International Journal of Scientific Research and Engineering Development-– Volume 3 Issue 3, May – June 2020

Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Handling Real-World Out of Memory Errors in Open-Source Framework Apache Spark

Ishita Agarwal*, Sharadadevi S Kaganurmath**

*(Department of Computer Science and Engineering, RV College of Engg., Bengaluru
ishitaagarwal.cs16@rvce.edu.in)

** (Department of Computer Science and Engineering, RV College of Engg., Bengaluru
sharadadeviks@rvce.edu.in)

--************************----------------------------

Abstract:

Apache Spark is a high-performance, distributed data processing engine framework engineered with the goal of outperforming disk-based engines like Apache Hadoop by implementing in-memory computing. It's usual to gather information in a many-to-many fashion, a stage referred to as the shuffle section, in distributed processing platforms. In Spark, several inefficiencies exist within the shuffle section that promise large performance enhancements once resolved. In data-intensive programs that execute on parallel and distributed frameworks, Out of memory (OOM) errors transpire repeatedly . It's difficult for users of Spark to precisely specify the origin and fix these OOM errors since the framework obscures the small print of concurrent execution. The paper tends to determine the holdups within the execution of the present style, and lookouts for prospects that solve the ascertained issues. The paper judges leads in terms of application level turnout via complete and distinguishing study on actual real-time OOM errors occurring day-to-day in data engines.

***Keywords* — Spark, in-memory computation, Resource Modeling, Memory Management, Task programming, MapReduce, out of memory, data processing.**

--************************----------------------------

I. INTRODUCTION

In recent times, distributed frameworks supporting Map-Reduce[1], like Hadoop and Spark, have emerged because of the characteristic information-parallel frameworks in demand to process large-scale data. Open-source implementations like Spark and Hadoop are now extensively used to develop data-intensive applications, like click-log mining, web-indexing, graph analysis and of course machine learning. These distributed general-purpose frameworks offer data scientists with implicit parallelism and conceal the complications of distributed implementation. This approach helps to zero in on the information process logic, however mixes up the error recognition once the user's data-parallel tasks spawn runtime errors[2]. A continual software error in these parallely running applications is Out of Memory (OOM).

Unlike most runtime errors such as cluster failures, OOM errors are caused by unrestrained memory consumption. OOM errors will directly result in the task failure, and can't be endured by framework's defense mechanisms like repetitive execution of the unsuccessful Map and Reduce tasks. When executing a data-intensive concurrent application, the Spark system buffers reserve the transitional information in buffer storage for higher performance, while the developer code stores intercessor computing leading to high memory consumption. Within the task once the computed results and in-memory information surpass the memory boundations of the system memory error occurs. The framework solely throws the OOM stack trace that cannot forthrightly mirror the foundation cause.

Sadly little or no work exists that examines OOM errors in data-intensive parallel applications. Information concentrated systems failures are commonly assumed to be caused by unspecified columns, incorrect schemas, invalid row format, and outlawed parameters, while not inspecting the prevailing OOM errors in these applications running on nodes. Studies found that unreasonable Garbage-Collection(GC)[3] activities along with Out-of-memory(OOM) errors are commonplace problems in parallel programs, that cause degraded performance and value. To assist users, an exclusive investigation on OOM errors was collected for this paper from open forums[4],[5] enlisting known causes, fixes and how to avoid these serious OOM errors. To see and perceive and determine memory management weakness, we also tried to coerce errors in the system[6].

Categorically, the paper proposes to acknowledge the subsequent analysis queries:

• The elemental reasons behind OOM errors in distributed data-distributed programmes. Are there existing patterns?

• What do programmers do to rectify OOM errors? There exist any obvious fixes?

• Prospective solutions to boost the identification process and make subsystems more spontaneous in handling errors[7].

The research gives a generic framework to tackle the memory issues of unrestrained data applications. Unlike previous models that employ a set pattern to resolve the resource demand, the paper tends to use various linear and nonlinear mathematical algorithms to model the necessities of executor and core demands. The central plan is to deploy collective models where every archetype is specialised for responding to a specific code, rather than employing a single monolithic model.

II. BACKGROUND

A. *Apache Spark*

Apache Spark is an open-source distributed cluster computing framework[8] with an interface for spawning clusters with inherent parallelism and error handling and scope for input in Java, Python, R and Scala. Exceeding 2,000 contributors in 2016, Spark is a very dynamic open-source unified analytics data engine for large-scale processing of big-data. Every Spark application operates a dissociated set of executors with committed memory for performing parallel jobs within the application as shown in Fig. 1. These nodes are correlated by a coordinating node with the driver program running. The Dataset for Spark is collected in a distributed file system and assembled as Resilient Distributed Datasets(RDDs)[9]. Each Spark executor for caching RDDs assigns its own heap memory space. Spark profits from the parallelization performed by RDDs that describes the application's memory conduct instead of squandering computing cycles.

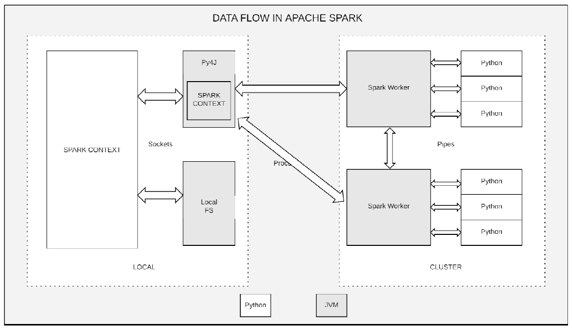

Figure 1. Parallel framework structure of Spark Application

B. *Distributed data-parallel application*

Any big data application can be illustrated as submitted data, algorithm structure and queries to execute this as shown in the Map-Reduce model[10] in Fig.2 . The input data is fed into data blocks and stocked in a file system like HDFS. Users need to specify beforehand the queries and configurations necessary to perform the parallel computations. Spark users are prescribed to write a driver program to execute the application. The driver program will also do the task of broadcasting data to each node and collecting when completed.

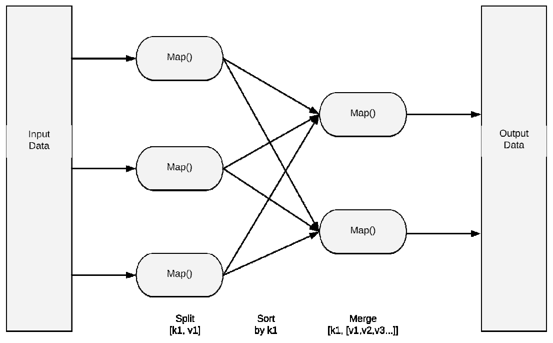

Figure 2: Map-Reduce model in distributed architecture

The application's organization comprises of following associations:

1. ***Memory-associated:*** Alignments that influence the memory usage directly. For example, the framework buffer size and memory limit stipulates the heap space of map/reduce tasks.
2. ***Dataflow-associated***: Configurations influence the bulk of the information that transcends between Mappers and Reducers. Partition() function for example designates segregation of the result of Map() function - hk and vi records , while the PartitionNumber() outputs generated partitions and launched partitions.[11]

C. Root Causes of Memory Problems

Memory errors are a very frequent matter with Spark applications which may be due to various reasons. Some of the most standard reasons are high concurrency, inefficient queries, and incorrect configuration[12]. Spark jobs or queries are broken down into multiple stages, and they are further divided into tasks. Spark demands some data structures and bookkeeping to reserve that much data. Techniques like dictionary encoding have some state saved in memory and all of them mandate memory resources. So with more concurrency, the overhead increases. For instance, if a broadcast join is involved, then the broadcast variables will also take some memory. These examples are picked from various websites, blogs, and popular mailing-lists and books. The condensed categories and cause patterns are described in the following table.

TABLE I
OOM Cause Patterns

Category	**Pattern**	**Pattern description**
Large data stored in framework	Large buffered data	Large intermediate data stored in buffers
	Large cached data	Large data cached in memory for reuse
Abnormal dataflow	Improper data partition	Some data partitions are extremely large
	Hotspot key	Large (k, list(v))
	Large single record	Large (k, v)
Memory consuming user code	Large external data	User code loaded large external data
	Large intermediate results	Large computing results are generated during processing a record
	Large accumulated results	Large computing results are accumulated in user code
	Large data generated in driver	The driver generates large data in memory
	Large results collected by driver	The driver collects large results of the tasks

III. METHODOLOGY

There are cases when an executor while executing fews tasks of a job, experiences memory OOM exceptions. There are many reasons for the OOM to happen, ranging from buggy application code (which allocates a lot of memory on random), to tasks handling skewed data sets, to spark scheduling memory heavy tasks on the same executor. Once the OOM happens, the tasks in execution are lost and all the running tasks must be rerun on some other executor. The paper will cover the aspects involved in the handling and learning OOM of executors due to the scheduling policy of the driver. The engaging as well as demanding part of the memory allocation pattern in Spark is to identify them. The paper tries to understand the behaviour once OOM happens, rather than proactively monitoring the tasks and the memory usage. Though a full blown solution for this problem has many dimensions to address, we will start with a very small part, where the scheduler understands the task's memory requirement once OOM happens. As a starting point, a new characteristic, named *coresNeeded* and *isTaint* are added to the task. The default values are *spark.task.cpus* and *false*, respectively.

A. *Identify the OOM type*

Memory allocation during the execution of the tasks in an executor could be either done by the task logic itself or by the processes spawned by the tasks (like shell process, python process).

1. ***Memory allocation by tasks:*** In case a task by itself allocates a lot of memory, and the overall memory used by JVM, generally specified by -Xmx, exceeds the JVM kills itself and the process terminates. The yarn when it knows about the process exit, terms it to be a "exitCausedByApp" and notifies the driver.

2. ***Memory allocation by process***: Tasks spawn processes like shell process, python process etc as mentioned in the task's body. These spawned processes could also allocate and consume a lot of memory. In this case, the YARN, which monitors the memory usage of the executor and its spawned processes, kills the executor and all its child processes. In this case the YARN informs the driver with error codes like KILLED_EXCEEDED_VMEM and KILLED_EXCEEDED_PMEM .

A new error type *ExecutorExited* is added for converting *executorLossReason* which indicates one of the above two scenarios.

B. *Driver Actions:*

The driver upon receiving the *ExecutorExited* error, assumes all the currently running tasks in the executor as memory intensive, and marks them as "taint". We can improve memory usage of tasks by trying to know the memory usage of each task, but we don't have the infrastructure as of now to get run time memory usage of tasks. The driver converts the memory needs of the tasks into scheduler friendly units, by increasing the tasks' coresNeeded to a higher value. We can increase the coresNeeded by the task in some arithmetic series like 1,2,3... or 1,2,4,8,.... or 1,3,6,.... .The solution code sets the value to *spark.execution.cores* which indicates that the task will grab the complete CPU resource of the executor, implying, no other task will run along with it, implying the complete memory of the executor is at the task's disposal.

C. *Scheduler Actions:*

The scheduler, as usual tries to pack an executor with the tasks that can fit the available cores of the executor. However, now, the scheduler will take into account the per task's CPU need and then schedule accordingly. It implies that when taint tasks are taken up for scheduling, the number of tasks that gets scheduled will be lesser than the number of tasks without taint. Also when the task completes, the freeCores is increased by the coresNeeded.

D. *Learning from taint tasks in a stage*

When a stage has many tasks tainting, it might be wise to bump up the whole stage's CPU need, i.e., we increase the *coresNeeded* for the rest of the tasks in the stage to be executed. This might come in a scenario where a stage is handling a staggering inflow of data where the overall memory requirement of most of the tasks is quite high. Thus, for each *TaskSet*, we introduce a variable to hold the minimum *coresNeeded* by the task. Thus, when a task is considered for scheduling, the *coresNeeded* by the task will be max.

IV. FUTURE WORK

As the runtime errors due to OOM can't be handled by current error evading techniques, it's important to style new mechanisms that can vanquish errors using OOM identification tools. Learned from the exponential method to handle executor loss and our exercise within the OOM identification, there is a tendency to find many potential solutions which will expedite the identification of cause and enhance the architecture's accountability.

A. Assist OOM source identification

Abnormal information flow may be a routine reason for OOM, however present frameworks offer restricted data-flow during runtime. OOM error identification can become easier if the frameworks will dispense statistical information flow. The data flow is derived from ordinary statistical functions (like min(), max(), average(), quartiles() and median()) on information partition and the input/output records in each cluster. Supporting this data, the frameworks will conduct some irregularity detection on the dataflow, like sleuthing the biases of the information segregation and mistreatment with applied mathematics ways.[13]

B. Upgrade framework's fault tolerance

1. Sanction dynamic memory management:

We frequently notice that OOM errors are spawned thanks to the massive information held within the frameworks. Although it's laborious to evaluate the memory consumption of a user programme, the framework will persistently track the memory utilization of the buffered or cached information along with the whole memory usage. The memory consumption for developer snippet is finally estimated by their difference.[14]

2. Avoid in-memory data structures:

Memory based information structures like List/ArrayList, Map/HashMap, Set/HashSet, and PriorityQueue are error prone[14]. A classic response is to use memory conjoined with disk information structures for acquisition when assigned to users. These novel information structures will have the original group of APIs built with C++ and Java. The merit is that they will tally the memory and mechanically swap between memory and disk.

V. CONCLUSION

The paper confers an all-inclusive study on known reasons and fixes for Out of Memory errors in application frameworks that are data-intensive and run in parallel manner like Apache Spark. The foundational causes of OOM relate to memory expending queries, deviant dataflow, and temporarily stored computations. The paper moreover mentioned common resolutions for fundamental runtime errors under OOM. The paper's target was to propound solutions and implement them to enhance framework's fault sufferance and expedite the OOM identification and tackling. The results will facilitate each user and therefore the framework architects to workaround OOM errors cohesively. Spark applications are getting popular as they drastically boost application throughput and system turnaround-time[15]. Using the mathematical routine of handling cores calculated by this framework's modification, a task-scheduler on a cluster smoothly transmits concurrent programmes. The methodology revamps the application's throughput and simultaneously ensures that memory consumed in total does not overshoot the allocated memory limit of the host.

ACKNOWLEDGEMENT

We thank our reviewers, Krupa Jain and Sheena Garg, for their indispensable comments and suggestions. We express sincere gratitude for Qubole Inc. and Mr. Vikram Agarwal for providing this opportunity and appreciation towards this project. We thank all the teaching staff and technical staff of the Computer Science and Engineering department, RVCE for their help and guidance.

REFERENCES

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

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

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

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

5. "Spark mailing list", [online] Available: http://apache-spark-user-list.1001560.n3.nabble.com/. 6. "Real-world OOM Errors in Distributed Data-parallel Applications", [online] Available: https://github.com/JerryLead/MyPaper/blob/master/OOM-Study.pdf.

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

8. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10). 10–10.

9. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 15-28, 2012.

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

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

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

13. H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, T. Qin, "An empirical study on quality issues of production big data platform", 37th International Conference on Software Engineering (ICSE), 2015.

14. L. Xu et al., "Experience report: A characteristic study on out of memory errors in distributed data-parallel applications," 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, MD, 2015, pp. 518-529.

15. 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.