Toward Elastic Memory Management for Cloud Data Analytics

Jingjing Wang and Magdalena Balazinska
Dept. of Computer Science & Engineering, University of Washington
{jwang,magda}@cs.washington.edu

Abstract

We present several key elements towards elastic memory management in modern big data systems. The goal of our approach is to avoid out-of-memory failures without over-provisioning but also to avoid garbage-collection overheads when possible.

1. INTRODUCTION

Recently, big data analytics systems have been emerging rapidly. These systems are often deployed in a cluster shared by multiple systems and users. Within each data analytics system, there can also be multiple analytics applications (*a.k.a.*, queries) running simultaneously. Different applications, systems, and users have different resource requirements. As a result, cluster-wide resource management is an important problem.

Among all types of resources, memory has become especially critical in recent years. There are two main reasons for that: (1) Modern data analytics systems, such as Spark [30], Impala [15], GraphLab [18], Giraph [23], and many more, strive to achieve high performance by maximally utilizing memory. (2) Data scientists have increasingly moved to using public clouds, where they pay directly for the resources that they need, and thus care to optimize them.

Different systems manage memory allocation in different ways: Some rely on the operating system to manage memory [18]. Others use a resource manager, which monitors the resource usage of the cluster and schedules applications accordingly: Some of those systems [11, 23] use external managers such as YARN [27] while others [15, 30] implement built-in resource managers but can also be scheduled by external ones.

The two key goals of resource scheduling are sharing and isolation. A naïve approach is to simply let all applications compete for resources as needed. This approach shares resources in the most flexible way, but would cause applications to interfere with each other thus complicating resource and fault isolation. Most contemporary analytics systems thus pick the alternate design. To enable a high degree of isolation between applications, they put applications inside containers with hard resource limits such that application resource requirements are both protected and constrained by the container. When an application needs to run, it must estimate its resource needs and communicate them to the resource manager. The latter then decides whether or not to schedule the application based on the amount of available resources. This is the most common approach used by contemporary systems with resource control, such as Spark [30], Impala [15] and Naiad [19].

However, it is very hard to estimate the memory needs of a data analytics application beforehand, which leads to out-of-memory failures, performance degradation due to disk-spilling, tedious trial-and-error, or wasteful over-provisioning. These challenges arise in both systems that self-manage memory, either C/C++ implementations [15, 18] or Java solutions using byte arrays [1], and systems that rely on automatic memory management provided by languages such as Java [11, 23, 29, 30] and C# [19]. While automatic memory management facilitates system development, garbage collection (GC) activities add another layer of unpredictability to query performance. As we show in Section 3, garbage collection can significantly slow down query execution in some cases.

To address the above problems, we propose a new approach where data analytics applications execute in separate containers but the global resource manager elastically adjusts the memory allocated to these containers. Our optimization goal is to minimize failures and the total execution time of all applications subject to the physical limit on the total amount of memory. There are several reasons why this is a difficult problem.

First, elastic memory allocation is not supported in most systems. For Java-based systems, the maximum heap size of a JVM stays constant during its lifetime. Even for C/C++-based systems such as Impala [15], limiting the resource of a process is usually done through Linux utilities such as `cgroups` or `ulimit`, which do not have the functionality to change the quota on-the-fly as well. In this paper, we focus on Java-based systems and address this problem by extending the JVM to enable dynamically changing the maximum heap size at runtime.

Second, in order to elastically and dynamically allocate memory to data analytics applications, we need to understand how extra memory can prevent failures and speed up applications. In this paper, we show how changing the heap size of an application can dramatically reduce GC overheads.

Third, given how GC and out-of-memory failures affect an application's execution time, we need to develop a model that predicts the overheads during execution to drive memory allocation decisions. In this paper, we present a preliminary, machine-learning based approach to perform these predictions.

BeyondMR'16, June 26-July 01 2016, San Francisco, CA, USA

 ISBN 978-1-4503-4311-4/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2926534.2926541

Finally, we discuss an algorithm that uses predicted garbage collection times and out-of-memory information to orchestrate memory allocation between different data analytics applications.

In summary, we make the following contributions:

- We quantitatively demonstrate the negative impact of GC on the execution time of data analytics queries in a modern, Java-based system. We show how changing the heap size directly impacts the execution time (Section 3).
- We show how to modify the JVM to enable dynamic modifications of the application heap layout and thus enable the elastic management of its memory utilization (Section 4.1).
- We develop a machine-learning based technique for predicting the GC overhead for an application and whether that application is expected to run out of memory (Section 4.2).
- Finally, we discuss an algorithm for dynamic memory management in a big data analytics system (Section 4.3).

2. BACKGROUND

Automatic memory management simplifies software development but the associated garbage collection is known to cause performance variations that are difficult to control: The GC policy, while customizable by the programmer to some extent, is usually controlled by the system. Depending on the policy and the heap state, the time and frequency of garbage collections may vary significantly, and, as we show in Section 3, may significantly impact query performance.

Among the languages with automatic memory management, Java has been widely used in data analytics systems [11, 23, 29, 30]. Over the past ten years, there have been several Java Virtual Machine (JVM) implementations with various GC algorithms. However, most of the contemporary ones share the concept of generations [2]. With this design, the heap space is partitioned into two generations for storing objects with different ages. Initial memory allocation requests go to the young generation. When the young generation fills up, a GC is triggered to move objects to the old generation if they survive the collection. A subsequent old generation collection, if needed, may get triggered for cleaning up more space. The triggering condition of a GC differs with different policies. It is possible to have a collection even if there is still a large amount of free memory.

To simplify the problem definition, we make the following assumptions in this work: First, we focus on Java and use OpenJDK as the reference JVM implementation. Second, we assume that there are two generations: the young generation and the old generation. Third, we assume that garbage collections are only triggered when there is not enough heap space. We argue that even with these assumptions, our approach is broadly applicable to modern Java-based analytics systems.

3. PERFORMANCE IMPACT OF GC

In this section, we show a concrete example of how garbage collection can impact query execution time: We execute a simple, data analytics application with a large memory footprint while varying its heap-size limit. The application is a self-join query on a synthetic dataset containing ten million tuples with two `long` columns, running on one Myria [11] worker using default GC collectors (`-XX:+UseParallelGC`).

Figure 1 shows the query run times for different heap-size limits. When the heap is large, the query run time converges to approximately 46 seconds, which is the pure execution time

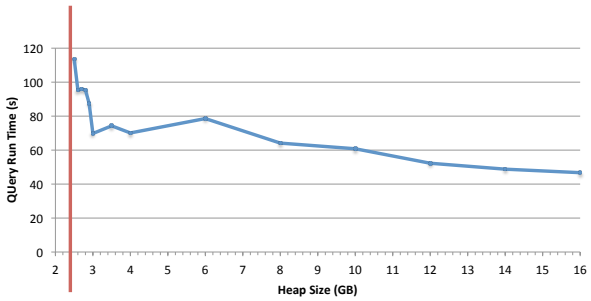


Figure 1: Impact of GC: Runs with heap-size limits below 2.4 GB (red vertical line) run out of memory.

with almost no GC. When we shrink the heap size, however, the run time increases moderately due to spending more time on GC, until the heap size is slightly less than 3 GB where the run time increases drastically to 95 seconds. The query fails with an out-of-memory error when the limit is 2.4 GB or less.

This experiment demonstrates that the performance impact of GC depends on the memory limit. More specifically, the query does not benefit from extra memory when the heap-size limit is large, but its run time can be dominated by GC when the limit is too close to the minimum amount of memory necessary to run the application. Of course, when memory is insufficient, the application cannot complete unless it spills to disk. However, we may avoid failures and GC thrashing by dynamic adjusting the heap size limit.

4. TOWARD ELASTIC MEMORY

We present the three core building blocks to enable elastic memory management for cloud analytics applications.

4.1 Enabling dynamic heap adjustments

The first building block is the ability to change the memory size limits of data analytics applications. Since our focus are Java-based implementations, we do so by modifying OpenJDK. This enables us to adjust memory limits for JVMs such as Spark worker processes and YARN containers.

OpenJDK manages an application's address space as follows: To launch a JVM process, its maximum heap size needs to be specified first. The JVM then asks the operating system for the corresponding address space size and divides the space into generations based on its internal size policy. The size of the total memory space then remains constant during its lifetime.

This rigid design, however, is unnecessary. For operating systems that support overcommitting memory, a logical space does not occupy any memory until it is used. This property, together with 64-bit address spaces, allow us to reserve an extremely large address space when launching a JVM. [1] The actual memory limits, including generations and other subspaces in the heap, can be monitored later during runtime.

We modify the source code of OpenJDK to implement this feature. We set the initial heap size to be a large number and add a socket-based API through which the JVM receives commands to report its memory usage and adjust memory limits. To control GCs directly, we also disable internal policies and make the JVM wait for an external command to trigger a GC. If more memory is needed but is unavailable given the current limits, we provide options for the JVM to either fail immediately or

[1]To avoid performance degradation due to virtual memory swapping, we disable swap in our experiments.

sleep until more memory is available.

4.2 Estimating GC costs

The second building block is the ability to estimate the impact of GC on a data analysis application. In this section, we develop a machine learning model that predicts the duration of the next GC for a given query. We define the GC cost as the total CPU time spent on GC.

The GC time depends primarily on the number and total size of the live and dead objects in the collected region. Unfortunately, getting such detailed statistics is expensive, as we need to traverse the object graph similarly as in a GC. Paying such a cost for each process at every prediction defeats the goal of avoiding GCs in the first place. What we can access more easily is the state of the execution plan: Data analytics queries consist of operators with large in-memory data structures, such as hash tables and buffers, which dominate the state of the heap and thus determine the GC time. Additionally, operator statistics determine the data structure sizes. Hence, our approach is to predict GC times based on operator statistics.

In this paper, we focus on two operators with large data structures in Myria: join and aggregate. We wrap these data structures with the functionality to report statistics, and instrument them during query execution. The statistics that we collect are: the number of input tuples processed (total and the delta since the last GC), the number of distinct join or aggregation key values (similarly, total and delta), and the number and the data types of the input columns. We then build models to predict GC times using these features.

We first build a model for each operator by running each operator independently with varying input datasets. We pick the M5P model with default settings among several models in Weka [10] since it gives us the most accurate predictions overall. We predict the user and the system space times spent on each generation separately, then sum them up to produce the total predicted GC CPU time. Figure 2(a) 2(b) show the results for aggregate and join respectively. The experiments were done on Amazon EC2 using `r3.xlarge` instances on synthetic datasets with different schemas (# of columns, data types) and different # of tuples and # of keys (total and delta). We use two metrics, namely *root relative squared error* (RRSE) and *relative absolute error* (RAE), to measure the accuracy of the predictions. Testing is done using 10-fold cross-validation. As the figures show, our per-operator models are able to predict GC times with good accuracy.

To predict the GC time for a query, we predict and add the GC time for each operator in the query. Figure 2(c) shows the result for predicting the GC time for a query containing one join followed by one aggregate. Although the error rates, comparing to single operator models, have increased, we posit that the precision suffices to make memory allocation decisions. We are studying more complex queries in ongoing work.

4.3 Dynamically allocating memory

The last component of elastic memory management is a global scheduler that monitors concurrently executing queries and makes dynamic memory re-allocation decisions to maximize a global objective function.

Making decisions based on the total future GC time for all queries is difficult because prediction errors increase for predictions far into the future. Instead, we propose to make decisions adaptively at each timestep t for some small period $[t, t + \delta_t]$. A query may transition into the following states in time δ_t: run out of memory (OOM), experience a GC (GC), or have no GC triggered (NOGC). To know which state the query will be in, we estimate the size of all the live objects and the used heap space at $t + \delta_t$ based on past statistics.

Query	Timestep 1	Timestep 2	Timestep 3
Q1	5, 1, NOGC	5 ⇒ 4, 1.5, NOGC	4 ⇒ 2, 2, **NOGC ⇒ GC**
Q2	5, 2, NOGC	5 ⇒ 6, 3, **GC ⇒ NOGC**	6, 4, NOGC
Q3	5 ⇒ 10, 4, **GC ⇒ NOGC**	10, 6, NOGC	10 ⇒ 12, 9, **OOM ⇒ GC**

Table 1: Dynamic memory allocation example.

The scheduler can change the heap size limit of a query at t to affect its states at $t + \delta_t$. Assuming two query states s_1 and s_2, we define the benefit of switching from s_1 to s_2 as:

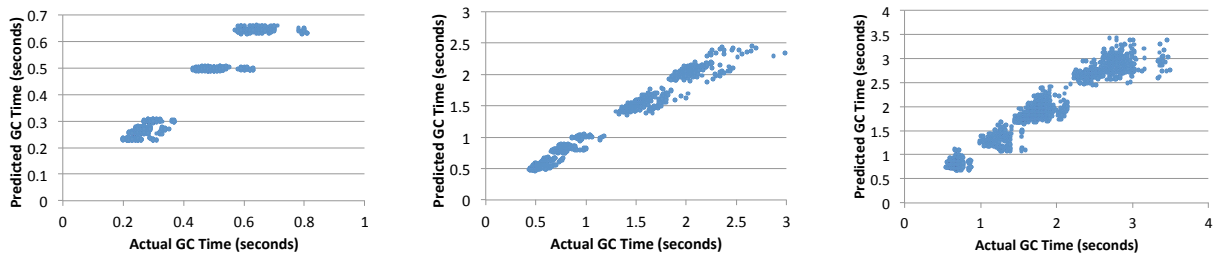
- 0: if both s_1 and s_2 are OOM or NOGC,
- the difference of predicted GC times: if both s_1 and s_2 are GC, and
- ∞: if s_1 is OOM and s_2 is GC or NOGC. [2]

The scheduler can then adjust per-query memory limits by maximizing the total benefit within the physical memory constraint. We explore the promise of such a scheduler through a detailed example. Consider the queries from Table 1. The state of each query at a timestep is a triple of: memory limit, used memory size, and predicted state in the next δ_t. Assuming a total of 20 GB of memory, we assign each query 5 GB (*e.g.*, initial assignments could be based on prior knowledge of the queries) in the beginning and have 5 GB left. At timestep 1, since we think that Q3 will experience a GC, we assign the 5 GB free memory to it to increase its limit to 10 GB to prevent the GC. Then at timestep 2, we decide to shift 1 GB from Q1 to Q2 to avoid the predicted GC of Q2 while not affecting the state of Q1. At timestep 3, since Q3 is predicted to be out-of-memory, we shift 2 GB from Q1 to save Q3. Although Q1 will have to GC, it is still better than having Q3 fail according to our objective function, and we pick Q1 instead of Q2 since the estimated GC time of Q1 is shorter. All the state changes are in bold. We prevent one OOM and save two GCs by paying the cost of triggering one GC with relatively small cost.

5. RELATED WORK

There is a rich body of work on memory allocation within a single machine. Generally, the performance models of single components are first built, then a scheduler makes decisions accordingly to maximize an objective function. The difference is in the components. Several approaches focus on **queries**. Some [5, 8, 21] make buffer allocation decisions based on query page access models. Others [4, 22] try to meet query performance goals in real-time database systems. A third set of methods [26] manages application resources in NUMA platforms by measuring performance counters. More recently, Narasayya *et al.* [20] develop techniques to share bufferpool with multiple tenants. Within a query, several approaches focus on **operators**. Anciaux *et al.* [3] discuss how to allocate memory among operators for memory-constrained computing devices. Davison *et al.* [7] develop models to sell resources to competing operators to maximize profit. Garofalakis *et al.* [9] schedules operators in parallel with multidimensional resource constraints in NUMA systems. Finally, Storm *et al.* [25] manage memory

[2] Here, we treat all OOMs equally for simplicity, but we could give them different costs based on query properties.



(a) One aggregate operator. RRSE: 16.69%, RAE: 22.44%.

(b) One join operator. RRSE: 10.77%, RAE: 14.44%.

(c) A query containing one join and one aggregate. RRSE: 31.60%, RAE: 28.68%

Figure 2: GC time estimations for single operators and a query containing multiple operators.

across **database system components**. Although they share the idea of memory management across multiple components for an overall goal, their focuses are single machine problems and do not investigate the impact of garbage collection.

In the world of cloud-based data analytics, the resources of a cluster are shared by multiple users and queries managed by a global resource manager. Some techniques schedule queries before execution. Li *et al.* [17] investigates how to partition queries on heterogeneous machines based on system calibrations and query optimizer statistics. Herodotou *et al.* [12, 13] tune Hadoop application parameters based on machine learning models built by collected job profiles to predict job run times. To save different systems from self-scheduling, several general-purpose resource managers and schedulers have emerged [14, 27, 28]. However, these techniques all lack the ability to adjust memory allocations dynamically.

Finally, some techniques make decisions adaptively. Lang *et al.* [16] develop a system to schedule transactional workloads on heterogeneous hardware resources for multiple tenants. Schaffner *et al.* [24] aim at minimizing the tail latency of tenant response times in column database clusters. However, their focuses are short-lived requests and do not study garbage collection. For adaptive GC tuning, Cook *et al.* [6] provide two GC triggering policies based on real-time statistics, but do not investigate memory management across applications.

To the best of our knowledge, we are not aware of any similar effort of modifying a JVM implementation to have dynamic memory sizes.

6. CONCLUSION

In this paper, we argue for elastic memory-management in big data systems supporting analytics applications and show preliminary results for the three necessary building blocks: dynamic heap-size adjustment, garbage collection overhead estimation, and dynamic memory allocation.

Acknowledgments: This project is supported in part by the NSF through grant IIS-1247469, gifts from EMC, Amazon, Facebook, and the Intel Science and Technology Center for Big Data.

7. REFERENCES

[1] Tungsten: Memory management and binary processing on spark. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[2] Memory management in the Java HotSpot™ virtual machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, 2006.

[3] N. Anciaux et al. Memory requirements for query execution in highly constrained devices. In *VLDB*, 2003.

[4] K. P. Brown et al. Managing memory to meet multiclass workload response time goals. In *VLDB*, 1993.

[5] C. Chen et al. Adaptive database buffer allocation using query feedback. In *VLDB*, 1993.

[6] J. E. Cook et al. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *SIGMOD*, 1996.

[7] D. L. Davison et al. Dynamic resource brokering for multi-user query execution. In *SIGMOD*, 1995.

[8] C. Faloutsos et al. Predictive load control for flexible buffer allocation. In *VLDB*, 1991.

[9] M. N. Garofalakis et al. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, 1997.

[10] M. Hall et al. The weka data mining software: An update. 2009.

[11] D. Halperin et al. Demo of the Myria big data management service. In *SIGMOD*, 2014.

[12] H. Herodotou et al. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.

[13] H. Herodotou et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.

[14] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[15] M. Kornacker et al. Impala: A modern, open-source SQL engine for hadoop. In *CIDR*, 2015.

[16] W. Lang et al. Towards multi-tenant performance slos. *IEEE Trans. Knowl. Data Eng.*, 2014.

[17] J. Li et al. Resource bricolage for parallel database systems. *Proc. of the VLDB Endow.*, 2014.

[18] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[19] D. G. Murray et al. Naiad: A timely dataflow system. In *SOSP*, 2013.

[20] V. R. Narasayya et al. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. of the VLDB Endow.*, 2015.

[21] R. T. Ng et al. Flexible buffer allocation based on marginal gains. In *SIGMOD*, 1991.

[22] H. Pang et al. Managing memory for real-time queries. In *SIGMOD*, 1994.

[23] T. A. Project. Apache Giraph, http://giraph.apache.org/.

[24] J. Schaffner et al. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, 2011.

[25] A. J. Storm et al. Adaptive self-tuning memory in DB2. In *VLDB*, 2006.

[26] P. Tembey et al. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *SoCC*, 2014.

[27] V. K. Vavilapalli et al. Apache hadoop YARN: yet another resource negotiator. In *SoCC*, 2013.

[28] M. Weimer et al. REEF: retainable evaluator execution framework. In *SIGMOD*, 2015.

[29] T. White. *Hadoop: The Definitive Guide*. 2009.

[30] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.