Elastic Memory Management for Cloud Data Analytics

Jingjing Wang and Magdalena Balazinska
Dept. of Computer Science & Engineering, University of Washington

Abstract

We develop an approach for the automatic and elastic management of memory in shared clusters executing data analytics applications. Our approach, called ElasticMem, comprises a technique for dynamically changing memory limits in Java virtual machines, models to predict memory usage and garbage collection cost, and a scheduling algorithm that dynamically reallocates memory between applications. Experiments with our prototype implementation show that our approach outperforms static memory allocation leading to fewer query failures when memory is scarce, up to 80% lower garbage collection overheads, and up to 30% lower query times when memory is abundant.

1 Introduction

The analysis of large datasets is an important problem and many big data systems are available to facilitate this task [2,29,33,36,48,53]. To handle large data sizes, these systems execute in shared-nothing clusters. Whether public or private, clusters are typically shared by many queries (also called "applications")[1] and even many systems executing in the same cluster at the same time. In such shared clusters, a resource manager [25,47] is responsible for the resource allocation between systems and applications. Modern resource managers rely on containers (*e.g.*, YARN [47], Docker [3], or Kubernetes [5] containers), which isolate applications that share the same machine and provide hard resource limits. Application resource requirements are both constrained and protected by the containers. Figure 1 illustrates the interaction between a resource manager and containers: the resource manager launches containers with resource limits and schedules applications inside those containers.

Many modern data analytics systems, such as Spark [53], Impala [29], GraphLab [33], Giraph [2], and Myria [22,48], strive to maximally utilize memory, yet memory remains an expensive resource. In this paper, we focus in particular on *memory* allocation. However, container-based scheduling has limitations for managing memory. When an application needs to run, it must estimate its resource requirements 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. The challenge, however, is that it is hard to estimate the memory need of a data analytics application before executing it because it may depend on multiple runtime factors including the cardinalities of intermediate results, which are known to be hard to estimate [27,31].

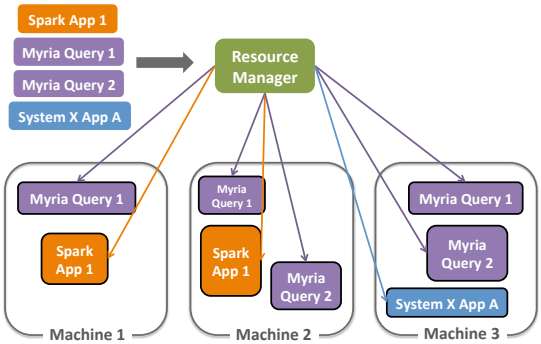

Figure 1: A resource manager schedules multiple applications from multiple systems (Spark [53], Myria [48], and System X) in a shared cluster. An application may have multiple processes across multiple machines. The resource manager schedules applications by putting them in containers with resource limits.

Having an inaccurate memory usage estimate can harm query performance in multiple ways. If the estimate is too high, cluster resources may be under-utilized. If the estimate is less than the minimum amount of memory needed to complete the query, the system must either spill data to disk, which leads to performance degradation, or fail with an out-of-memory error, wasting the resources

[1] In this paper, we focus on applications that correspond to analytical queries and use the terms "application" and "query" interchangeably.

already consumed by the query. This challenge exists in systems with manual memory management, such as those written in C/C++ [29, 33], in Java-based systems that use byte arrays [8], and in systems that rely on automatic memory management provided by runtimes such as Java [2, 48, 52, 53] and the .NET Common Language Runtime (CLR) [36]. The situation is more complicated when garbage collection (GC) is used for automatic memory management, since GC activities add another layer of unpredictability to query performance. Even if the resource estimates are sufficient for the query to complete, garbage collection in some cases can significantly slow down query execution. As a concrete example, we demonstrate how changing the maximum heap size of Java-based systems can significantly impact query time in Section 2.

To address these problems, we develop a new approach, called *ElasticMem*, where data analytics applications execute in separate containers, but the resource manager *elastically* adjusts the memory allocated to these containers. The optimization goal is to jointly minimize failures and total execution time of all applications subject to the physical limit on the total amount of memory in the cluster. We presented the vision behind the approach and a few preliminary results in a short workshop paper [49]. In this paper, we develop the approach in full.

Elastic container memory management is a difficult problem. First, elastic memory allocation is not supported in most systems. For Java-based systems, the maximum heap size of a Java virtual machine (JVM) stays constant during its lifetime. For C/C++-based systems such as Impala [29], limiting the resource of a process is usually done through Linux utilities such as `cgroups`, which do not expose functionality to change resource limits at runtime. For systems that run in CLR [36], the problem is opposite: No control on the heap size can be specified, so the heap can grow arbitrarily up to the total physical memory. 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 these applications. We need models of GC benefits and overheads. Finally, we need an algorithm that uses the models to orchestrate memory allocation across multiple data analytics applications.

We present our approach to address all three challenges. We focus on analytical applications, in particular *relational algebra* queries on large data, and Java-based systems. Since memory management in Java containers (e.g., YARN [47]) is determined by JVMs internally, we focus on how and when to change the memory layouts of JVMs. Specifically, our contributions are the following:

- We show how to modify the JVM to enable dynamic changes to an application's heap layout for elastic management of its memory utilizations (Section 3.1).
- Our key contribution is an algorithm for elastically managing memory across multiple applications in a big data analytics system to achieve an overall optimization goal (Section 3.2). In this paper, we present scenarios where each query runs in one JVM and multiple queries run in one machine, but our approach can be extended to a multi-machine setting.
- In support of elastic memory management, we develop a machine-learning based technique for predicting the heap state and GC overhead for a relational query and whether it is expected to run out of memory (Section 3.3) based on operator statistics. Since the common approach for implementing relational operators in memory, such as joins and aggregates, is to use hash tables [19], we build models that use hash table statistics as input.

We evaluate our elastic memory management techniques using TPC-H queries [6] on Myria [22, 48], a shared-nothing data analytics system, against containers with fixed memory limits. In our experiments, our approach outperforms static allocation: It reduces the number of query failures; it reduces query times by up to 30%, GC times by up to 80%, and overall resource utilization (Section 4).

2 Performance Impact of Automatic Memory Management

Many big data analytics systems today, including Spark [53], Flink [1], Hadoop [52], Giraph [2], and Myria [48], are written in programming languages with automatic memory management, specifically Java. Garbage collection associated with automatic memory management is known to cause performance variations that are hard to control: The GC policy, although customizable by the programmer to some extent, is controlled by the runtime internally. Depending on the policy and heap state, the time and frequency of GCs may vary significantly and, as we later show in this section, may significantly impact query performance.

Over the past decade, there have been several JVM implementations with various GC algorithms. However, most of the contemporary ones share the concept of generations [9]. With this design, the heap space is partitioned into multiple generations for storing objects with different ages. Figure 2 illustrates the internal state of a JVM heap with two generations. Initial memory allocation requests always go to the young generation. When it fills up, a GC is triggered to clean up dead objects. There are different types of GCs as shown in Figure 2. In a young collection, live objects in the young generation are promoted to the old generation. In a full collection, dead objects are cleaned from both generations in addition to promotions. The type of collection to trigger depends on whether a promotion failure, *i.e.*, insufficient space for promoting

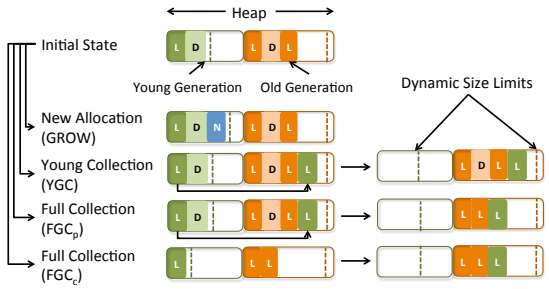

Figure 2: Internal heap states of a JVM before and after actions of new object allocation, young generation collection, and full collections, starting from an initial state. Dark blocks: (L)ive objects, light blocks: (D)ead objects, blue blocks: (N)ew objects. Dashed lines: generation size limits that can be changed in real time by our approach. We describe `FGC`$_p$ and `FGC`$_c$ in Section 3.2.3.

objects from the young generation, is expected to occur or actually occurs. In this paper, we use OpenJDK as the reference JVM implementation. We focus on the common class of GC algorithms that use a young and old generation, and leave extensions to other languages and GC algorithms to future work.

We show a concrete example of how GC can impact query execution by executing a self-join query on a synthetic dataset containing ten million tuples with two `int` columns, on three systems: Myria, Spark 1.1 and Spark 2.0, using one process on one machine with default GC collectors (`-XX:+UseParallelGC`). Figure 3 shows the query execution times with different heap-size limits. Each data point is the average of five trials with error bars showing the minimum and maximum values. For both Myria and Spark 2.0, when the heap is large, the query time converges to approximately 35 seconds, which is the pure query time with almost no GC. When we shrink the heap size, however, the run times increase moderately due to more GC time. For Myria, the run time increases from 35 seconds to 55 seconds when the heap size goes from 16 GB to 3 GB, and further increases drastically to 141 seconds when the heap size shrinks from 3 GB to 2 GB. Eventually, Myria fails with an out-of-memory error when the limit is less than 2 GB. Similarly, the query time for Spark 1.1 has a steep increase from 86 to 466 seconds when the heap size changes from 5 to 4 GB, and the query fails when the heap size is less than 4 GB. Spark 2.0 follows a similar trend as Myria, but does not fail even with only 500 MB of memory because it is able to spill data to disk when memory is insufficient. As a result, however, its execution time increases to 127 seconds.

3 Elastic Memory Allocation

In this section, we present our approach, called *ElasticMem*, for elastic memory allocation. ElasticMem comprises three key components. First, ElasticMem needs

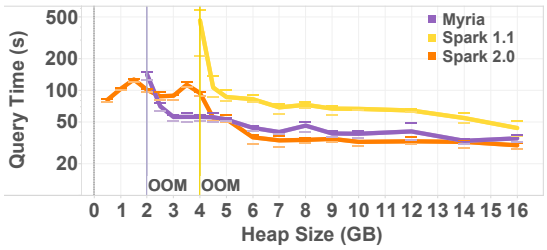

Figure 3: Impact of GC on query execution time in Myria, Spark 1.1, and Spark 2.0. The y-axis uses a log scale.

JVMs that can change memory limits dynamically, and we describe how we modify OpenJDK to enable this feature in Section 3.1. Second, the heart of ElasticMem is a memory manager that dynamically allocates memory across multiple queries (Section 3.2). Finally, to drive the manager's allocation decisions, ElasticMem uses models that predict the heap state and the GC costs (*i.e.*, impact on run time) and benefits (*i.e.*, expected freed memory) at any point during query execution (Section 3.3). The implementation of our approach is available at our website [4].

3.1 Implementing Dynamic Heap Adjustment in a JVM

OpenJDK manages an application's memory as follows: First, the user specifies the maximum heap size of a JVM process before launching it. The JVM then asks the operating system to reserve the heap space and divides the space into generations based on its internal size policy as in Figure 2. During program execution, if a memory allocation request cannot be satisfied due to insufficient memory, the JVM may trigger GCs to release some memory. If not much memory is released after spending a large amount of time on GC, the JVM throws an `OutOfMemory` error. The maximum heap size stays constant during a JVM's lifetime. It cannot be increased even if an `OutOfMemory` is thrown while more memory is available on the machine, or decreased if heap space is underutilized.

This rigid design, however, is unnecessary. For operating systems that support overcommitting memory, a logical address space does not physically occupy any memory until it is used. This property, together with 64-bit address spaces, allow us to reserve and commit a large address space when launching a JVM. The actual memory limits on heap spaces, such as generations, can be modified later during runtime.

We modify the source code of OpenJDK to implement this feature. We change the JVM to reserve and commit a continuous address space of a specified maximum heap size (`-Xmx`) when it launches. The initial size limit of each generation is set according to the JVM's internal policy. We make the maximum heap size large enough such that the per-generation limits are sufficiently large to

become irrelevant. Additionally, we add our new dynamic size limits to both the young and old generation of a JVM p, denoted with $y_{limit}(p)$ and $o_{limit}(p)$ respectively. Our memory manager changes these limits at runtime. We set their initial values to reasonably small numbers (*e.g.*, 1 GB) and prevent each generation from using more memory than its dynamic limit.

To interact with the JVM, we add a socket-based API through which the JVM receives instructions such as requests for the current heap state, memory limit adjustments, or GC triggers. We disable the JVM's internal GC policies to let our memory manager control when and which GCs to happen. We modify GC implementations to always release recycled memory to the OS. If more memory is needed but unavailable given the current limits, we let the JVM pause until more memory is available. We implement our changes on top of OpenJDK 7u85's default heap implementation (`ParallelScavengeHeap`), which contains approximately 1000 lines of code.

3.2 Dynamic Memory Allocation

The main component of ElasticMem is a memory manager. It monitors concurrently executing queries and alters their JVMs' memory utilizations by performing actions on the JVMs, such as triggering a GC or killing the JVM. Each action has a value, and the objective is to maximize the sum of all action values. A value is a combination of several factors, including whether the action kills a JVM, causes a JVM to pause, or how efficiently it enables the JVM to acquire memory: *i.e.*, the ratio of time spent over space acquired (from the OS or recovered through a GC).

The manager makes decisions according to two pieces of information: the JVM heap states and the estimated values of performing actions on the JVMs. Because predicting these values far into the future carries significant uncertainty, and because our changes to the JVM enable us to adjust memory limits without any overhead, we develop a dynamic memory manager. The manager makes decisions adaptively at each timestep t for some small period $[t, t+\delta_t]$. At t, the manager gathers runtime statistics from each JVM and performs actions on it. Queries then execute for time δ_t. Their states change and the manager makes another round of decisions at $t+\delta_t$. We describe our allocation algorithms in this section, starting with a more precise problem statement.

3.2.1 Problem Statement

We start with a single-node and a one-process-per-query scenario. As introduced in Section 1, each JVM is a container that executes a single query (or query partition). We model query execution as the process of accommodating the memory growth of the corresponding JVM. For a period $[t, t+\delta_t]$, the memory usage of a JVM may grow by some amount. We can perform various actions to the JVM to affect its memory utilization: allocate enough memory for the expected growth, trigger a GC, which may require extra memory in the short term but free up memory in the longer term, kill the JVM to release all its memory, or do nothing, which may stall a JVM if it cannot grow its memory utilization as needed.

Consider a single physical machine with a total amount of memory M. A set of N JVMs $\{p_1, \ldots, p_N\}$ is running on it, each has used some space in both the young and the old generation. At the current timestep t, we need to allocate M across the N JVMs, such that the total memory used does not exceed M, while minimizing a global objective function.

The memory that must be allocated to a JVM is entirely determined by the action that the manager selects. For example, to perform a young generation GC, the old generation needs to have enough space to accommodate the promoted young generation live objects. The manager must increase the memory limit for the old generation to accommodate the added space requirement. We denote with $y_{cap}(p_i, a_i)$ and $o_{cap}(p_i, a_i)$, the minimal amount of memory that must be allocated to the young and old generation of JVM p_i, if the manager chooses action a_i. These values refer to the new required totals and not increments.

Each action has a value that contributes to the global objective function. We denote the value of action a_i on p_i with $value(p_i, a_i)$. The objective function is thus:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^{N} value(p_i, a_i), a_i \in Actions, \\ &\text{subject to} && \sum_{i=1}^{N} (y_{cap}(p_i, a_i) + o_{cap}(p_i, a_i)) \leq M, \end{aligned}$$

where *Actions* is the set of possible actions. In our approach, the $value(p_i, a_i)$ is a structure with multiple fields. We describe its internal structure and how to sum and compare values in Section 3.2.3 below.

The above definition can be extended to a shared-nothing cluster scenario by letting the manager make decisions independently for each machine.

3.2.2 Runtime Metrics

Several runtime metrics are needed to compute the value and the space requirements of actions. Some are reported by the JVM while others are estimated by the manager:

Metrics reported by the JVM: For a JVM p at timestep t, $y_{limit}(p,t)$ and $o_{limit}(p,t)$ are the current memory limits of the young and old generation. The manager sets those limits at the previous timestep. However, only some of the space in each generation is used at t, and the JVM reports the used sizes as $y_{used}(p,t)$ and $o_{used}(p,t)$.

Metrics estimated by the manager: Besides the above metrics, we also need to estimate some values

Value	Meaning
$y_{limit}(p,t)$	Size limit of the young gen
$o_{limit}(p,t)$	Size limit of the old gen
$y_{used}(p,t)$	Total used space in the young gen
$o_{used}(p,t)$	Total used space in the old gen
$\hat{y}_{live}(p,t)$	Total size of live objects in the young gen
$\hat{o}_{live}(p,t)$	Total size of live objects in the old gen
$\hat{y}_{dead}(p,t)$	Total size of dead objects in the young gen
$\hat{o}_{dead}(p,t)$	Total size of dead objects in the old gen
$\hat{grw}(p,t)$	Estimated heap growth until next timestep
$\hat{gc}_y(y_{obj}(p,t))$	Time to perform a young collection
$\hat{gc}_o(o_{obj}(p,t))$	Time to perform an old collection

Table 1: Runtime metrics reported by JVM p or estimated by the manager at timestep t. $\hat{x}$ indicates that x is estimated. "gen" is short for generation.

that are not directly available. First, the space used in the young and old generation of a JVM is further divided into live and dead objects. The manager estimates the total size of those objects, which we denote with $\hat{y}_{live}(p,t)$, $\hat{y}_{dead}(p,t)$, $\hat{o}_{live}(p,t)$ and $\hat{o}_{dead}(p,t)$. We use $\hat{x}$ to indicate that a value x is estimated by the manager. Second, the manager needs to estimate p's heap growth, $\hat{grw}(p,t)$, before the next timestep, where $\hat{grw}(p,t) = \hat{y}_{used}(p,t+\delta_t) - y_{used}(p,t)$. Finally, to model the impact of a GC, the manager needs to know how much memory a GC will free, and how much time it will take. Since the target of a GC is the set of all objects in the generation(s) undergoing the GC, we use $y_{obj}(p,t)$ to denote the set of all the objects in the young generation and similarly $o_{obj}(p,t)$ for the old generation.[2] $\hat{gc}_y(y_{obj}(p,t))$ and $\hat{gc}_o(o_{obj}(p,t))$ are then the estimated times for a young and an old GC. We describe how the manager estimates these metrics in Section 3.3.

Table 1 summarizes the notation. Since t is the only used timestep, we omit t and only use p as the argument in the rest of the paper when the context is clear.

3.2.3 Space of Possible Actions

There are four types of actions that the manager can choose for each JVM: allowing the JVM to grow by asking the operating system for more memory, reducing the memory assigned to the JVM by performing a garbage collection and recycling space,[3] pausing the JVM if it cannot either grow or recycle enough memory, or as a last resort, killing a JVM to release its entire memory. The manager performs an action for every JVM at each timestep. An action a on a JVM p has value, $value(p,a)$, with a minimum amount of memory needed for p's young and old generations, ($y_{cap}(p,a)$ and $o_{cap}(p,a)$). We denote the time to perform a on p with $time(p,a)$, and the size of the newly available space made by a with $space(p,a)$. The *cost* of an action is the amount of time needed to acquire a given amount of space, or $\frac{time(p,a)}{space(p,a)}$. The manager uses this ratio to compare and choose actions.

The detailed set of *Actions* is as follows:

- `GROW`: Let the JVM grow to continue query execution. In order to reserve space for the growth, the manager must allocate $y_{cap}(p,\texttt{GROW}) = y_{used}(p) + \hat{grw}(p)$ to the young generation and $o_{cap}(p,\texttt{GROW}) = \hat{y}_{live}(p) + o_{used}(p)$ to the old generation. We reserve extra space in the old generation for prospective promotions to preserve the possibilities of having all types of GCs in the future. The cost is the time it takes to request and access the new space, which depends on the size of the space change given by: $y_{cap}(p,\texttt{GROW}) + o_{cap}(p,\texttt{GROW}) - y_{limit}(p) - o_{limit}(p)$. Under normal circumstances, this will be the commonly selected action until space becomes tight and JVMs must start garbage collection or must pause before being able to grow again.
- `YGC`: Trigger a young generation GC. The JVM needs at least the current used space, $y_{used}(p)$, for the young generation, and $\hat{y}_{live}(p) + o_{used}(p)$ for the old generation to avoid a promotion failure. The cost is the GC time $\hat{gc}_y(y_{obj}(p))$, and we expect memory of size $\hat{y}_{dead}(p)$ to be recycled.
- $\texttt{FGC}_\texttt{p}$: Trigger a full GC by first performing a young generation collection to promote live objects to the old generation then performing a GC on the old generation. Similar to `YGC`, we need at least $y_{used}(p)$ and $\hat{y}_{live}(p) + o_{used}(p)$ for the young and old generations respectively. The cost is the GC time $\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p))$ and the space to be recycled is $\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$.
- $\texttt{FGC}_\texttt{c}$: Trigger a full GC by first performing a GC on the whole heap, then trying to promote young generation live objects if possible, without changing the total heap size. Free space from the young generation after the first GC gets shifted to the old generation to make space for copying. Different from $\texttt{FGC}_\texttt{p}$, we only need $y_{used}(p)$ and $o_{used}(p)$ for the young and old generation since the promotion is not mandatory. However, more GC time is needed since the full collection is now performed on both generations instead of only the old generation. We assume that the time grows proportionally to the size of live objects and use $\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p)) * (\hat{y}_{live}(p) + \hat{o}_{live}(p))/\hat{o}_{live(p)}$ as the GC time estimate. The memory to be recycled is also $\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$.
- `NOOP`: Do nothing to the JVM, keep the current limits $y_{limit}(p)$ and $o_{limit}(p)$. As a consequence, the JVM is expected to pause since it cannot either grow or recycle enough memory by doing garbage collection.
- `KILL`: Kill the JVM immediately. As a consequence, the query running in this JVM will fail.

$\texttt{FGC}_\texttt{p}$, which promotes first, is the default behavior in

[2] $y_{obj}(p,t)$ is the union of all the live and dead objects in the young generation of p at t, similarly to $o_{obj}(p,t)$.

[3] The recycled memory is always reclaimed by the OS.

OpenJDK. However, the promotion may fail if the old generation does not have enough free space to absorb young generation live objects, and when it happens, JVM spends much time on copying the live objects back, so that the young generation remains the same as it was before the promotion. In other words, triggering $\texttt{FGC}_\texttt{p}$ while expecting a promotion failure is not cost effective. However, when memory is scarce, the manager may not be able to allocate extra space to avoid the promotion failure. In this case, we need a GC which can still recycle space without increasing the limits. We solve this problem by implementing another full GC procedure, $\texttt{FGC}_\texttt{c}$: We first collect both generations, then shift young generation free space to the old generation to keep the total heap limit unchanged. A young GC is then performed if there is enough space for promoting.

Table 2 summarizes the properties of all actions. $os(m)$ denotes the time to access new memory of size m. We obtain its value by running a calibration program since this value changes for different systems and settings. Figure 2 illustrates the effect of all the actions except for NOOP and KILL, which have the obvious effects.

We define the value of an action with three attributes, where only one of them is set to a non-zero value. For NOOP and KILL, we set the corresponding attributes to 1. For other actions, we use their cost, or time/space efficiency, as the value: *i.e.*, how much time the action needs per unit of space that it makes available. Then for an action a on a VM p, its value $value(p,a)$ is defined as:

$$\begin{cases} value(p,a).cost = \dfrac{time(p,a)}{space(p,a)}, & \text{for } \texttt{GROW}, \texttt{YGC}, \\ & \quad \texttt{FGC}_\texttt{p}, \texttt{FGC}_\texttt{c}, \\ value(p,a).\texttt{NOOP} = 1, & \text{for } \texttt{NOOP}, \\ value(p,a).\texttt{KILL} = 1, & \text{for } \texttt{KILL}, \end{cases}$$

With the above definition, our manager can favor actions by comparing these three attributes in a certain order, as we describe in Section 3.2.4.

3.2.4 Memory Allocation Algorithm

Next, we discuss the allocation algorithm, which allocates memory to the JVMs by performing actions on them at each timestep. We model the problem as a 0-1 knapsack problem. The capacity of the knapsack is the total amount of memory, and the items are actions performed on JVMs. Each action has a value and a minimum space requirement as described in Table 2. The goal is to maximize the total item value in the knapsack without exceeding its capacity.

The 0-1 knapsack problem is known to be NP-complete with a pseudo-polynomial dynamic programming solution [14]. Let $opt_{N,M}$ denote the value of the best scheme for allocating memory of size M to the first N JVMs, $p_1 \cdots p_N$. If a JVM p_i is undergoing a GC, the manager skips it to wait for the GC to complete. Otherwise, it derives $opt_{i,j}$ by enumerating possible actions on p_i and picking the one that leads to the largest value for $opt_{i,j}$. We define the sum of two values as the sum of their three attributes, then the state transition function is defined as:

$$opt_{i,j} = \begin{cases} opt_{i,j} & \text{if } opt_{i,j} > opt_{i-1,j-m} + v, \\ opt_{i-1,j-m} + v & \text{otherwise,} \end{cases}$$

where $v = value(p_i,a), a \in Actions, i \in [1,N], j \in [0,M]$.

To choose between two values, we first check which one has a lower value for attribute KILL, then fewer NOOPs to reduce pausing time, then a smaller time/space ratio. The one with fewer KILL, then fewer NOOP, then smaller time/space ratio, has a *higher* value. To be precise, given two values a and b, we define $a > b$ as:

```
bool operator>(const Value& a,
               const Value& b) {
  if a.KILL < b.KILL return true
  if a.NOOP < b.NOOP return true
  if a.cost < b.cost return true
  return false
}
```

The complexity of the dynamic programming is $O(N * M)$, where N is the number of JVMs and M is the total amount of memory. On modern servers, M can be large if the memory-size units are fine-grained, which would prevent the manager from making fast decisions. At the same time, allocating memory at fine granularity is unnecessary. To enable fast memory-allocation decisions, we define U as the unit of memory allocation, and any allocation is represented as a multiple of U. We discuss two ways of setting U: as a constant or as a dynamically computed variable based on the current heap state, and evaluate their impact on performance in Section 4.

Algorithm 1 and Algorithm 2 show the detailed allocation algorithms. Function ALLOCATE allocates memory of size M across the list of JVMs, P, at the current timestep, and it returns the best allocation scheme, act^{best}, which is a vector of actions for each $p \in P$. The algorithm works as follows: First, we find all the JVMs that are not undergoing a GC as $P - P_{\texttt{INGC}}$ to compute their actions. Because the algorithm allocates memory as increments of U, but $y_{limit}(p)$ and $o_{limit}(p)$ of a JVM p at the current timestep may not be increments of U when U is a dynamic variable, we do not include NOOP in Algorithm 2. Instead, we consider all the combinations of $P - P_{\texttt{INGC}}$ as potential $P_{\texttt{NOOP}}$ (line 4) and use $P' = P - (P_{\texttt{INGC}} \cup P_{\texttt{NOOP}})$ to denote the remaining JVMs. The remaining memory to be allocated is of size M' (line 7). We then apply Algorithm 2 on P' and memory of size M' ($= K$ units of size U). Function KNAPSACK returns the best solution with its value. The generation size limits and value of an action on a JVM are computed as in Table 2. The size limits are aligned to increments of U by function $align(size,U)$ defined as:

Action a	$\mathbf{y_{cap}(p,a)}$	$\mathbf{o_{cap}(p,a)}$	$\mathbf{space(p,a)}$	$\mathbf{time(p,a)}$
GROW	$y_{used}(p) + g\hat{r}w(p)$	$\hat{y}_{live}(p) + o_{used}(p)$	$y_{cap}(p,a) + o_{cap}(p,a)$ $-y_{limit}(p) - o_{limit}(p)$	$os(space(p,a))$
YGC	$y_{used}(p)$	$\hat{y}_{live}(p) + o_{used}(p)$	$\hat{y}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))$
$\texttt{FGC}_\texttt{p}$	$y_{used}(p)$	$\hat{y}_{live}(p) + o_{used}(p)$	$\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p))$
$\texttt{FGC}_\texttt{c}$	$y_{used}(p)$	$o_{used}(p)$	$\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p)) * r,$ $r = (\hat{y}_{live}(p) + \hat{o}_{live}(p))/\hat{o}_{live}(p)$
NOOP	$y_{limit}(p)$	$o_{limit}(p)$		
KILL	0	0		

Table 2: Per-generation size limit requirements, sizes of created space, and time taken for each action a in *Actions* on JVM p at the current timestep. $os(m)$ is the time to access memory of size m. Other symbols are defined in Table 1.

Algorithm 1 The scheduling algorithm: allocates memory of size M across the list of JVMs P, returns the allocation scheme.

function ALLOCATE(P, M)
  $value^{best} = act^{best} = None$
  $P_{\texttt{INGC}} = \{p \in P, p$ is undergoing a GC$\}$
  for $P_{\texttt{NOOP}} \in$ power set of $P - P_{\texttt{INGC}}$ **do**
    $act_p = \texttt{NOOP}, p \in P_{\texttt{NOOP}}$
    $P' = P - (P_{\texttt{INGC}} \cup P_{\texttt{NOOP}})$
    $M' = M - \sum_{p \in P_{\texttt{INGC}} \cup P_{\texttt{NOOP}}} (y_{limit}(p) + o_{limit}(p))$
    Compute U, let $K = M'/U$
    $act', value' = Knapsack(P', K, U)$
    $act_p = act'_p, p \in P'$
    $value.cost = value'.cost$, $value.\texttt{KILL} = value'.\texttt{KILL}$
    $value.\texttt{NOOP} =$ size of $P_{\texttt{NOOP}}$
    if $value > value^{best}$ **then**
      $value^{best} = value$, $act^{best} = act$
  if act^{best} contains only NOOP **then**
    Pick $P_{kill} \subseteq P$, let $act_p^{best} = \texttt{KILL}, p \in P_{kill}$
  return act^{best}

Algorithm 2 The knapsack problem: given the list of JVMs P and K memory units of size U, returns the best allocation and its value.

function KNAPSACK(P, K, U)
  $N =$ size of P
  $opt_{0,j} = 0, j \in [0, K]$
  for $i \leftarrow 1, N$ **do**
    for $j \leftarrow 0, K$ **do**
      for $a \in [\texttt{GROW}, \texttt{YGC}, \texttt{FGC}_\texttt{c}, \texttt{FGC}_\texttt{p}, \texttt{KILL}]$ **do**
        if $a \in [\texttt{YGC}, \texttt{FGC}_\texttt{c}, \texttt{FGC}_\texttt{p}]$ and
          $space(p_i, a) < mingcsave$ **then** continue
        $y_{unit} = align(y_{cap}(p_i, a), U)$
        $o_{unit} = align(o_{cap}(p_i, a), U)$
        if $opt_{i-1, j-y_{unit}-o_{unit}}$ is valid **then**
          $v = opt_{i-1, j-y_{unit}-o_{unit}} + value(p_i, a)$
          if $v > opt_{i,j}$ **then**
            $opt_{i,j} = v$, $trans_{i,j} = (a, y_{unit} + o_{unit})$
  Derive act_p of each $p \in P$ from $opt_{N,K}$ and $trans_{N,K}$
  return $act, opt_{N,K}$

$align(size, U) = ceiling(size/U)$. For GC actions, we defined a constant *mingcsave* to avoid GCs that only recycle a negligible amount of space. We derive *act* from the transition actions *trans* and return them together with the value. They are then merged with $P_{\texttt{NOOP}}$ and $P_{\texttt{INGC}}$ to get the final allocation. We maintain the best allocation and its value across all the powersets. In the end, if the best allocation only contains NOOP actions, we pick some JVMs to kill to make progress. In this work, we pick the query that occupies the largest amount of memory and kill all its JVMs, and we leave other strategies as future work.

3.3 Estimating Runtime Values

The last piece of ElasticMem is the models that estimate JVM values that are necessary for memory allocation decisions yet not directly available as indicated in Table 1.

3.3.1 Heap Growth

To allocate memory to a JVM for the next timestep, the memory manager needs to estimate its memory growth. Different approaches are possible. In this paper, we adopt a simple approach. To estimate the heap growth of JVM p at timestep t, $g\hat{r}w(p,t)$, the manager maintains the maximum change in the young generation's usage during the past b timesteps. To be precise, we define: $g\hat{r}w(p,t) = \max |y_{used}(p,t') - y_{used}(p,t' - \delta_t)|, t' \in [t - b * \delta_t, t]$. In our experiments, we set $b = 3$ empirically. We show in Section 4 that this value yields good performance.

3.3.2 GC Time and Space Saving

The GC time and space saving depend 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 reference graph similarly as in a GC. Paying such a cost for each JVM at every prediction defeats the purpose of reducing GC costs in the first place.

We observe, however, that a query operator's data structures and their update patterns determine the state of live and dead objects, which determines GC times and the amount of reclaimable memory. Our approach is thus to monitor the state of major data structures in query operators, collect statistics from them as features, and use these

Feature	Meaning
nt	Total # of processed tuples
ntd	Delta # of processed tuples since the last GC
nk	Total # of distinct keys in the hash table
nkd	Delta # of distinct keys since the last GC
num_{long}	# of `long` columns
num_{str}	# of `String` columns
sum_{str}	Avg. sum of lengths of all `String` columns

Table 3: Features collected from a hash table.

features to build models. While there are many operators in a big data system, most keep their state in a small set of data structures, for example, hash tables. So instead of changing the operators, we wrap data structures with the functionality to report statistics, and instrument them during query execution to get per-data structure statistics. There are many large data structures, but in data analytics systems, the most commonly used ones by operators with large in-memory state, such as join and aggregate, are hash tables. In this paper, we focus on the hash table data structure. To get predictions for the whole query, we first build models for one hash table, then compute the sum of per-hash-table predictions as the prediction for the whole query. Our approach, however, can easily be extended to other data structures and operators.

Table 3 lists the statistics that we collect for a hash table. A hash table stores tuples consist of columns. A tuple has a key defined by some columns and a value formed by the remaining columns. We collect the number of tuples and keys in a hash table in both generations (both the total and the delta since the previous GC), since new objects are put in the young generation only until a GC. These features are *nt*, *ntd*, *nk* and *nkd*. The schema also affects memory consumption. In particular, primitive types, such as `long`, are stored internally using primitive arrays (e.g. `long[]`) in many systems that optimize memory consumption. However, data structures with Java object types, such as `String`, cannot be handled in the same way, as their representations have large overhead. So we treat them separately by introducing features for primitive types (num_{long}) and `String` types (num_{str} and sum_{str}). The overhead of getting these values from hash tables is negligible. We then build machine learning models to predict the GC times and the total size of live and dead objects as specified in Table 1.

To build models, our first approach to collect training examples is to randomly trigger GCs during execution to collect statistics. The models built from them, however, yielded poor predictions for test points that happen to fall in regions with insufficient training data. As a second approach, we collected training data using a coarse-grained multidimensional grid with one dimension per feature. The examples were uniformly distributed throughout the feature space but they all had the same small set of distinct feature values, the values from the grid. As a result, predictions were excellent for values on the grid but poor otherwise. Using a fine-grained grid, however, is too expensive since the feature space has eight dimensions. For example, if we divide each dimension in four, the total number of grid points is $(4+1)^7 = 78,125$. Assuming that collecting one data point requires 30 seconds, we need $78,125/2/60 \approx 651$ machine hours. Our final approach is thus to combine the previous two: We first collect data using a coarse-grained grid to ensure uniform coverage of the entire feature space, then for each grid cell, we introduce some diversity by collecting two randomly selected data points inside of it. The union of the grid and the random points is the training set. To collect a data point for a hash table, we run a query with only that hash table and a synthetically generated dataset as the input. This approach enables us to precisely control the feature values when we trigger a GC. We then can use any off-the-shelf approach to build a regression model. In our implementation, we use the M5P model [40, 50] from Weka [20] since it gives us the most accurate predictions overall. We evaluate our models in Section 4.2.

4 Evaluation

We evaluate the performance of our memory manager and the accuracy of our models. We perform all experiments on Amazon EC2 using `r3.4xlarge` instances. We do not set swap space to avoid performance degradation due to virtual memory swapping. We execute TPC-H queries [6] on Myria [48], a shared-nothing data management and analytics system written in Java. The TPC-H queries are written in MyriaL, which is Myria's declarative query language, and they are publicly available at [7]. We modify or omit several queries because MyriaL does not support some language features, such as nulls and `ORDER BY`. The final set consists of 17 TPC-H queries: *Q1-Q6*, *Q8-Q12*, and *Q14-19*. To experiment with a broad range of query memory consumption, we execute each query on two databases with scale factors one and two.

4.1 Scheduling

We first compare our elastic manager (*Elastic*) against the original JVM with fixed maximum heap size (*Original*). For Original, we assume that each running JVM gets an equal share of the total memory. We pick 4 memory-intensive TPC-H queries, Q4, Q9, Q18, and Q19, and execute each on two databases, which leads to a total of 8 queries. In all experiments, we execute these 8 queries on one EC2 instance together with our memory manager. All data points are averages of five trials, and we report the minimal and maximal values as floating error bars. Each run of the allocation algorithm takes about 0.15 seconds.

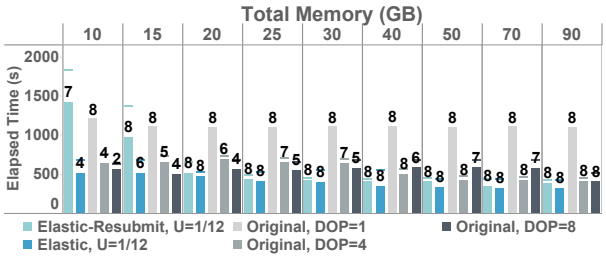

Figure 4: Average elapsed times and # of completed queries (labeled on top of each bar).

We empirically set the constant *mingcsave* from Algorithm 2 to 30 MB. The value of the function $os(m)$ is obtained by running a calibration program, which asks the operating system for memory of size m using `mmap` and accesses it using variable assignments. We take the system time as $os(m)$. For `r3.4xlarge`, we get $os(m) = 0.35s * \frac{m}{1\,\text{GB}}$. We set the interval between timesteps, δ_t, to 0.5 seconds except in Section 4.1.3, where we compare different values of δ_t. In order to avoid query hanging due to frequent GCs that do not recycle much memory, we kill a query after 8 minutes if it is still running. Based on our observation, 8 minutes is long enough for any query to complete with a reasonable amount of memory.

One extreme of Original is serial execution where queries are executed one at a time, while the other extreme is to execute all queries simultaneously. The former approach requires the least amount of memory for all queries to complete but takes longer time, while the latter finishes all queries the fastest when memory is sufficient, however may fail more queries when memory is scarce. We vary the degree of parallelism (*DOP*) for Original to compare these alternatives. To make it fair for Elastic, we also introduce a variant of Elastic, which allows executions to be delayed by resubmitting killed queries serially after all queries either complete or get killed. We call this variant *Elastic-Resubmit*. To avoid livelocks, we only resubmit each killed query once, and each resubmitted query runs only by itself. We leave resubmitting multiple queries simultaneously as future work.

Another important parameter is the size of the memory increment unit U. The value of U can be either fixed or derived in real time. We test fixed sizes of 100 MB, 500 MB, and 1000 MB, and variable sizes as 1/8, 1/12, and 1/16 of the total free space at the current timestep.

4.1.1 Scheduling Simultaneous Queries

First, we submit all queries at the same time. Figure 4 shows the elapsed times, together with the numbers of completed queries while varying the total memory size. The elapsed times are the times for all queries to complete. In this figure, we use U=1/12 as the representative of our elastic manager because it provides the best overall performance across all experiments. We further discuss the performance of different values of U in Figure 5. When memory is abundant ($\geq$ 20 GB), both Elastic managers yield more completed queries and also shorter elapsed times than all the three Original variants. When memory is scarce ($\leq$ 15 GB) and only suffices to execute one query at a time, for 15 GB, Elastic-Resubmit is able to complete all queries with less time than Original, DOP=1. For 10 GB, it only misses one query with a slightly longer time comparing to DOP=1. Based on our observation, the query failed because our manager needs to allocate memory as increments of U, however U is not sufficiently fine-grained. The overhead of elapsed time is due to the elastic method striving to accommodate all queries together before degrading to serial execution. As a proof of concept, we calculate the in-memory sizes of dominant large hash tables of the 8 queries and find that the sum of them is about 14 GB. This experiment shows the advantage of using the elastic manager: it automatically adjusts the degree of parallelism, enabling the system to get high-performance while avoiding out-of-memory failures when possible.

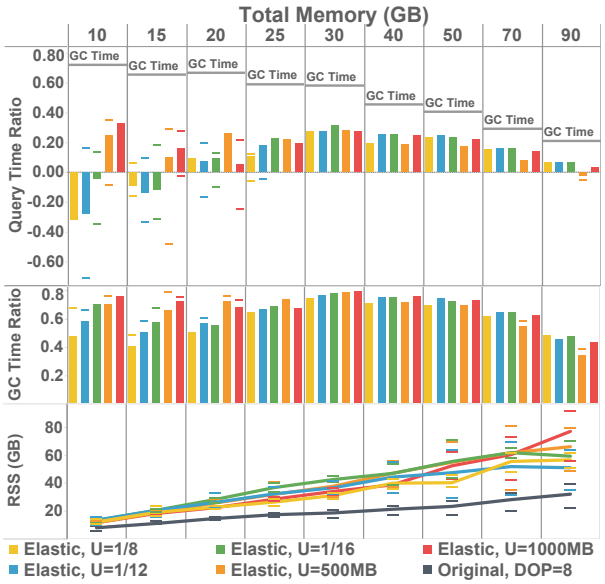

Figure 5: Relative total query time improvement ratio (top) and GC time improvement ratio (middle) for Elastic over Original, DOP=8, and resident set size, RSS (bottom).

In Figure 5, we further drill down on the performance of different variants of our approach. We seek to determine which variant yields the greatest performance improvement compared with non-elastic memory management. Because the elapsed times of Original, DOP=1 are significantly longer than the other two variants, we use Original, DOP=8 as the baseline in this experiment, which also brings fair comparison with our approach. We measure performance in terms of total query execution time, which is the sum of the per-query execution times, and

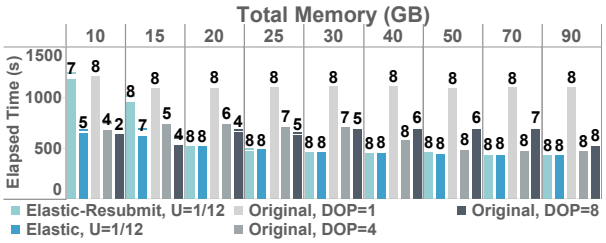

Figure 6: Average elapsed times and # of completed queries (labeled on top of each bar) with 30 seconds delay.

total GC time, the sum of the GC times of all queries. Figure 5 shows the relative improvement percentages in total query execution time and GC time of Elastic over Original, DOP=8, for different values of U, and also the actual physical memory usage (resident set size, RSS).[4] Higher bars indicate greater improvements. When memory is scarce ($\leq$ 15 GB), Elastic with variable values of U (1/8, 1/12 and 1/16) takes longer to execute each query because it strives to finish more queries than Original, DOP=8, as shown previously in Figure 4. When memory is abundant ($\geq$ 20 GB), for any of the values of U, Elastic outperforms Original, DOP=8 on both total query time and GC time. The percentage improvements are between 10% and 30% for query time and 40% to 80% for GC time. We observe that it is caused by Original, DOP=8 triggering GCs that do not recycle much space especially in late stages for large queries but being unable to shift memory quota from small queries, while Elastic can dynamically allocate memory across all queries. The improvement ratios of query time decrease after 70 GB because GC time takes a less portion of query time when memory is abundant. To show the maximum improvement that we can achieve by reducing GC time to zero, we also show the ratios of total GC time to query time in the top subfigure as a reference. Finally, the bottom subfigure shows that our elastic manager is also able to utilize a larger fraction of available physical memory to save on GC time and query time. Importantly, all values of U, especially the three variable ones, yield similar performance indicating that careful tuning is not required.

4.1.2 Scheduling Queries with Delays

To better simulate a real cluster, instead of issuing all the queries at the same time, we submit the above 8 queries with delays. Each query is submitted 30 seconds later than the previous one. Figure 6 shows the elapsed times and the numbers of completed queries. The patterns are similar to the experiment above with no delay (Figure 4), but also different as Elastic can finish the same number of queries with less time when memory is scarce (10 GB), and always beats all variants of Original in terms of both query completion and elapsed time. This is due to the memory flexibility that ElasticMem has: the number of simultaneously running queries is lower when delay is introduced, so Elastic is able to finish more queries faster, while Original stays the same.

4.1.3 Timestep Interval

Finally, we evaluate the sensitivity of the approach to different values of δ_t varying from 0.1, 0.5, or 1 second for U=500MB and U=1/12. We find that when memory is scarce, 0.5 seconds slightly outperforms others by completing more queries with less time, although in general the three δ_ts yield similar performance, which indicates that the approach is not sensitive to small differences when using variable sizes of U and thus careful tuning is not necessary. We omit details due to space constraints.

4.2 GC Models

An important component of ElasticMem is its models that predict the GC time and the space that will be freed (Section 3.3). We evaluate its models in this section. We limit the training space to 12 million tuples and 12 million keys for a hash table, with the schema varying from 1 to 7 `long` columns and 0 to 8 `String` columns with a total of 0 to 96 characters. This training space is large enough to fit all hash tables from TPC-H queries. As described in Section 3.3, we collect approximately 1080 grid points and 1082 random points together as the training set. We also collect a test set of 7696 data points by randomly triggering GC for the 17 TPC-H queries on both databases.

We set the JVM to use one thread for GC (`-XX:ParallelGCThreads=1`) because we observe that the JVM is not always able to distribute work evenly across multiple GC threads. We do not use thread-local buffers (`-XX:-UseTLAB`). We let the JVM always sweep live objects to the beginning of the old generation after each collection (`-XX:MarkSweepAlwaysCompactCount=1`) instead of every few collections to reduce GC cost variance. Among several models available in Weka [20], we pick the M5P model with default settings for its overall accuracy. M5P is a decision tree where leaves are linear regressions [40,50]. We use *relative absolute error* (RAE) to measure the prediction accuracies.[5]

Figure 7 shows the results for both doing 10-fold cross validation on the training set and testing on the random TPC-H test set. For cross validation, the predictions yield RAEs below 5% for every value except o_{dead}. For testing, both y_{dead} and o_{dead} cannot be predicted well, while all others have RAEs lower than 25%. This is because that

[4] We define the improvement percentage as $(x-y)/x$, where x is the value of Original and y is the value of Elastic.

[5] The RAE of a list of predictions P_i and corresponding real values R_i is defined as: $\sum_{i=1}^{n}|P_i-R_i|/\sum_{i=1}^{n}|\bar{R}-R_i|$.

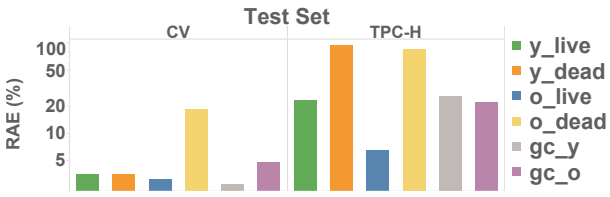

Figure 7: GC Model accuracies on 10-fold cross validation and random TPC-H test set.

the size of dead objects is not strongly correlated with the objects in data structures. Fortunately, the fact that the sum of dead and live objects is the total used size gives us a way to avoid predicting y_{dead} and o_{dead}. Instead, we let $\hat{y}_{dead} = y_{used} - \hat{y}_{live}$ and $\hat{o}_{dead} = o_{used} - \hat{o}_{live}$, where y_{used} and o_{used} can be obtained precisely. Overall, the prediction error rates are low and, as we showed in Section 4.1, suffice to achieve good memory allocation decisions.

5 Related Work

Memory allocation within a single machine: Many approaches focus on sharing memory across multiple objects on a single machine. Several techniques have *queries* as the objects: Some [12, 16, 38] allocate buffer space across queries based on page access models to reduce page faults. Others [11, 39] tune buffer allocation policies to meet performance goals in real-time database systems. A third set of methods [45] uses application resource sensitivities to guide allocation. More recently, Narasayya *et al.* [37] develop techniques to share a bufferpool across multiple *tenants*. Several approaches focus on *operators* within a query. Anciaux *et al.* [10] allocate memory across operators on memory-constrained devices. Davison *et al.* [15] sell resources to competing operators to maximize profit. Garofalakis *et al.* [17] schedule operators with multidimensional resource constraints in NUMA systems. Finally, Storm *et al.* [44] manage memory across *database system components*. Although they share the idea of managing memory for multiple objects with a global objective function, the problems are restricted to single machines, and they ignore GC. Salomie *et al.* [41] move memory across JVMs dynamically by adding a balloon space to OpenJDK but have no performance models or scheduling algorithms. Ginkgo [26] dynamically manages memory for multiple Java applications by changing layouts using Java Native Interface. However, it models performance by profiling specific workloads, while our approach is applicable to arbitrary relational queries.

Cluster-wide resource scheduling: Some techniques develop models to understand how resources affect the runtime characteristics of applications. Li *et al.* [32] partition queries on heterogeneous machines based on system calibrations and optimizer statistics. Herodotou *et al.* [23, 24] tune Hadoop application parameters based on machine learning models built by job profiles. Some other techniques focus on *short-lived requests*. Lang *et al.* [30] schedule transactional workloads on heterogeneous hardware resources for multiple tenants. Schaffner *et al.* [42] minimize tail latency of tenant response times in column database clusters. BlowFish [28] adaptively adjusts storage for performance of random access and search queries by switching between array layers with different sampling rates based on certain thresholds. In contrast, our focus is relational queries on Java-based systems with no sampling. To provide a unified framework for resource sharing and application scheduling, several *general-purpose* resource managers have emerged [25,47,51]. However, they all lack the ability to adjust memory limits dynamically.

Adaptive GC tuning: Cook *et al.* [13] provide two GC triggering policies based on real-time statistics, but do not investigate memory management across applications. Simo *et al.* [43] study the performance impact of JVM heap growth policies by evaluating them on several benchmarks. Maas *et al.* [35] observe that GC coordination is important for distributed applications. They let users specify coordination policy to make all JVMs trigger GC at the same time under certain conditions.

Region-based memory management: Another line of work uses region-based memory management (RBMM) [46] to avoid GC overhead. Broom [18] categorizes Naiad [36] objects into three types with a region assigned to each. Deca [34] manipulates Spark Scala objects in-memory representations as byte arrays and allocates pages for them. While RBMM may reduce GC overhead, it requires that the programmer declare object-to-region mappings and adds complexity to compilation, without eliminating space safety concerns [21].

6 Conclusion and Future Work

In this paper, we presented ElasticMem, an approach for the automatic and elastic memory management for big data analytics applications running in shared-nothing clusters. Our approach includes a technique to dynamically change JVM memory limits, an approach to model memory usage and garbage collection cost during query execution, and a memory manager that performs actions on JVMs to reduce total failures and run times. We evaluated our approach in Myria and showed that our approach outperformed static memory allocation both on query failures and execution times. We leave extensions to other data structures and experiments with more diverse workloads and systems as future work.

Acknowledgment: This project is supported in part by the National Science Foundation through grant IIS-1247469 and the Intel Science and Technology Center for Big Data.

References

[1] Apache Flink. http://flink.apache.org/.

[2] Apache Giraph. http://giraph.apache.org/.

[3] Docker Container. https://www.docker.com/.

[4] ElasticMem. http://myria.cs.washington.edu/projects/2015/09/12/cloud_service.html#elasticmem.

[5] Kubernetes. http://kubernetes.io/.

[6] TPC-H. http://www.tpc.org/tpch/.

[7] TPC-H queries in MyriaL. https://github.com/uwescience/tpch-myrial.

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

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

[10] N. Anciaux, L. Bouganim, and P. Pucheral. Memory requirements for query execution in highly constrained devices. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 694–705. VLDB Endowment, 2003.

[11] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 328–341, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[12] C.-M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 342–353, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[13] J. E. Cook, A. W. Klauser, A. L. Wolf, and B. G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 377–388, New York, NY, USA, 1996. ACM.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 16, pages 425–427. The MIT Press, 3nd edition, 2009.

[15] D. L. Davison and G. Graefe. Dynamic resource brokering for multi-user query execution. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 281–292, New York, NY, USA, 1995. ACM.

[16] C. Faloutsos, R. T. Ng, and T. K. Sellis. Predictive load control for flexible buffer allocation. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 265–274, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[17] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 296–305, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[18] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[21] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM.

[22] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. ACM.

[23] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[24] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272, 2011.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[26] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 130–137, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 268–277, New York, NY, USA, 1991. ACM.

[28] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[29] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[30] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance slos. *IEEE Trans. on Knowl. and Data Eng.*, 26(6):1447–1463, June 2014.

[31] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, Nov. 2015.

[32] J. Li, J. Naughton, and R. V. Nehme. Resource bricolage for parallel database systems. *Proc. VLDB Endow.*, 8(1):25–36, Sept. 2014.

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[34] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9(12):936–947, Aug. 2016.

[35] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[36] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[37] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, Feb. 2015.

[38] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 387–396, New York, NY, USA, 1991. ACM.

[39] H. H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 221–232, New York, NY, USA, 1994. ACM.

[40] R. J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific.

[41] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA, 2013. ACM.

[42] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, pages 1264–1275. IEEE Computer Society, 2011.

[43] J. Simão and L. Veiga. Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. *Comput. Syst. Sci. Eng.*, 28(6), 2013.

[44] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.

[45] P. Tembey, A. Gavrilovska, and K. Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[46] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[47] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[48] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria big data management and analytics system and cloud services. In *CIDR*, 2017.

[49] J. Wang and M. Balazinska. Toward elastic memory management for cloud data analytics. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, pages 7:1–7:4, New York, NY, USA, 2016. ACM.

[50] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Poster papers of the 9th European Conference on Machine Learning*. Springer, 1997.

[51] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. Reef: Retainable evaluator execution framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1343–1355, New York, NY, USA, 2015. ACM.

[52] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[53] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.