reducer (e04).

Finding 1: Fifteen percent of the OOM errors are caused by improper memory/dataflow-related configurations, such as large framework buffers and improper data partition, which can lead to large in-memory intermediate data.
Implication: Before running a MapReduce application, it is hard for users to set the right configurations to limit the application's runtime memory usage.

3.2. Cause category: Data skew

Since MapReduce framework processes the $\langle k, v \rangle$ records in a distributed fashion and the records may have a non-uniform distribution, the application may generate runtime skewed data such as large $\langle k, list(v) \rangle$ group (a result of *hotspot key*) and large single $\langle k, v \rangle$ record.

3.2.1. Pattern 3: Hotspot key

Although the $\langle k, v \rangle$ records in the same data partition will be processed by the same user code, they are first merged into different $\langle k, list(v) \rangle$ groups according to their keys. Then, user code (*reduce*() or *combine*()) will process the $\langle k, list(v) \rangle$ groups one by one. *Hotspot key* means that some $\langle k, list(v) \rangle$ groups are much larger (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 *aggregated partition* AP_1 in Figure 2, if $\langle k_5, list(v) \rangle$ is much larger than $\langle k_1, list(v) \rangle$, the framework may run out of memory while aggregating the $\langle k_5, list(v) \rangle$. Furthermore, the following *reduce*() may run out of memory while processing the large $\langle k_5, list(v) \rangle$ group.

This pattern has 15 OOM errors (28%), all of which are caused by the huge values associated with one key. For example, a user reports that some keys only return 1 or 2 items but some other keys return 100,000 items (e05). Another user reports the key $\langle$*custid, domain, level, device*$\rangle$ is significantly skewed, and about 42% of the records have the same key (e06).

3.2.2. Pattern 4: 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, 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 3.3.2. Since the record size is determined at runtime, dataflow-related configurations cannot control its size.

This pattern has 6 OOM errors (10%), 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*) (e07). Another user reports that some records are 1MB but some are 100MB non-splittable blob (e08). 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 (e09).

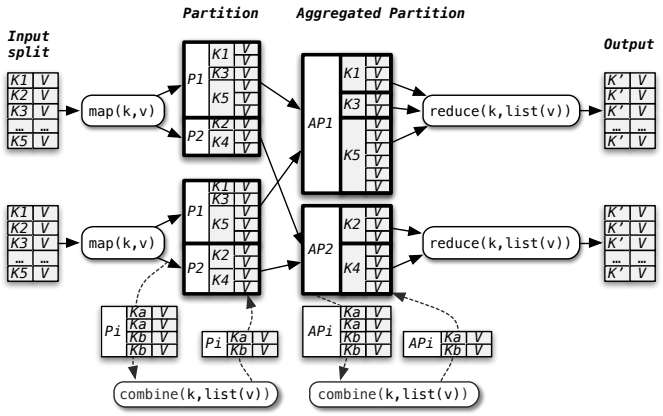

Figure 2: MapReduce dataflow

Finding 2: Data skew is another common cause of OOM errors (38%), which can lead to unexpected large runtime data, such as hotspot key and large single key/value record.
Implication: Framework's current data-parallel mechanisms do not properly consider the runtime data property (e.g., key distribution) and cannot limit the input data of user code.

3.3. Cause category: 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 results* (generated for a single record) and *large accumulated results*. Another pattern is that user code loads large external data in memory.

3.3.1. Pattern 5: Large external data

Different from the buffered 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 (15%). Three errors occur in Mahout applications, where mappers try to load large trained models for classification (e10) and for clustering (e11). One error occurs in a Hive application that tries to load a large external table (e12). One error occurs in a Pig script, in which the UDF (User Defined Function) tries to load a big file (e13).

3.3.2. Pattern 6: Large intermediate results

The 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 *Set*s, Cartesian product of them is orders of magnitude larger than this input record.

This pattern has 4 OOM errors (7%). In 2 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 (e14). The last error occurs in a text processing application (e15), which allocates large dynamic data structures during processing the input text.

3.3.3. Pattern 7: Large accumulated results

If the intermediate results generated at current input record are cached in memory for future use, they become accumulated 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 30 OOM errors (53%). 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 (e16). In other errors, users try to accumulate the intermediate results, such as the word's frequency of occurrence (e17) and the training weights (e18). User code accumulates the intermediate results to find distinct tuples (e19), perform in-memory sort, compute median value, or take a cross product (e20). The last error occurs in a reducer, which tries to keep the word co-occurrence matrix of a large document in memory (e21).

Finding 3: Most OOM errors (75%) are caused by memory-consuming user code, which carelessly processes unexpected large data or generates large in-memory 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.

4. Memory Profiler Design and Implementation

Our empirical study has narrowed the root causes of OOM errors down to 7 cause patterns. However, it is still hard to diagnose the root causes of OOM errors in a running MapReduce application. To diagnose the OOM errors, we design a memory profiling tool named *Mprof* as shown in Figure 3. *Mprof* can automatically profile and quantify the correlation between a MapReduce application's runtime memory usage and its static information (input data, configurations, and user code). *Mprof* achieves this through modeling and profiling the application's dataflow, memory usage of user code, and performing correlation analysis on them. *Mprof* only relies on tasks' enhanced

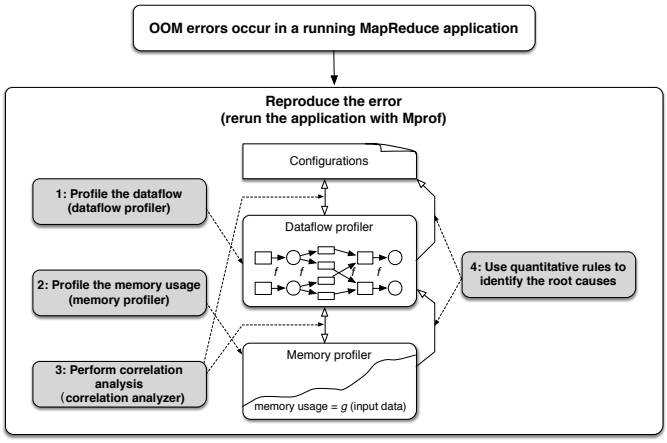

Figure 3: Mprof Overview

logs (with statistics of processed $\langle k, v\rangle$ records), dataflow counters, and heap dumps, without any modifications to the user code. Based on the correlation, *Mprof* uses quantitative rules to trace OOM errors back to the problematic user code, data, and configurations.

Mprof mainly contains four parts: (1) a *dataflow profiler* that models and profiles the application's dataflow; (2) a *user code memory profiler* that profiles the memory usage of user code; (3) a correlation analyzer that performs correlation analysis on the applicaiton's configurations, dataflow, and memory usage; (4) two types of quantitative rules (i.e., rules for user code and rules for dataflow) that identify the root causes of OOM errors, which will be detailed in Section 5.

4.1. Dataflow profiler

Dataflow profiler aims to profile the application's dataflow through building a dataflow model and fitting the model with runtime dataflow counters (extracted from tasks' enhanced logs and runtime dataflow monitors). The dataflow model quantifies the runtime data in each processing step, and also performs skew analysis (Section 5.3) for diagnosing OOM errors.

Dataflow model is built according to the fixed data dependencies in MapReduce dataflow. Although MapReduce dataflow contains many data processing steps, it is composed of three primitive steps: *map*(), *key aggregation*, and *reduce*(). According to the data dependencies in Figure 2, the three steps can be formulated by the functions shown in Table 3. $mapInputRecords_i$ represents the number of input records of mapper i, while $P_{ij}(k,v)$ represents the number of records in j-th partition outputted by mapper i. N_p is the *partition number*. The number of records in an *aggregated partition* AP_j is equal to $\sum_{i=1}^{N_m} P_{ij}(k,v)$, where N_m is the number of mappers. After *key aggregation*, the number of $\langle k, list(v)\rangle$ groups in *aggregated partition* AP_j is N_k, and the records in g-th $\langle k, list(v)\rangle$ group in AP_j is $G_{gj}(k, list(v))$. The final $reduceOutputRecord_j$ represents the number of records outputted by reducer j.

Dataflow-related configurations are involved in the model. For example, the total size of $mapInputRecords_i$ equals *input split size*. N_p equals *partition number*. *Partition function* is

Table 3: Dataflow model

Primitive steps	Input records	Output records
$map_i()$	$mapInputRecords_i$	$P_{ij}(k,v), 1 \le j \le N_p$
$AggrPartition_j$	$P_{ij}(k,v), 1 \le i \le N_m$	$G_{gj}(k,list(v)), 1 \le g \le N_k$
$reduce_j()$	$G_{gj}(k,list(v)), 1 \le g \le N_k$	$reduceOutputRecord_j$

* $combine()$ has the similar data dependency and functions with $reduce()$

Table 4: An example of map/reduce tasks' enhanced logs

M1	[***map***() starts]
	[***map***() ends] **InputRecords** = 17,535, **OutputRecords** = 17,525
	[***map output***][Partition 1] **Records** = 8,653, **Keys** = 3,943
	[***map output***][Partition 2] **Records** = 8,559, **Keys** = 3,715
R1	[***shuffling***] Partition (**Records** = 8,653, **Keys** = 3,943) from **mapper1**
	[***shuffling***] Partition (**Records** = 8,882, **Keys** = 4,036) from **mapper2**
	[***shuffling***] Partition (**Records** = 9,016, **Keys** = 4,087) from **mapper3**
	[***combine***() starts] to merge the buffered partitions from **mapper (1, 2)**
	[***combine***() ends] **OutputRecords** = 7,979, **Keys** = 7,979
	[***reduce***() starts] **InputRecords** = 16,995, **Keys** = 12,066

* The above counters are not available in current Hadoop tasks' logs.

modeled by computing the ratio of the records in each aggregated partition (AP).

To fit the dataflow model, dataflow profiler automatically extracts dataflow counters from tasks' enhanced logs (as shown in Table 4) and runtime dataflow monitors. Our enhanced logs can output (1) the processing steps, such as $map()$, shuffling, and $reduce()$; (2) the data source, such as "partition shuffled from mapper1"; (3) the statistics of *Records* and *Keys* in each *Partition* and in each user code. For example in Table 4, our profiler can compute the number of records in the *aggregated partition* AP_1 by adding the records in the three shuffled partitions (i.e., 8,653 + 8,882 + 9,016 = 26,551). Moreover, from the runtime dataflow monitor provided by Hadoop, our profiler can obtain real-time number of processed records/groups of user code, which is used to compute the records in each group (i.e., $G_{gj}(k,list(v))$).

4.2. User code memory profiler

User code memory profiler aims to figure out the correlation between memory usage of user code and its input data. A simple strategy is to continuously monitor the size of user objects (i.e., trying to obtain the gray curve in Figure 4)). However, this strategy is impractical. If we want to obtain the size of user objects in a task at time t, we need to dump the task's heap at that time, analyze which objects in the heap are user objects (i.e., referenced by user code), and then compute their total size. Since user code usually processes thousands and millions of input records, it is too time-consuming to dump the heaps and analyze them.

We observe that user objects defined in different locations in user code have different but fixed lifecycles. User objects with different lifecycles are related to different parts of the input data, and can be classified into *record-level intermediate results* or *accumulated results*. Based on this feature, we design a *lifecycle-aware memory monitoring strategy* that can use limited heap dumps to figure out the correlation between intermediate/accumulated results and their related input $\langle k,v \rangle$ records.

4.2.1. User code templates and object lifecycles

User code processes the input records in a streaming style. This feature results in the fixed lifecycles of user objects. We summarize these code templates from the real-world Hadoop MapReduce examples, and examples in MapReduce Design Patterns [18].

***Object lifecycles in map()*:** The programming model of $map(k,v)$ is shown below. $map(K,V)$ is invoked for each input $\langle k,v \rangle$ record, so objects generated in mapper have two lifecycles: (1) *map-level* (labeled ①): objects defined outside the method $map(K,V)$ can exist in memory until all the input records are processed. So, map-level buffers such as *ArrayList* can be used to store the intermediate results. (2) *record-level* (labeled ②): objects generated in the method $map(K,V)$ are regarded as record-level intermediate results (i.e., *iResults*), and will be cleared from memory when the next record comes in. Once these record-level intermediate results are cached in the map-level buffer, they become accumulated results.

```
public class Mapper {
  private Object mapLevelBuffer; // ①
  public void map(K key, V value) { // map(K,V)
    Object iResults = process(key, value); // ②
    emit(newKey, newValue); // using iResults
  }
}
```
Object lifecylces in map()

***Object lifecycles in reduce()*:** The programming model of $reduce()$ is a bit complex. The method $reduce(K,list(V))$ is invoked for each $\langle k,list(v) \rangle$ group and each $\langle k,v \rangle$ record in the group is processed one by one. Objects in reducer have three lifecycles: (1) *reduce-level* (labeled ③): objects defined outside the method $reduce(K,list(V))$ can exist in memory until all the groups are processed. So, accumulated results in reduce-level buffer are related to all the groups. (2) *group-level* (labeled ①): objects allocated in the method $reduce(K,list(V))$ but outside $while()$ can exist in memory until all the records in a group are processed. So, accumulated results in group-level buffer are related to all the records in the group. (3) *record-level* (labeled ②): objects generated in the $while()$ statement are record-level intermediate results and will be reclaimed when the next record in the $\langle k,list(v) \rangle$ group comes in. Similar to $map()$, if the intermediate results are cached by high-level (reduce/group-level) buffers, they become accumulated results. The programming model of $combine()$ is the same as that of $reduce()$.

```
public class Reducer {
  private Object reduceLevelbuffer; // ③

  public void reduce(K key, Iterable<V> values) {
    Object groupLevelBuffer; // ①
    // foreach <k,v> record in the group, like map(K,V)
    while (values.hasNext()) {
      V value = values.next();
      Object iResults = process(key, value); // ②
      emit(newKey, newValue); // may be here
    }
    emit(newKey, newValue); // may be here too
  }
}
```
Object lifecylces in reduce()

4.2.2. Lifecycle-aware memory monitoring strategy

At time t, user objects consist of accumulated results and record-level intermediate results. So, we can divide the problem (i.e., figure out the correlation between user objects and the input records) into two small problems as follows.

Profile the accumulated results: To profile the accumulated results, we choose to capture user objects at some specific time points. For map-level accumulated results, the candidate time points are when user code just finishes processing a record (R_i) and is going to process the next record (R_{i+1}). At that time, record-level intermediate results have been reclaimed, so the size of accumulated results is exactly the size of user objects. For reduce-level accumulated results, the candidate time points are when user code just finishes processing a group (G_i) and is going to process the next group (G_{i+1}). For efficiency, our strategy dumps the task's heap at intervals of records/groups (as shown in Figure 4, vertical dotted lines denote heap dumps). For map-level accumulated results, the record interval is $(n-1)/h$, where n is the number of processed records at time t, and h is the desired number of heap dumps specified by users. For group-level accumulated results (we only care about their sizes in the last group such as G_m in Figure 4), the monitoring strategy is as same as that in the $map()$. Finally, our profiler calculates the sizes of user objects from these heap dumps and draws the trend line of accumulated results.

Profile the record-level intermediate results: We only care about the intermediate results generated at current record R_n, so the time points to dump the heap are: *when* R_n is going to be processed and at the time of OOM. The size difference between the two heaps (i.e., ② in Figure 4) can be regarded as the size of record-level intermediate results. However, our strategy needs to further verify if this size is only related to R_n. The checking process is simple: our profiler reruns the user code, lets it only process R_n, and then calculates the size difference of user objects before and after R_n is processed. If this size difference is as same as ② (as shown in Figure 4), our profiler concludes that the intermediate results are only related to R_n.

We have implemented the above monitoring strategies in Hadoop through inserting the heap dump code into the framework's APIs, such as *RecordReader.nextKeyValue*() and *ValueIterator.next*(), which prepare the next input record/group for user code. Users do not need to modify their user code but to configure the number of desired heap dumps (7 is default). To perform the strategy, we need to rerun the job at the time of OOM. So, the heap dumps can be automatically generated.

4.3. Correlation analysis

After profiling the dataflow and memory usage of user code, *Mprof* performs correlation analysis on the application's configurations, dataflow, and memory usage. Dataflow model has quantified the correlation between configurations and dataflow. So, the next step is to quantify the correlation between dataflow and memory usage. The memory usage of buffered data is equal to *min*(*buffer size, intermediate data*), while the correlation between accumulated/intermediate results and their related input records are calculated by statistical methods (using linear/non-linear regression to fit the memory usage curve of intermediate and accumulated results separately).

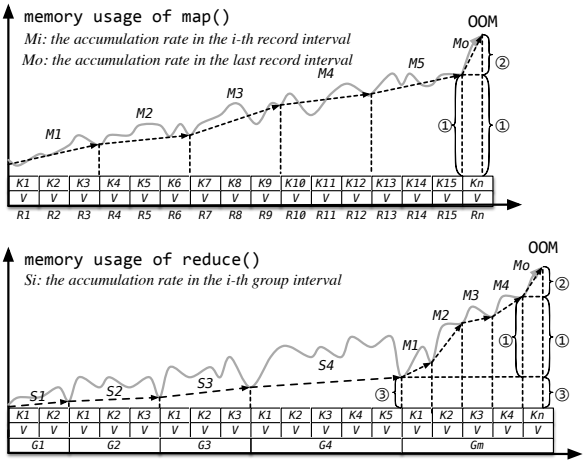

Figure 4: Lifecycle-aware memory monitoring. Slope M_i denotes the accumulation rate in the i-th record interval, while slope S_j denotes the accumulation rate in the j-th group interval. In addition, if user code loads external data, the user objects generated from external data are the objects at R_1 (for $map()$) or the objects at R_1 in G_1 (for $reduce()$).

Based on the correlation, *Mprof* can further identify the memory usage patterns of user code. The memory growth pattern of accumulated results can reflect the space complexity of user code. For example in $map()$ in Figure 4, linear growth in $[R_1, R_{15}]$ indicates that the space complexity of $map()$ is $O(n)$. So, users can easily figure out *why* their code is so memory-consuming and *which* ranges of input records are related to the OOM errors. Our profiler also performs outlier detection to figure out if the memory growths in some records are extremely large. For example in $reduce()$ in Figure 4, if the slope of memory growth in G_m is much sharper than that in other groups, the corresponding input records (records in G_m) may be abnormal (e.g., a result of *hotspot key*).

5. OOM Cause Identification

After figuring out the correlation among the application's configurations, dataflow, and memory usage, *Mprof* can further identify the root causes of OOM errors. *Mprof* first extracts large objects from heap dumps, and then uses quantitative rules to trace the large objects back to the problematic user code, data, and configurations. Figure 5 illustrates the procedure of OOM cause identification when an OOM error occurs in a reducer. The concrete steps are as follows.

5.1. Identify memory-consuming code snippets

This step first extracts large user objects from the OOM heap dump, and then identifies the objects' referenced code snippets. A heap dump is actually an object graph, in which each vertex represents an object and each edge denotes the object reference. To extract the large objects, the object graph is first transformed into a dominator tree [20]. Then, large (e.g., 3MB+) user objects (U) are extracted by identifying the objects' referenced methods. For example in Figure 5, if

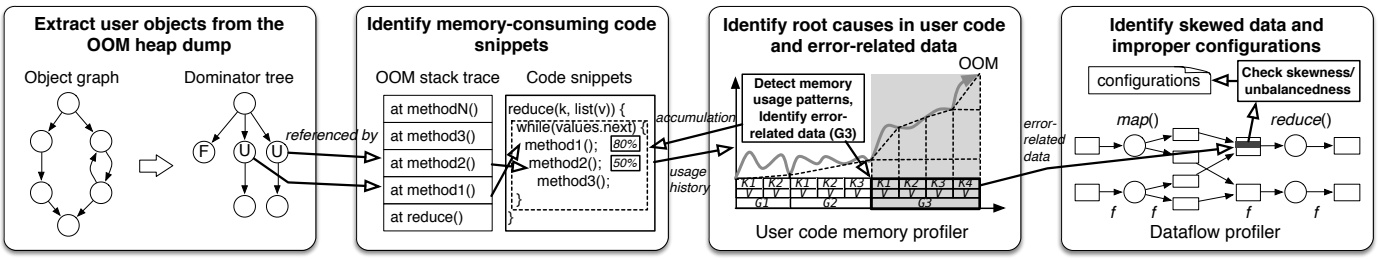

Figure 5: The procedure of cause identification when an OOM error occurs in *reduce*()

Table 5: Rules to identify the root causes in user code and the error-related data

ID	Symptoms	Root causes in user code	Error-related data	Next action
1	Continuous growth in $[R_1, R_n)$	Large map-level accumulated results	All the input records	Check *input split size*
2	Continuous growth in $[G_1, G_m)$	Large reduce-level accumulated results	All the input records	Check data partition (Rule 6)
3	Continuous growth in group G_m	Large group-level accumulated results	Records in group G_m	Check hotspot key (Rule 7)
4	Sharp growth at R_n	Large record-level intermediate results	Current record R_n	Check large single record (Rule 8)
5	Objects at R_1 are huge	Large external data	The external data	Output *Size*(*Objects at* R_1/G_1)

*** Sharp growth:** The slope of memory growth at R_n (i.e., M_o in Figure 4) is an *outlier* compared to the slopes of memory growth in the other records (e.g., $M_1, M_2, \ldots, M_5$ in Figure 4).
*** Objects at R_1 are huge:** The size of user objects at R_1 is huge (e.g., occupies 30% of the memory space).

an *ArrayList* is referenced by *method*2() and indirectly referenced by *reduce*(), it is a user object generated in *reduce*(). Memory-consuming code snippets (methods) are identified by calculating the total size of user objects referenced by each method in the OOM stack. For example in Figure 5, 80% (e.g., 80MB) user objects are referenced by *method*1() and 50% user objects are referenced by *method*2(). Listing 1 shows that a 411MB user object *wcMap:HashMap* is now referenced by *map*(), so the memory-consuming method is *Mapper.map*(*InMemWordCount.java:*49).

```
at java.util.HashMap.put(HashMap.java:372)
at mapper.InMemWordCount$Mapper.map(InMemWordCount.java:49)
   => wcMap:java.util.HashMap @ 0xdbc5e0c0 (430,875,440 B)
at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:203)
...
at org.apache.hadoop.mapred.Child.main(Child.java:404)
```

Listing 1: Memory-consuming user object and its referenced methods

The buffered data can also be extracted from the heap dump by identifying the objects' names. In Hadoop applications, *org.apache.hadoop.mapred.Merger.Segment* denotes a buffered partition and *kvbuffer* denotes the *map buffer*.

5.2. Identify root causes in user code and error-related data

To figure out *why* user objects become so large and *which* records are error-related, *Mprof* detects the memory usage patterns from the memory usage curve of accumulated and intermediate results. Suppose that an OOM error occurs at R_n in *map*() or at R_n in G_m in *reduce*(). If map/reduce/group-level objects demonstrate *continuous memory growth*, *Mprof* concludes that user code accumulates large intermediate results in memory and the error-related data are the objects' corresponding input records/groups (i.e., $[R_1, R_n)$, $[G_1, G_m)$, or $[R_1, R_n)$ in G_m). If the memory growth at R_n is a *sharp growth* (i.e., the slope of memory growth is an *outlier* compared to that in the other records), *Mprof* concludes that the user code generates *large record-level intermediate results* and the error-related data is current record R_n. *Continuous growth* includes *linear growth* and *non-linear growth* (detected by performing linear/non-linear regression). *Outlier* has a statistical definition in [21], where a value x_i larger than the *UpperInnerFence* of all the values $(x_1, x_2, \ldots, x_n)$ is regarded as an *outlier*. For example in Figure 6, x_{15} and x_{16} are detected as outliers, because their corresponding values (150 and 140) are larger than the *UpperInnerFence* (135) of all the values.

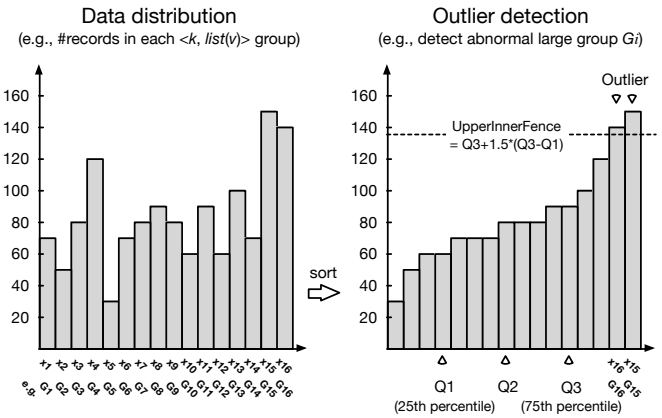

Figure 6: Outlier detection

Based on the detected memory usage patterns, we designed 5 statistical rules in Table 5 to identify the root causes in user code and the error-related data. Rule 1, 4, and 5 are applied in *map*(), while rule 2, 3, 4, and 5 are applied in *reduce*(). Once the symptoms of rule *r* are matched, rule *r* will be performed. For example in rule 3, the *continuous growth* of user objects in $[R_1, R_n)$ in G_m denotes *large group-level accumulated results*, which indicates that (1) the objects generated in the *while*()

Table 6: Skew measurement

Skew type	Description	Metrics
Partition skew	Partitions are unbalanced	$Gini(records\ in\ AP_1,\ records\ in\ AP_2,\ \ldots,\ records\ in\ AP_r) > 0.4$*
Group skew	G_m is an outlier group in AP_i	$Records\ in\ G_m \geq UpperInnerFence(records\ in\ G_1,\ records\ in\ G_2,\ \ldots,\ records\ in\ G_m)$⋄
Record skew	R_n is an outlier record in G_m	$R_n \geq UpperInnerFence(R_1,\ R_2,\ \ldots,\ R_n)$

* AP_i represents the number of records in *aggregated partition i*, while r denotes the reduce number.
⋄ $UpperInnerFence(x_1,x_2,\ldots,x_n) = Q3[x_1,x_2,\ldots,x_n] + 1.5 * IQ[x_1,x_2,\ldots,x_n]$, where $Q3$ is the third quartile of $[x_1,x_2,\ldots,x_n]$, and IQ is the interquartile of $[x_1,x_2,\ldots,x_n]$. G_j represents the j-th group in AP_i. R_j denotes the size of j-th record in group G_m.

Table 7: Rules to identify the skewed data and improper configurations

ID	Symptoms	Root causes	Improper configurations
6-1	Rule 2 is matched & partition skew is not matched	Improper data partition	Small *reduce number*
6-2	Rule 2 is matched & partition skew is matched	Improper data partition	Unbalanced *partition function*
7	Sharp growth in G_m & group skew is matched	Hotspot key	The *Key* design is bad
8	Rule 4 is matched & record skew is matched	Large single record	None
9-1	Fixed buffer exists & *map*() is nearly finished	Large framework buffer	Large *fixed buffer*
9-2	Virtual buffer is full & *combine*()/*reduce*() is nearly finished	Large framework buffer	Large *virtual buffer*

* **Sharp growth**: The slope of memory growth in G_m (i.e., S_m) is an *outlier* compared to the slopes of memory growth in the other groups (i.e., $S_1,\ldots,S_{m-1}$).
* **Nearly finished**: x% (e.g., 80%) of the input records have been processed. If we lower the buffer size, user code may finish successfully.

statement (as shown in Figure 5) are accumulated in group-level buffer; (2) the error-related data is the records in G_m (e.g., G_3 in Figure 5). The next action of rule 3 is to invoke rule 7 to check whether G_m is a result of *hotspot key*. For another example in rule 4, the *sharp growth* at current R_n indicates that user code generates *large intermediate results* during processing R_n. So, the next action is to invoke rule 8 to check whether R_n is a *large single record*. If the memory usage curve cannot be matched by any rules, our profiler will figure out the highest slope of memory growth (i.e., the largest S_i and M_i), and regard the corresponding input records as the error-related data.

5.3. Identify the skewed data and improper configurations

To identify *whether* the error-related data is a result of data skew and *which* configurations are problematic, *Mprof* performs skew measurement (listed in Table 6) on the dataflow, including partition skew, group skew (extremely large $\langle k, list(v)\rangle$ group), and record skew (extremely large record). *Gini index* is commonly used to measure the inequality among values of a frequency distribution (e.g., the income inequality). Here, it is used to measure the partition skew (i.e., the inequality of record distribution in all the partitions). To compute *gini index*, *Mprof* first sorts the partitions by their record numbers (#records). Then, *Mprof* plots the proportion of the total #records (y-axis) that is cumulatively earned by the bottom (smallest) x% of the partitions. For example in Figure 7, the left figure represents the sorted number of records in ten partitions, while the right figure demonstrates how to compute *gini index*. For each partition AP_i, its x-value represents the partition proportion ($i/\#partition$), while its y-value represents the record proportion (i.e., the ratio of cumulative records in $AP_{1\to i}$ to total records). *Gini index* is equal to A divided by $(A+B)$, which measures *how far* the record distribution is from uniform distribution. A *gini index* of zero denotes perfect equality. *Gini index* > 0.4 is well-recognized as inequality, so *Mprof* uses this inequation to determine partition skew. Outlier algorithm [21] is used to measure group skew (whether current G_m has much more records than the other groups) and record skew (whether current R_n is much larger than the other records in G_m). For example in Figure 6, if x-axis represents group G_i and y-axis represents the number of records in G_i, the detected outlier groups are G_{15} and G_{16}.

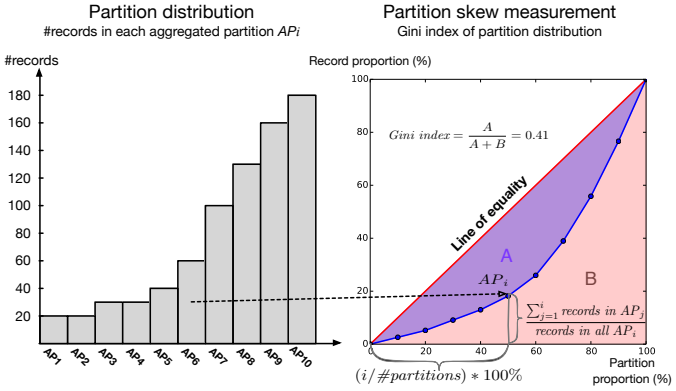

Figure 7: Partition skew measurement

Based on the skew analysis, we designed 4 statistical rules (6-1, 6-2, 7, and 8) in Table 7 to identify the skewed data and improper configurations. For example in rule 6-2, if the error-related data is the whole aggregated partition and partition skew is identified, the root cause is *unbalanced partition function*. In rule 7, if the memory growth in G_m is much steeper than that in the other groups (i.e., *sharp growth*) and G_m is an outlier (extremely large) group, *hotspot key* is identified. In addition, Rule 9-1 and 9-2 are designed to figure out whether the root causes are *large framework buffers*.

6. Evaluation

Our evaluation answers the following three questions:

- **RQ1: Can *Mprof* effectively diagnose the real-world OOM errors in MapReduce applications?** We reproduced 28 real-world OOM errors, and then performed *Mprof* on them to see whether the root causes can be correctly identified.
- **RQ2: How much overhead does *Mprof* add to the running jobs?** We rerun each job with and without *Mprof*, and regard their time difference as the overhead.
- **RQ3: How does *Mprof* trace OOM errors back to the problematic user code, skewed data, and improper configurations in real-world OOM errors?** We performed 3 case studies to demonstrate the diagnosis procedure.

6.1. Experimental setup

We reproduced 28 real-world OOM errors that occur in diverse Hadoop MapReduce applications (jobs), including raw MapReduce code, Apache Pig [6], Apache Hive [12], Apache Mahout [13], and Cloud[9] [14]. Twenty of the errors are selected from the OOM cases described in our empirical study. The selection criteria are that the error should be reproducible (i.e., with detailed data characteristics, user code, and OOM stack trace). Eight errors are new reproducible OOM errors collected from Mahout/Hive/Pig JIRAs and open forums after the empirical study was performed. Since we do not have the original dataset, we use public dataset (Wikipedia) and synthetic dataset (random text and a well-known benchmark [22] that can generate skewed data) instead. We made sure that the OOM stack traces of our reproduced jobs are the same as that reported by the users. All the jobs are conducted on a 11-node cluster using our enhanced Hadoop-1.2 [23], which supports enhanced logs and generating heap dumps at specified R_i and G_i. Each node has 16GB RAM and the heap size of each mapper/reducer is set to 1GB. The default job configurations are as follow: *input split size* is 256MB, *map buffer size* is 400MB, *shuffle buffer size* is 60%, *reduce number* is 10, and the *partition function* is *HashPartition*. If the job does not generate OOM errors under default configurations, we will increase the job's *input split size*, *map buffer size*, *shuffle buffer size*, and decrease its *reduce number*. Finally, we got 28 jobs that can generate OOM errors under specific configurations listed in Table 8.

For each job, we first run it as a normal job. When an OOM error occurs, we set the number of heap dumps to be 7 and rerun the job[3]. While running, *Mprof* profiles the job's dataflow using dataflow counters and profile the memory usage of user code using heap dumps. *Mprof* leverages Eclipse MAT [24] to extract user objects and the buffered data from each heap dump. After that, *Mprof* performs correlation analysis on the configurations, dataflow, and memory usage. Finally, *Mprof* uses the diagnostic rules to identify the root causes.

[3]In general, more heap dumps means more accurate the memory profiling will be. *Mprof* needs at least 3 heap dumps (before the first record is processed, before the last record is processed, and at the time of OOM). Here, we want to check if a small number can achieve both low overhead and effectiveness. According to our experience, we chose 7. How to choose the number of heap dumps is discussed in Section 7.

6.2. Overall results

Table 8 illustrates the error diagnosis results, including job names (with configurations and urls to the real-world OOM errors), matched rules, the causes identified by *Mprof*, and whether the identified causes are the root causes. **In 23 cases, the causes identified by *Mprof* are the same as the root causes. In the other 5 cases, the identified causes are partially the same as the root causes (i.e., the 1st root cause is identified but the 2nd root cause has not been identified).** The reason of partial correctness is that our current skew measurement (illustrated in Table 6) is incomplete. For example, in case *Wordcount-like*, both *large shuffle buffer* and *improper data partition* are the root causes. However, the second root cause is missing, because our skew measurement currently only detects partition skew but does not detect whether each partition is extremely large. In another case *ReduceMerge*, the root causes contain *hotspot key*, which means that the k's corresponding $list(v)$ size is an outlier compared with other k's $list(v)$ size. However, our skew measurement currently only detects the $list(v)$ length skew (i.e., the number of v in $list(v)$ is an outlier compared with others), not $list(v)$ size skew. More skew measurement will be added in future work. In addition, in 6 cases (labeled $\diamond$), the memory-consuming code snippets cannot be identified, because the user code is generated by high-level languages (e.g., SQL-like Pig/Hive script).

Above results indicate that *Mprof* can effectively identify the root causes of OOM errors in MapReduce applications (41 out of the 46 root causes are correctly identified). The cause patterns of problematic user code are all identified due to the lifecycle-aware memory monitoring strategy and memory usage pattern detection. The cause patterns of skewed data and improper configurations are partially identified due to the dataflow model, skew measurement, and statistical rules.

6.3. Performance overhead

Since *Mprof* relies on enhanced logs and heap dumps to profile the job's dataflow and memory usage of user code, it adds some overhead to the running jobs. The absolute *overhead* is defined as $ExecutionTime(job\ with\ Mprof) - ExecutionTime(job\ without\ Mprof)$. The last column in Table 8 lists *Mprof*'s overhead and the job's total execution time with *Mprof*. For example in case *NLPLemmatizer*, 135 means that the overhead is 135 seconds, which occupies 23% of the job's execution time (575 seconds). **The absolute overhead spans from 82 seconds to 231 seconds, and the mean is 137.4 seconds.** Since the overhead mainly comes form heap dumping and heap analysis, we can conclude that the average time of dumping and analyzing a single heap is 137.4/7 = 19.6 seconds (7 is the number of heap dumps). This time cost is linearly related to the heap size. For example, it takes 10-22 seconds to dump and analyze a 1GB heap, while it takes 6-8 seconds to dump and analyze a 300MB heap. **The relative overhead (i.e., the ratio of absolute overhead to the total execution time)**

Table 8: Results of OOM error diagnosis

Phase	Job name (confs)	Rules	Identified causes	Root causes?	Verification(FixMethods)	Overhead
Map	NLPLemmatizer (256M,400M,0.6,10,Hash)	4 8	Large intermediate results (282MB) Large single $\langle k,v\rangle$ record (50MB)	(1st) ✓ (2nd) ✓	Split R_n into small records	135 (23%) 575 (sec)
	MahoutSSVD (512M,400M,0.6,10,Hash)	4 8	Large intermediate results (326MB) Large single $\langle k,v\rangle$ record (110MB)	(1st) ✓ (2nd) ✓	Split R_n into small records	112 (17%) 661
	InMemWordCount (512M,400M,0.6,10,Hash)	1 1	Large map-level accumulated results (412MB) Large input split size (512MB)	(1st) ✓ (2nd) ✓	↓ Input split size, Use FixMethod(a)	121 (18%) 687
	MapSideAggregation (256M,400M,0.6,10,Hash)	1 1	Large map-level accumulated results (476MB) Large input split size (256MB)	(1st) ✓ ◇ (2nd) ✓	↓ Input split size, Use FixMethod(a)	95 (22%) 432
	GroupByOperator (512M,400M,0.6,10,Hash)	1 1	Large map-level accumulated results (498MB) Large input split size (512MB)	(1st) ✓ ◇ (2nd) ✓	↓ Input split size, Use FixMethod(a)	97 (19%) 501
	PigDistinctCount (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (358MB) Hotspot key ($G_8 > UpperInnerFence(G_1,\ldots,G_8)$)	(1st) ✓ ◇ (2nd) ✓	Use FixMethod (a)	82 (16%) 497
	CDHJob (256M,500M,0.6,10,Hash)	9-1	Large map buffer (500MB)	(1st) ✓	↓ map buffer size	113 (38%) 297
	MahoutBayes (256M,400M,0.6,10,Hash)	5	Large external data (2.8GB)	(1st) ✓	↓ external data (trained model)	87 (31%) 273
	MahoutConvertText (256M,400M,0.6,10,Hash)	5	Large external data (1.8GB)	(1st) ✓+	↓ external data (training data)	91 (31%) 292
	HashJoin (256M,400M,0.6,10,Hash)	5	Large external data (1.6GB)	(1st) ✓+	↓ external data (the first table)	114 (25%) 449
	HashSetJoin (256M,400M,0.6,10,Hash)	5	Large external data (1.1GB)	(1st) ✓	↓ external data (100K+ join keys)	87(14%) 620
Shuffle	ShuffleInMemory (256M,400M,0.6,5,Hash)	6-2	Improper data partition ($Gini(AP_1,\ldots,AP_5) > 0.4$)	(1st) ✓+	Use RangePartition	110 (38%) 285
	Wordcount-like (256M,400M,0.7,5,Hash)	9-2	Large shuffle buffer (70%) ~~Improper data partition ($Gini(AP_1,\ldots,AP_5) < 0.4$)~~	(1st) ✓ (2nd) missing	↓ shuffle buffer ↑ partition number	129 (47%) 272
	PigJoin (256M,400M,0.7,5,Hash)	9-2	Large shuffle buffer (70%) ~~Improper data partition ($Gini(AP_1,\ldots,AP_5) < 0.4$)~~	(1st) ✓ (2nd) missing	↓ shuffle buffer ↑ partition number	123 (27%) 455
	ShuffleError (256M,400M,0.7,10,Hash)	9-2	Large shuffle buffer (70%)	(1st) ✓	↓ shuffle buffer	104 (24%) 427
	NestedDISTINCT (256M,400M,0.6,10,Hash)	3	Large group-level accumulated results (437MB)	(1st) ✓+ ◇	Use FixMethod (b)	179 (31%) 583
	PigOrderLimit (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (402MB) Hotspot key ($G_{91} > UpperInnerFence(G_1,\ldots,G_{91})$)	(1st) ✓ ◇ (2nd) ✓	Use FixMethod (b)	194 (27%) 723
Reduce	CollaborativeFiltering (256M,400M,0.6,10,Hash)	5	Large external data (1.3GB)	(1st) ✓	↓ external data (100million+ items)	89 (23%) 385
	GraphPartitioner (256M,400M,0.6,10,Hash)	3	Large group-level accumulated results (433MB)	(1st) ✓	Use FixMethod (c)	205 (21%) 971
	FindFrequentValues (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (391MB) Hotspot key ($G_{31} > UpperInnerFence(G_1,\ldots,G_{31})$)	(1st) ✓ (2nd) ✓	Use FixMethod (b)	138 (11%) 1231
	ReduceMerge (256M,400M,0.6,10,Hash)	3	Large group-level accumulated results (372MB) ~~Hotspot key ($G_{12} < UpperInnerFence(G_1,\ldots,G_{12})$)~~	(1st) ✓ (2nd) missing	Use FixMethod (a), Shrink $\langle k,list(v)\rangle$ group	136 (9%) 1522
	PositionalIndexer (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (396MB) Hotspot key ($G_{25} > UpperInnerFence(G_1,\ldots,G_{25})$)	(1st) ✓ (2nd) ✓	Use FixMethod (a)	144 (17%) 853
	BuildInvertedIndex (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (415MB) Hotspot key ($G_{231} > UpperInnerFence(G_1,\ldots,G_{231})$)	(1st) ✓ (2nd) ✓	Use FixMethod (c)	231 (16%) 1478
	CooccurMatrix (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (372MB) Hotspot key ($G_{16} > UpperInnerFence(G_1,\ldots,G_{16})$)	(1st) ✓ (2nd) ✓	Use FixMethod (c)	155 (6%) 2532
	JoinLargeGroups (256M,400M,0.6,10,Hash)	3	Large group-level accumulated results (396MB) ~~Hotspot key ($G_{61} < UpperInnerFence(G_1,\ldots,G_{61})$)~~	(1st) ✓+ ◇ (2nd) missing	Use FixMethod (a), Shrink $\langle k,list(v)\rangle$ group	187 (10%) 1914
	CrawlDatum (256M,400M,0.6,10,Hash)	3	Large group-level accumulated results (421MB) ~~Hotspot key ($G_{319} < UpperInnerFence(G_1,\ldots,G_{319})$)~~	(1st) ✓ (2nd) missing	Use FixMethod (c) Shrink $\langle k,list(v)\rangle$ group	202 (21%) 953
	TraceDataAnalysis (256M,400M,0.6,10,Hash)	3 7	Large group-level accumulated results (516MB) Hotspot key ($G_{115} > UpperInnerFence(G_1,\ldots,G_{115})$)	(1st) ✓ (2nd) ✓	Use FixMethod (c) Shrink $\langle k,list(v)\rangle$ group	196 (11%) 1725
	PutSortReducer (256M,400M,0.6,10,Hash)	3 8	Large group-level accumulated results (658MB) Large intermediate results	(1st) ✓ (2nd) ✓	Use FixMethod (c)	171 (12%) 1386

*** (1st)** ✓ means that the first root cause has been correctly identified. **(2nd) missing** means that the second root cause has not been identified. ◇ means that the memory-consuming code snippets cannot be identified (because the user code is generated by high-level languages). **+** means that the root causes are not identified by the experts, but by indirect verification (i.e., trying the fix methods to verify if the OOM errors will disappear).

*** Shrink** $\langle k,list(v)\rangle$ **group:** Redesign the k to avoid *hotspot key*. For example, using composite key $\langle\langle k,k'\rangle,v\rangle$ instead of $\langle k,v\rangle$ may reduce the number of records in each group.

*** FixMethod: (a)** Change the accumulative operation to one-pass streaming operation. For example, to deduplicate the records in a group, user code does not need to accumulate the distinct records in a *Set*. Instead, it can utilize the framework's sort mechanism to sort the records (i.e., Secondary Sort [25]). Then, sequential comparison can be performed to find the distinct records. **(b)** Divide the accumulative operation into several memory-efficient lightweight operations. **(c)** Spill the accumulated results into disk periodically and then perform on-disk merge.

*** Overhead:** The first number (e.g., 135 seconds in *NLPLemmatizer*) denotes the absolute overhead, while the x% (e.g., 23%) represents the relative overhead. The second number (e.g., 575 seconds) denotes the job's total execution time with $Mprof$.

spans from 6% to 47%, and the mean is 21.6%. This range is wide because the overhead is not sensitive to the job's execution time (mainly depends on the number and sizes of heap dumps). If the total execution time is short, the relative overhead will be high. The overhead of enhanced logs can be omitted, since it only takes less than 1 second to count the dataflow and perform logging.

Above results show that *Mprof* adds an average of $19.6 * heapNumber$ seconds to the running jobs. We believe that this overhead is acceptable for a diagnostic tool. Since the dataflow profiler and user code memory profiler can be turned off in normal jobs, *Mprof* adds no overhead to the normal jobs if no OOM errors occur.

6.4. Case studies

The representative MapReduce application types contain text processing, data mining, SQL analysis, Web indexing, graph computing, etc. Since the number of cases used in this study is limited, our selection criteria try to cover (1) representative application types as many as possible; (2) different types of user code; (3) OOM errors that occur in *map*(), *reduce*(), and *combine*(). Based on these criteria, we selected three applications as shown in Table 9.

Table 9: Coverage of the selected cases

Application	Type	User code	OOM in
NLPLemmatizer	Text processing	Raw Java code	*map*()
CooccurMatrix	Data mining	Raw Java code	*reduce*()
PigDistinctCount	Big SQL	SQL-like script	*combine*()

The following case studies illustrate how *Mprof* figures out the correlation among configurations, dataflow, and memory usage. The studies also detail how *Mprof* uses quantitative rules to identify the root causes of real-world OOM errors.

6.4.1. Case study 1: NLPLemmatizer

This case aims to lemmatize the words in Wikipedia through invoking a third-party library named *StanfordLemmatizer*. This OOM error occurs in *Line* 5 in the following *map*(), while it is processing the 76*th* record (R_{76}). Each record denotes a *line* of the Wikipedia (its text has been compacted). From the code, we cannot directly identify the root causes, which can be *large accumulated results* kept in *slem*, *large intermediate results* generated at R_{76}, or both. After deducing the trend of map-level user objects (shown in Figure 8 (a)), our profiler identifies that there is *not* a linear/non-linear growth in $[R_1, R_{76})$, but a *sharp growth* at current record R_{76}. Our profiler further verifies that this *sharp growth* is only related to R_{76} (drawn as a red solid line). So, Rule 4 is matched and the root cause is that *map*() generates *large record-level intermediate results* during processing R_{76}. Our profiler also figures out that the *large intermediate results* consist of a 232MB *ArrayList* and a 50MB *String*, which are referenced by the memory-consuming method *slem.lemmatize*(). Then, Rule 4 invokes Rule 8 to identify if R_{76} is a *large single record*. Our profiler figures out that R_{76} is 50MB, which is much larger than the other records (the second largest record is 12MB). So, Rule 8 is matched and the second root cause is that R_{76} is a *large single record* (a super long line). Our identified causes are the same as the root causes identified by the expert (the author): For tagging each word in the *line*, *lemmatize*() allocates large temporary data structures that might be 3 orders of magnitude larger than the *line*. To fix this error, users can split the super long *line* into multiple small *lines* (records).

```
1  public class Mapper {
2    StanfordLemmatizer slem = new StanfordLemmatizer();
3    public void map(Long key, Text value) {
4      String line = value.toString();
5      for(String word: slem.lemmatize(line)) => 282 MB
6        emit(word, 1);
7    }
8  }                                          Case Study 1
```

6.4.2. Case study 2: CooccurMatrix

This case aims to compute the word co-occurrence matrix of Wikipedia. As shown in the following code, *reduce*() allocates a self-defined data structure named *OHMap* to aggregate the records in each $\langle k, list(v)\rangle$ group. Each *key* is a *word* and *value* is an *OHMap* that holds this word's neighboring words. The OOM error occurs in *Line* 6 of *reduce*(), while it is processing the 779,551*st* record in the 16*th* group (G_{16}). From the source code, we do not know the semantics of *plus*() and cannot identify the root causes. Our profiler identifies that there is *not* a linear/non-linear growth in $[G_1, G_{16})$, since the corresponding slopes of memory growth in $[G_1, G_{16})$ are 0, as shown in Figure 8 (b). However, there is a *linear growth* in group G_{16}, as shown in Figure 8 (c). So, Rule 3 is matched and the root cause is *large group-level accumulated results*. The group-level accumulated results consist of a 372MB *wordMap:OHMap* referenced by memory-consuming method *plus*() (i.e., *plus*() accumulates too many words in the group-level buffer *wordMap*). Then, Rule 3 invokes Rule 7 to check if the *too many words* are the result of *hotspot key*. Our profiler figures out that the number of records in group G_{16} has 6 times more records than the other groups and the *hotspot key* (*hotspot word*) is "of". In addition, there is a *sharp growth* at current record $R_{779,551}$. However, our profiler verifies that this growth (drawn as a red dotted line) is related to all the records in G_{16}, because the growth is only 30MB when *reduce*() only processes $R_{779,551}$. The keyword "Arrays.copyOf()" in this error's OOM stack trace indicates that this *sharp growth* is a result of *data structure expansion*. Data structures such as *ArrayList* and *HashMap* can expand 1.5 or 2 times of the original size when they are nearly full. To fix this error, users need to change this in-memory aggregation to on-disk merge/aggregation.

```
 1 public class Reducer {
 2   void reduce(Text key, Iterable<OHMap> values) {
 3     Iterator<OHMap> iter = values.iterator();
 4     OHMap wordMap = new OHMap();
 5     while (iter.hasNext()) { // for(V value: values)
 6       wordMap.plus(iter.next()); => wordMap:OHMap(372MB)
 7     }
 8     emit(key, wordMap);
 9   }
10 }                                          Case Study 2
```

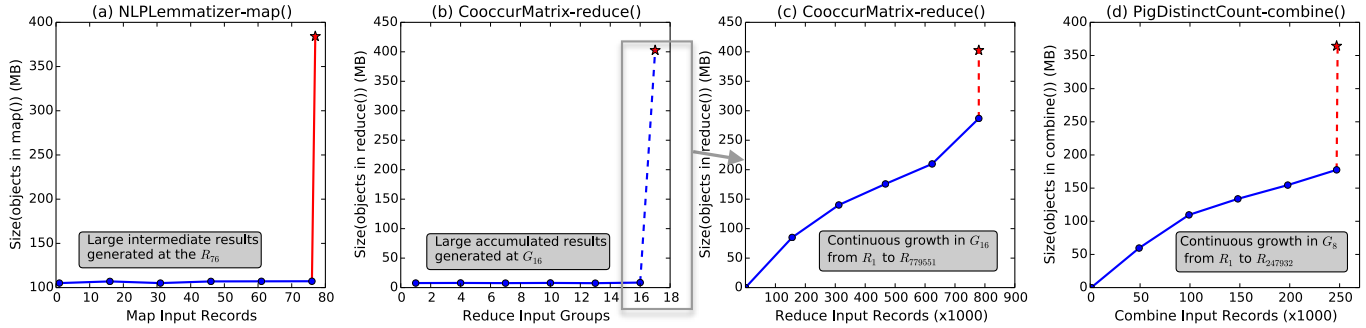

Figure 8: Quantified correlation between user objects and the input data of user code

6.4.3. Case study 3: PigDistinctCount

This job is automatically generated by a SQL-like Pig script, which tries to count the number of distinct values in column *pageurl* for each distinct *pagerank* in *tableA*. This script first aggregates the tuples using *GroupBy*(*pagerank*). Then, in each group, it uses *count*(*distinct pageurl*) to deduplicate the tuples that have the same value of *pageurl*. The stack trace shows that the OOM error occurs in *combine*(), while it is processing the 247,932*nd* record in 8*th* group (G_8). Since we cannot see the explicit *combine*(), we cannot identify the root causes.

```
1 pTable = LOAD tableA as (pagerank, pageurl, duration);
2 rankTable = GROUP pTable BY pagerank;
3 urlTable = FOREACH rankTable {
4   urls = DISTINCT urlTable.pageurl;
5   GENERATE group, COUNT(urls), SUM(pTable.duration);
6 };
7 STORE urlTable into "/output/newTable";    Case Study 3
```

Our profiler figures out that there is a *continuous growth* in group G_8 (as shown in Figure 8 (d)), and G_8 has much more records (247,932 records) than the other groups (the second largest group has 69,257 records). So, Rule 3 is matched and the root cause is that *combine*() generates *large group-level accumulated results* during processing 247,932 records in G_8 (i.e., a group-level buffer holds too many distinct tuples that have the same *pagerank*). Similar to Case 2, the *sharp growth* at $R_{247,932}$ is related to the all the records in G_8 and is the result of *data structure expansion*. Our profiler further figures out the *group-level accumulated results* contain a 281MB *pig.data.InternalDistinctBag* and a 77MB *pig.data.BinSedesTuple*, which are referenced by the memory-consuming method *PigCombine.processOnePackageOutput*(). To identify the *group-level buffer*, we refer to the Pig API and find the buffer is an *ArrayList* allocated by *InternalDistinctBag* for in-memory sort. To identify the high-level memory-consuming operator, we further refer to the Pig manual [26] and identify that the *DISTINCT* is the cause. This operator allocates a data structure (i.e., *InternalDistinctBag*) to accumulate all the input records to sort and deduplicate them. To fix this error, users can change the *key* to shrink the group or divide *DISTINCT* operator into multiple streaming operators.

The above case studies reveal that (1) *Mprof* can figure out the correlation between memory usage and dataflow through memory profiling and identifying the memory usage patterns. (2) *Mprof* can figure out the correlation between dataflow and configurations through dataflow profiling and skew measurement. (3) *Mprof* can use quantitative rules to identify the cause patterns. (4) *Mprof* can identify the memory-consuming code snippets if user code is written manually (i.e., not generated by high-level languages).

7. Limitation and Discussion

This section discusses the limitation of our empirical study, *Mprof*'s limitation, *Mprof*'s generality, how to select the number of heap dumps, and the potential ways to improve *Mporf*'s performance.

7.1. Limitation of our empirical study

Representativeness of applications Our empirical study only covers the applications that run atop open-source frameworks. Although many companies (e.g., Facebook and Yahoo!) use Hadoop frameworks, some other companies have built their own (e.g., Dryad in Microsoft). We have not studied the applications on these private frameworks. However, our studied cases have covered widely-used Hadoop applications, such as Apache Pig, Apache Hive, and Apache Mahout.

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.

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.

7.2. Limitation and discussion of Mprof

Limitation The main drawback of our profiler is that it requires a little manual effort. (1) For memory profiling, users need to rerun the failed job and set the number of heap dumps. (2) For cause identification, users need to link the identified cause patterns (e.g., large accumulated results, memory-consuming methods, and skewed data) with the code semantics, especially when user code is generated by high-level languages.

For example in the case *NLPLemmatizer*, our profiler can figure out the *large record-level intermediate results* contain an *ArrayList*, but it does not know *what* this data structure is used for. It is challenging to automatically understand user code semantics.

Generality Although our profiler is designed for MapReduce, it is useful for other big data frameworks such as Dryad [27] and Spark [28]. These MapReduce-like frameworks have flexible DAG-based dataflow, but their primitive data dependencies are the same as MapReduce dataflow. These frameworks support *pipelining*, but user code in their applications still processes the $\langle k, v \rangle$ records in a streaming style. So, the user objects still have fixed lifecycles. Moreover, job configurations such as *buffer size*, *partition number*, and *partition function* are also available in these frameworks. In addition, our profiler can also be used to diagnose the causes of high memory consumption.

Selection criteria of the number of heap dumps How to select the number of heap dumps (h) is a trade-off between the overhead and the accuracy of memory profiling. The evaluation shows that the average time of dumping and analyzing a single heap is 19.6 second and a small number (7) is still effective. So, users can choose this number or adjust the number according to their expected overhead ($19.6 * h$ seconds).

8. Related Work

Failure study on big data applications Many researchers have studied the failures in big data applications/systems. Li *et al.* [29] 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.* [30] analyzed 4100 failed Hadoop jobs, and found 36% failures are Array indexing errors and 23% failures are IOExceptions. Xiao *et al.* [31] 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.* [32] 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.* [33] studied 3655 development and deployment issues in cloud systems such as Hadoop and HBase [34]. 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 of OOM errors.

Memory leak detection Memory leak means that users forget to release the useless objects. Memory leaks can lead to OOM errors, so researchers have proposed many memory leak detectors. For example, Cork [10] uses object's type growth to identify the data structures that may contain useless objects. Container profiling [11] tracks the operations on containers to identify the unused data entries. However, these detectors cannot be directly used to diagnose OOM errors in MapReduce applications, since they cannot figure out the correlation between a MapReduce application's runtime memory usage and its static information. In our empirical 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.

Memory management optimization FACADE [35] 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 [36] 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. Our study shows that OOM errors can be caused by data skew, while SkewTune [37] provides a dynamic repartition mechanism for MapReduce framework. SkewTune can be potentially incorporated into our tool to optimize the memory consumption and fix the OOM errors caused by data skew.

9. Conclusion and Future Work

MapReduce applications frequently suffer from out of memory errors. In this paper, we performed a comprehensive study on 56 real-world OOM errors in MapReduce applications, and found the OOM root causes are improper job configurations, data skew, and memory-consuming user code. To diagnose OOM errors, we propose a memory profiler that can automatically figure out the correlation between a MapReduce application's runtime memory usage and its static information. Based on the correlation, our profiler uses quantitative rules to diagnose OOM errors. The evaluation shows that our profiler can effectively identify the root causes of OOM errors in real-world MapReduce applications.

In the future, we can improve our work in three aspects. (1) We further enhance our diagnostic rules to reduce the missing rate. (2) Our current profiler has a little heavy overhead, thus we further design lightweight memory profiling technique to reduce the overhead. (3) We plan to mitigate our approach to other big data frameworks (e.g., Spark), and evaluate our profiler on their applications.

10. Acknowledgements

This work was supported by the National Key Research and Development Plan (2016YFB1000803), National Natural Science Foundation of China (61672506), and Beijing Natural Science Foundation (4164104).

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] "Why the identity mapper can get out of memory?" [Online]. Available: http://stackoverflow.com/questions/12302708/why-the-identity-mapper-can-get-out-of-memory

[3] "java.lang.OutOfMemoryError on running Hadoop job." [Online]. Available: http://stackoverflow.com/questions/20247185/java-lang-outofmemoryerror-on-running-hadoop-job

[4] "Reducer's Heap out of memory." [Online]. Available: http://stackoverflow.com/questions/8705911/reducers-heap-out-of-memory

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

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

[7] "Eclipse Memory Analyzer," http://www.eclipse.org/mat.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] "Mprof: A Memory Profiler for Diagnosing Memory Problems in MapReduce Applications," https://github.com/JerryLead/Mprof.

[16] 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.

[17] "Hadoop Distributed File System." [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

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

[19] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[20] "Dominator tree." [Online]. Available: http://help.eclipse.org/mars/topic/org.eclipse.mat.ui.help/concepts/dominatortree.html?cp=44_2_2

[21] "What are outliers in the data?" [Online]. Available: http://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm

[22] 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.

[23] "Enhanced Hadoop-1.2." [Online]. Available: https://github.com/JerryLead/hadoop-1.2.0-enhanced

[24] "Enhanced Eclipse MAT." [Online]. Available: https://github.com/JerryLead/enhanced-Eclipse-MAT

[25] T. White, *Hadoop - The Definitive Guide*. O'Reilly Media, 2009.

[26] "DISTINCT operator in Pig Latin." [Online]. Available: http://pig.apache.org/docs/r0.13.0/basic.html#distinct

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. H. 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.

[36] 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.

[37] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012, pp. 25–36.

APPENDIX

Table 10: 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	Reducer's Heap out of memory	Pig 0.8	http://tinyurl.com/mftlvvv
e04	Reducers fail with OutOfMemoryError while copying Map outputs	Hadoop	http://tinyurl.com/j9k4oyw
e05	Building Inverted Index exceed the Java Heap Size	Hadoop	http://tinyurl.com/ojf9npb
e06	memoryjava.lang.OutOfMemoryError related with number of reducer?	Hadoop	http://tinyurl.com/q6jomja
e07	Hadoop Streaming Memory Usage	Hadoop	http://tinyurl.com/orfv3n3
e08	Hadoop Pipes: how to pass large data records to map/reduce tasks	Hadoop	http://tinyurl.com/phwdob4
e09	Hadoop Error: Java heap space	Hadoop 2.2	http://tinyurl.com/qy6wyj9
e10	OutOfMemory Error when running the wikipedia bayes example on mahout	Mahout	http://tinyurl.com/p3cj4ve
e11	Mahout on Elastic MapReduce: Java Heap Space	Mahout 0.6	http://tinyurl.com/na5wodj
e12	Hive: Whenever it fires a map reduce it gives me this error	Hive 0.10	http://tinyurl.com/p32aqfd
e13	OutOfMemoryError of PIG job (UDF loads big file)	Pig	http://tinyurl.com/ne6o6z3
e14	Writing a Hadoop Reducer which writes to a Stream	Hadoop	http://tinyurl.com/p46zupz
e15	java.lang.OutOfMemoryError on running Hadoop job	Hadoop 0.18.0	http://tinyurl.com/odydwfx
e16	Why does the last reducer stop with java heap error during merge step	Hadoop	http://tinyurl.com/crbd6q8
e17	MapReduce Algorithm - in Map Combining	Hadoop	http://tinyurl.com/ohcue2r
e18	how to solve reducer memory problem?	Hadoop	http://tinyurl.com/okq74kp
e19	java.lang.OutOfMemoryError while running Pig Job	Pig	http://tinyurl.com/ovpo8th
e20	A join operation using Hadoop MapReduce	Hadoop	http://tinyurl.com/b5m72hv
e21	Set number Reducer per machines	Cloud9	http://tinyurl.com/m4fo6wr