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 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. 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 (input data, configurations, user code). 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. 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)
38% errors are caused by data skew, which can lead to unexpected large runtime data, including large \((k, list(v))\) group and large single \((k, v)\) 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 CloudT [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].

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 lists the related work and Section 8 concludes this paper.

2. Background

A MapReduce application can be generally represented as \(<input \text{ dataset}, \text{ configurations, user code}>\). 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

- **Map stage**: \( map(k_1, v_1) \Rightarrow list(k_2, v_2) \)
- **Reduce stage**: \( reduce(k_2, list(v_2)) \Rightarrow list(k_3, v_3) \)

In the map stage, \( map(k, v) \) reads \( (k_1, v_1) \) records one by one from an input block, processes each record, and outputs \( (k_2, v_2) \) records. In the reduce stage, the framework groups the \( (k_2, v_2) \) records into \( (k_2, list(v_2)) \) by the key \( k_2 \), and then launches \( reduce(k, list(v)) \) to process each \( (k_2, list(v_2)) \) 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 \( (k_1, v_1) \) records from an input split, performs \( map() \) on each record, and outputs \( (k_2, v_2) \) 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 \( (k_2, list(v_2)) \) records from the merged records, performs \( reduce() \) on each record, and outputs \( (k_3, v_3) \) 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 \( (k,v) \) 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.

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>
<thead>
<tr>
<th>Sources</th>
<th>Raw code</th>
<th>Pig</th>
<th>Hive</th>
<th>Mahout</th>
<th>Cloud9</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackOverflow.com</td>
<td>20</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>Hadoop mailing list</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Developers’ blogs</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>MapReduce books</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td>Total</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>56</td>
</tr>
</tbody>
</table>

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

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

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 [15], this table is categorized according to the cause’s relationship with configurations, runtime data, and user code.

According 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\((\cdot)\) or input data of reduce\((\cdot)\)). 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\[\cdot\]). (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). 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).

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\((\cdot)\) outputs its \((k, v)\) records into different data partitions according to the k’s partition id. For example in Figure 2, \(k_1, k_3, k_5\) 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\((\cdot)\) or combine\((\cdot)\)). 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\((\cdot)\) 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 \((k,v)\) records in a distributed fashion and the records may have a non-uniform distribution, the application may generate runtime skewed data such as large \((k, list(v))\) group (a result of hotspot key) and large single \((k,v)\) record.

#### 3.2.1. Pattern 3: Hotspot key

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

### 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 \((k,v)\) 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 \((k,v)\)
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 results. So, more records are processed, more intermediate results may accumulate in memory for future use, they become accumulated.

This pattern has 3 OOM errors (53%). In 11 errors, users allocate large intermediate results due to the extremely large input record. In 1 error, reduce(e) 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.

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 logs (with statistics of processed ⟨k, v⟩ 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 application’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: represents the number of input records of mapper i, while P_j(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 \( \sum_{k=1}^{N_m} P_j(k,v) \), where N_m is the number of mappers. After key aggregation, the number of (k, list(v)) groups in aggregated partition AP_j is N_k, and the records in g-th (k, list(v)) 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_s equals input split size. N_p equals partition number. Partition function is
Table 3: Dataflow model

<table>
<thead>
<tr>
<th>Primitive steps</th>
<th>Input records</th>
<th>Output records</th>
</tr>
</thead>
<tbody>
<tr>
<td>map()</td>
<td>mapInputRecords</td>
<td>P_i(k,v), 1 ≤ j ≤ N_p</td>
</tr>
<tr>
<td>Aggregator</td>
<td>P_i(k,v), 1 ≤ j ≤ N_m</td>
<td>G_g(j,k,list(v)), 1 ≤ g ≤ N_k</td>
</tr>
<tr>
<td>reduce()</td>
<td>G_g(j,k,list(v)), 1 ≤ g ≤ N_k</td>
<td>reduceOutputRecord_g</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>MI</th>
<th>[map() starts]</th>
<th>[map() ends]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>InputRecords</td>
<td>OutputRecords</td>
</tr>
<tr>
<td>1</td>
<td>8,553</td>
<td>7,979</td>
</tr>
<tr>
<td>2</td>
<td>4,087</td>
<td>3,943</td>
</tr>
</tbody>
</table>

Object lifecycles in map(): The programming model of map(k,v) is shown below. map(K,V) is invoked for each input (k,v) 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.

```java
class Mapper {
    private Object iResults = process();
}
class Reducer {
    private Object intermediateBuffer = process();
}
```

Object lifecycles in reduce(): The programming model of reduce() is a bit complex. The method reduce(K,list(V)) is invoked for each (k,list(v)) group and each (k,v) record in the group is processed one by one. Objects in reduce 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 [4]): 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 [5]): objects generated in the while() statement are record-level intermediate results and will be reclaimed when the next record in the (k,list(v)) 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().

```java
public class Mapper {
    private Object mapLevelBuffer;
    public void map(K key, V value) {
        Object iResults = process(key, value);
        emit(key, iResults);
    }
}
public class Reducer {
    private Object reduceLevelBuffer;
    public void reduce(K key, Iterable<V> values) {
        Object groupLevelBuffer;
        for_each <k,v> record in the group, like map(K,V) {
            V value = values.next();
            Object iResults = process(key, value);
            emit(key, newValue);
        }
    }
}
```
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 5, the monitoring strategy is as same as that in the \textit{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., \( \odot \) 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 \( \odot \) (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 \texttt{RecordReader.nextKeyValue()} and \texttt{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, \textit{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 \( \min (\text{buffer size, intermediate data}) \). 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).

Based on the correlation, \textit{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 \textit{map}() in Figure 4 linear growth in \([R_1, R_{15}]\) indicates that the space complexity of \textit{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 \textit{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, \textit{Mprof} can further identify the root causes of OOM errors. \textit{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
Extract user objects from the OOM heap dump

Figure 5: The procedure of cause identification when an OOM error occurs in \textit{reduce}()

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

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

* Sharp growth: The slope of memory growth at \(R_m\) (i.e., \(M_m\) 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 \texttt{method2}() and indirectly referenced by \texttt{reduce}(), it is a user object generated in \texttt{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 \texttt{method1}() and 50% user objects are referenced by \texttt{method2}(). Listing 1 shows that a 411MB user object \texttt{wcMap:HashMap} is now referenced by \texttt{map()}, so the memory-consuming method is \texttt{Mapper.map(InMemWordCount.java:49)}.

\begin{verbatim}
 at java.util.HashMap.put(HashMap.java:372)
 at mapper.InMemWordCount1.Mapper.map(Mapper.java:104)
 \end{verbatim}

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.MergerSEGment 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 \(R_m\) in \texttt{map()} or at \(R_m\) in \(G_m\) in \texttt{reduce}(). If map/reduce/group-level objects demonstrate \textit{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., \([R_1, R_m]\), \([G_1, G_m]\), or \([R_1, R_m]\) in \(G_m\)). If the memory growth at \(R_m\) is a \textit{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 \(R_m\). \textit{Continuous growth} includes linear growth and non-linear growth (detected by performing linear/non-linear regression). \textit{Outlier} has a statistical definition in [21], where a value \(x_i\) larger than the Upper Inner Fence 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 Upper Inner Fence (135) of all the values.

\begin{figure}[h]
    \centering
    \includegraphics[width=\textwidth]{outlier_detection.png}
    \caption{Outlier detection}
    \end{figure}

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 \texttt{map()}, while rule 2, 3, and 4 are applied in \texttt{reduce}(). Once the symptoms of rule \(r\) are matched, rule \(r\) will be performed. For example in rule 3, the \textit{continuous growth} of user objects in \([R_1, R_m]\) in \(G_m\) denotes large group-level accumulated results, which indicates that (1) the objects generated in the \texttt{while()}
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 (k.list(y)) 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 APᵢ, 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 APᵢ 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 Gm has much more records than the other groups) and record skew (whether current Rᵢ is much larger than the other records in Gm). For example in Figure 6, if x-axis represents group Gᵢ and y-axis represents the number of records in Gᵢ, the detected outlier groups are G₁₅ and G₁₆.

Table 6: Skew measurement

<table>
<thead>
<tr>
<th>Skew type</th>
<th>Description</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition skew</td>
<td>Partitions are unbalanced</td>
<td>Gini(records in AP₁, records in AP₂, ..., records in APᵢ) &gt; 0.4*</td>
</tr>
<tr>
<td>Group skew</td>
<td>Ga is an outlier group in APᵢ</td>
<td>Records in Ga ≥ UpperInnerFence(records in G₁, records in G₂, ..., records in Gᵢ)</td>
</tr>
<tr>
<td>Record skew</td>
<td>Rᵢ is an outlier record in Gᵢ</td>
<td>Rᵢ ≥ UpperInnerFence(R₁, R₂, ..., Rᵢ)</td>
</tr>
</tbody>
</table>

* APᵢ represents the number of records in aggregated partition i, while r denotes the reduce number.  

Figures 4-6 and Figure 7: Partition skew measurement

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

* Sharp growth: The slope of memory growth in Gᵢ (i.e., Sᵢ) is an outlier compared to the slopes of memory growth in the other groups (i.e., S₁,...,Sᵢ₋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.

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 Gᵢ is much steeper than that in the other groups (i.e., sharp growth) and Gᵢ 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 [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 heap dumps at specified G and C. 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. 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.

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 size is an outlier compared with other k’s list size. However, our skew measurement currently only detects the list length skew (i.e., the number of v in list size is an outlier compared with others), not list size skew. More skew measurement will be added in future work. In addition, in 6 cases (labeled ⊙), 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

<table>
<thead>
<tr>
<th>Phase</th>
<th>Job name (confs)</th>
<th>Rules</th>
<th>Identified causes</th>
<th>Root causes?</th>
<th>Verification (FixMethods)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>NLPLemmatizer</td>
<td>4</td>
<td>Large intermediate results (282MB)</td>
<td>(1st) ✓</td>
<td>Split K into small records</td>
<td>135 (23%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>Large single (k, v) record (50MB)</td>
<td>(2nd) ✓</td>
<td>Split K into small records</td>
<td>575 (sec)</td>
</tr>
<tr>
<td></td>
<td>MAhoutSS.V1</td>
<td>4</td>
<td>Large intermediate results (120MB)</td>
<td>(1st) ✓</td>
<td>Split K into small records</td>
<td>112 (17%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>Large single (k, v) record (110MB)</td>
<td>(2nd) ✓</td>
<td>Split K into small records</td>
<td>661</td>
</tr>
<tr>
<td></td>
<td>InMmemWordCount</td>
<td>1</td>
<td>Large map-level accumulated results (412MB)</td>
<td>(1st) ✓</td>
<td>Input split size, Use FixMethod(a)</td>
<td>121 (18%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Large input split size (512MB)</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(a)</td>
<td>687</td>
</tr>
<tr>
<td></td>
<td>MapSideAggregation</td>
<td>1</td>
<td>Large map-level accumulated results (476MB)</td>
<td>(1st) ✓</td>
<td>Input split size, Use FixMethod(a)</td>
<td>95 (22%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Large input split size (256MB)</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(a)</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>GroupByOperator</td>
<td>1</td>
<td>Large map-level accumulated results (498MB)</td>
<td>(1st) ✓</td>
<td>Input split size, Use FixMethod(a)</td>
<td>97 (19%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Large input split size (512MB)</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(a)</td>
<td>501</td>
</tr>
<tr>
<td></td>
<td>PglDistinctCount1</td>
<td>3</td>
<td>Large group-level accumulated results (355MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(a)</td>
<td>82 (16%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(a)</td>
<td>497</td>
</tr>
<tr>
<td></td>
<td>CDHJob</td>
<td>9-1</td>
<td>Large map buffer (500MB)</td>
<td>(1st) ✓</td>
<td>Map buffer size</td>
<td>113 (38%)</td>
</tr>
<tr>
<td>Shuffle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>297</td>
</tr>
<tr>
<td></td>
<td>MAhoutKeys</td>
<td>5</td>
<td>Large external data (2.80G)</td>
<td>(1st) ✓</td>
<td>External data (trained model)</td>
<td>87 (31%)</td>
</tr>
<tr>
<td></td>
<td>MAhoutConvertFix</td>
<td>5</td>
<td>Large external data (1.80B)</td>
<td>(1st) ✓</td>
<td>External data (training data)</td>
<td>91 (31%)</td>
</tr>
<tr>
<td></td>
<td>HashJoin</td>
<td>5</td>
<td>Large external data (1.68G)</td>
<td>(1st) ✓</td>
<td>External data (the first table)</td>
<td>114 (25%)</td>
</tr>
<tr>
<td></td>
<td>HashNewJoin</td>
<td>5</td>
<td>Large external data (1.1GB)</td>
<td>(1st) ✓</td>
<td>External data (100K+ join keys)</td>
<td>87 (14%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>620</td>
</tr>
<tr>
<td></td>
<td>ShuffleInMemory</td>
<td>6-2</td>
<td>Improper data partition (Gini(AP1, ..., AP8) &gt; 0.4)</td>
<td>(1st) ✓</td>
<td>Use RangePartition</td>
<td>110 (38%)</td>
</tr>
<tr>
<td></td>
<td>Wordcount-like</td>
<td>9-2</td>
<td>Large shuffle buffer (70%)</td>
<td>(1st) ✓</td>
<td>Large shuffle buffer</td>
<td>219 (27%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Improper data partition (Gini(AP1, ..., AP8) &gt; 0.4)</td>
<td>(2nd) ✓</td>
<td>Large shuffle buffer</td>
<td>272 (27%)</td>
</tr>
<tr>
<td></td>
<td>PglJoin</td>
<td>9-2</td>
<td>Improper data partition (Gini(AP1, ..., AP8) &gt; 0.4)</td>
<td>(1st) ✓</td>
<td>Large shuffle buffer</td>
<td>219 (27%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2nd) ✓</td>
<td>Large shuffle buffer</td>
<td>272 (27%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ShuffleError</td>
<td>9-2</td>
<td>Large shuffle buffer (70%)</td>
<td>(1st) ✓</td>
<td>Large shuffle buffer</td>
<td>104 (24%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>427</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (437MB)</td>
<td>(1st) +</td>
<td>Use FixMethod(b)</td>
<td>179 (31%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(b)</td>
<td>583</td>
</tr>
<tr>
<td>Reduce</td>
<td>CollaborativeFiltering</td>
<td>5</td>
<td>Large external data (1.3GB)</td>
<td>(1st) ✓</td>
<td>External data</td>
<td>89 (23%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(100million+ items)</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (433MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c)</td>
<td>205 (21%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(b)</td>
<td>971</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (391MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(b)</td>
<td>138 (11%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(b)</td>
<td>1231</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (372MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c), Shrunk (k, list(v)) group</td>
<td>136 (9%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c), Shrunk (k, list(v)) group</td>
<td>1522</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (396MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(a)</td>
<td>144 (17%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(a)</td>
<td>853</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (415MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c)</td>
<td>231 (16%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c)</td>
<td>1478</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (372MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c)</td>
<td>155 (6%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c)</td>
<td>2532</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (396MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(a), Shrunk (k, list(v)) group</td>
<td>187 (10%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c)</td>
<td>1914</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (415MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c), Shrunk (k, list(v)) group</td>
<td>202 (21%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c), Shrunk (k, list(v)) group</td>
<td>953</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (516MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c), Shrunk (k, list(v)) group</td>
<td>196 (11%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>Hotspot key</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c), Shrunk (k, list(v)) group</td>
<td>1725</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Large group-level accumulated results (658MB)</td>
<td>(1st) ✓</td>
<td>Use FixMethod(c)</td>
<td>171 (12%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>Large intermediate results</td>
<td>(2nd) ✓</td>
<td>Use FixMethod(c)</td>
<td>1386</td>
</tr>
</tbody>
</table>

* (1st) ✓ means that the first root cause has been correctly identified. (2nd) ✓ means that the second root cause has 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 (k, list(v)) group:** Redesign the k to avoid hotspot key. For example, using composite key (⟨k, v⟩, r) instead of (k, v) 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 s% (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

<table>
<thead>
<tr>
<th>Application</th>
<th>Type</th>
<th>User code</th>
<th>OOM in</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPLemmatizer</td>
<td>Text processing</td>
<td>Raw Java code</td>
<td>map()</td>
</tr>
<tr>
<td>CooccurMatrix</td>
<td>Data mining</td>
<td>Raw Java code</td>
<td>reduce()</td>
</tr>
<tr>
<td>PigDistinctCount</td>
<td>Big SQL</td>
<td>SQL-like script</td>
<td>combine()</td>
</tr>
</tbody>
</table>

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: NPLemmatizer

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 76th record (R76). 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 R76, 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 [R1, R76], but a sharp growth at current record R76. Our profiler further verifies that this sharp growth is only related to R76 (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 R76. 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 R76 is a large single record. Our profiler figures out that R76 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 R76 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 (k, list(v)) 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 779,551st record in the 16th group (G16). 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 [G1, G16], since the corresponding slopes of memory growth in [G1, G16] are 0, as shown in Figure 8 (b). However, there is a linear growth in group G16, 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 two many words are the result of hotspot key. Our profiler figures out that the number of records in group G16 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 R779,551. However, our profiler verifies that this growth (drawn as a red dotted line) is related to all the records in G16, because the growth is only 30MB when reduce() only processes R779,551. 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, Iterator<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
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 247,932nd record in 8th group (G8). 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 G8 (as shown in Figure 8 (d)), and G8 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 G8 (i.e., a group-level buffer holds too many distinct tuples that have the same `pagerank`). Similar to Case 2, the `sharp growth` at R247,932 is related to the all the records in G8 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 `Mprof`’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 \((k,v)\) 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 \((19.6 \times h\) seconds).

8. Related Work

**Failure study on big data applications** Many researchers have studied the failures in big data applications/systems. Li et al. [31] 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


## APPENDIX

Table 10: Links of real-world OOM errors referred in this paper

<table>
<thead>
<tr>
<th>ID</th>
<th>The real-world OOM errors referred in this paper</th>
<th>Version Info</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>e01</td>
<td>Out of heap space errors on T7s</td>
<td>Hadoop 0.20.2</td>
<td><a href="http://tinyurl.com/p9pray">http://tinyurl.com/p9pray</a></td>
</tr>
<tr>
<td>e02</td>
<td>pig join gets OutOfMemoryError in reducer when mapred.job.shuffle.input.buffer.percent=0.70</td>
<td>Pig</td>
<td><a href="http://tinyurl.com/kb4zm0">http://tinyurl.com/kb4zm0</a></td>
</tr>
<tr>
<td>e03</td>
<td>Reducer's Heap out of memory</td>
<td>Pig 0.8</td>
<td><a href="http://tinyurl.com/mft1v">http://tinyurl.com/mft1v</a></td>
</tr>
<tr>
<td>e04</td>
<td>Reducers fail with OutOfMemoryError while copying Map outputs</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/jk4koy">http://tinyurl.com/jk4koy</a></td>
</tr>
<tr>
<td>e05</td>
<td>Building Inverted Index exceed the Java Heap Size</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/q99pnb">http://tinyurl.com/q99pnb</a></td>
</tr>
<tr>
<td>e06</td>
<td>memoryjava.lang.OutOfMemoryError related with number of reducer</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/kbqy3">http://tinyurl.com/kbqy3</a></td>
</tr>
<tr>
<td>e07</td>
<td>Hadoop Streaming Memory Usage</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/orlv3n">http://tinyurl.com/orlv3n</a></td>
</tr>
<tr>
<td>e08</td>
<td>Hadoop Pipes: how to pass large data records to mapreduce tasks</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/pjw6q8">http://tinyurl.com/pjw6q8</a></td>
</tr>
<tr>
<td>e09</td>
<td>Hadoop Error: Java heap space</td>
<td>Hadoop 2.2</td>
<td><a href="http://tinyurl.com/qy6wy">http://tinyurl.com/qy6wy</a></td>
</tr>
<tr>
<td>e10</td>
<td>OutOfMemory Error when running the wikipedia bayes example on mahout</td>
<td>Mahout</td>
<td><a href="http://tinyurl.com/p3cj4ve">http://tinyurl.com/p3cj4ve</a></td>
</tr>
<tr>
<td>e11</td>
<td>Mahout on Elastic MapReduce: Java Heap Space</td>
<td>Mahout 0.6</td>
<td><a href="http://tinyurl.com/na5wod">http://tinyurl.com/na5wod</a></td>
</tr>
<tr>
<td>e12</td>
<td>Hive: Whenever it tires a map reduce it gives me this error</td>
<td>Hive 0.10</td>
<td><a href="http://tinyurl.com/p32aq">http://tinyurl.com/p32aq</a></td>
</tr>
<tr>
<td>e13</td>
<td>OutOfMemoryError of PIG job (UDF loads big file)</td>
<td>Pig</td>
<td><a href="http://tinyurl.com/m6o63">http://tinyurl.com/m6o63</a></td>
</tr>
<tr>
<td>e14</td>
<td>Writing a Hadoop Reducer which writes to a Stream</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/p6b2z">http://tinyurl.com/p6b2z</a></td>
</tr>
<tr>
<td>e15</td>
<td>java.lang.OutOfMemoryError on running Hadoop job</td>
<td>Hadoop 0.18.0</td>
<td><a href="http://tinyurl.com/odly4w1">http://tinyurl.com/odly4w1</a></td>
</tr>
<tr>
<td>e16</td>
<td>Why does the last reducer stop with java heap error during merge step</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/ehr6z">http://tinyurl.com/ehr6z</a></td>
</tr>
<tr>
<td>e17</td>
<td>MapReduce Algorithm - in Map Combining</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/ohcue2">http://tinyurl.com/ohcue2</a></td>
</tr>
<tr>
<td>e18</td>
<td>how to solve reducer memory problem/</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/okq4k">http://tinyurl.com/okq4k</a></td>
</tr>
<tr>
<td>e19</td>
<td>java.lang.OutOfMemoryError while running Pig Job</td>
<td>Pig</td>
<td><a href="http://tinyurl.com/pvpo9">http://tinyurl.com/pvpo9</a></td>
</tr>
<tr>
<td>e20</td>
<td>A join operation using Hadoop MapReduce</td>
<td>Hadoop</td>
<td><a href="http://tinyurl.com/b9m2l">http://tinyurl.com/b9m2l</a></td>
</tr>
<tr>
<td>e21</td>
<td>Set number Reducer per machines</td>
<td>Cloud9</td>
<td><a href="http://tinyurl.com/m4fo6w">http://tinyurl.com/m4fo6w</a></td>
</tr>
</tbody>
</table>