Elastic Memory: Bring Elasticity Back to In-Memory Big Data Analytics

Joo Seong Jeong, Woo-Yeon Lee, Yunseong Lee, Youngseok Yang, Brian Cho, Byung-Gon Chun
Seoul National University

Abstract

Recent big data processing systems provide quick answers to users by keeping data in memory across a cluster. As a simple way to manage data in memory, the systems are deployed as long-running workers on a static allocation of the cluster resources. This simplicity comes at a cost: elasticity is lost. Using today's resource managers such as YARN and Mesos, this severely reduces the utilization of the shared cluster and limits the performance of such systems. In this paper, we propose Elastic Memory, an abstraction that can dynamically change the allocated memory resource to improve resource utilization and performance. With Elastic Memory, we outline how we enable elastic interactive query processing and machine learning.

1 Introduction

Over the past decade, large-scale big data analytics has been widely adopted. Google's MapReduce [8], Apache Hadoop [16], and Dryad [11] were seminal in allowing analytics on data centers of commodity machines. The simple functional programming model combined with the runtime that supports elastic scale-out and fault-tolerant execution has spurred wide adoption of the technology. Distinct MapReduce/DAG jobs are often run together on a large shared cluster; each job runs on resource allocations given by the cluster's resource manager (e.g. YARN [18] and Mesos [10]). The resource manager enables MapReduce/DAG jobs to elastically share resource slices, improving peak job performance and providing high utilization of cluster resources [18].

The resource manager abstraction is great for MapReduce/DAG jobs, but new types of in-memory data processing do not fit well to this abstraction. We look at two types, interactive query processing and machine learning.

Recent query processing systems such as Impala [2] and SparkSQL [6] provide quick answers to user queries by keeping processed data in memory. As a simple way to manage data in memory, the systems are deployed as long-running workers on a static allocation of the cluster resources. This simplicity comes at a cost: elasticity is lost. The workers hold on to their resources even while they remain idle during periods without client queries. Using today's resource managers, this severely reduces the utilization of the shared cluster (the case for scale-in). In other cases, the workers may spill data to disks when they do not have enough memory resources. If memory resources could be expanded, these queries would be served in memory (the case for scale-out).

Recent machine learning systems such as Spark MLlib [5] perform iterative processing on statically allocated resources. A machine learning job typically consists of tasks, each of which processes partitioned state in memory. Job execution should consider the trade-off between computation and communication. If the job is computation heavy, it is better to allocate more memory in other machines to exploit computation parallelism (the case for scale-out). In contrast, if the job is communication heavy, it is better to shrink the number of machines to reduce communication overheads (the case for scale-in).

In this paper, we propose Elastic Memory, an abstraction that brings elasticity back to in-memory big data analytics. Elastic Memory provides a set of primitives for dynamically expanding and shrinking memory resources and splitting and merging state with a hook to set elasticity policies. We then discuss how Elastic Memory enables elastic interactive query processing and elastic machine learning.

2 Elastic Memory

Despite various kinds of data analytics workloads, such as batch processing, stream processing, query processing, and distributed machine learning, the way in-memory processing frameworks [2, 5, 6] execute such jobs can be characterized by the following common properties.

Master-slave pattern. Slaves process data under the control of a master, which manages execution of a job. Each slave and the master are deployed within its own container, which represents computing resources such as CPU cores and memory in a cluster allocated by a resource manager.

Data parallelism. To efficiently process large volumes of data, each slave performs in parallel the same computation on a different subset of the data.

In-memory caching. Data loaded from a distributed filesystem as well as intermediate data generated from previous computations is cached in memory, allowing multiple stages of reads and writes without incurring disk access.

Taking such characteristics into account, we propose

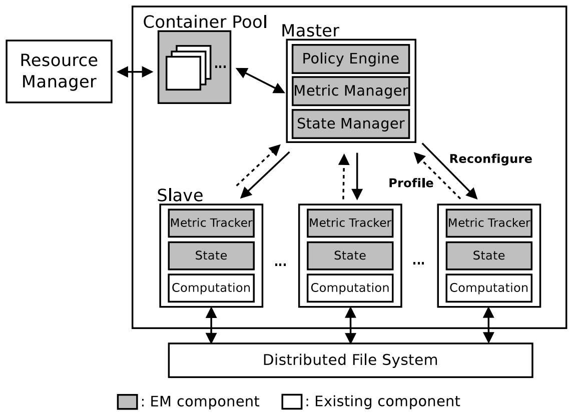

Figure 1: The EM architecture

Elastic Memory (EM) shown in Figure 1. EM adds new modules and abstractions to the existing model to enable elasticity. First, the state manager in the master manages and reconfigures state, an abstraction for reconfigurable in-memory data in a container. Second, the metric manager in the master manages metrics with the help of metric trackers in slaves. Third, the policy engine in the master enforces user policies. Fourth, to be able to quickly expand its resource capacity, EM keeps a pre-allocated pool of containers that can be quickly preempted for other jobs in the cluster to use and quickly reclaimed if needed.

2.1 State

State is the in-memory data within a container that is used and processed by a job. Figure 2 depicts how state is represented in EM. A container's state consists of user-defined atomic entities of elasticity called *units*. Each unit has a *type* representing its semantic meaning within the job computation. Units within a container are grouped into *subsets* of the same type. EM reorganizes state by transferring units that make up all or a part of a subset between containers.

2.2 Mechanism

2.2.1 Profiling

The metric tracker in each slave tracks metrics within its container and sends them to the metric manager in the master. The metric manager aggregates and processes the received metrics into a form which can be used by the policy engine. Users can configure not only app-specific metrics to be profiled by metric trackers, but also how they should be aggregated by the metric manager. For example, each metric tracker can be configured to send the number of state access requests per second to the metric manager, which can also be configured to compute the 5-second moving average of this metric.

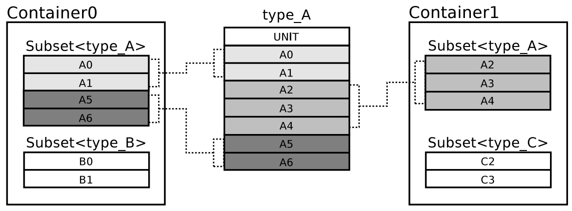

Figure 2: State representation. In-memory data of `type_A` is divided into three parts and distributed in two containers. Two parts reside in `Container0` and compose a subset, while the remaining part is in `Container1` as another subset. `Container0` also contains a subset of `type_B` which, together with the subset of `type_A`, forms the state of `Container0`.

2.2.2 Reconfiguring State

The policy engine compares the metrics processed by the metric manager with conditions specified in the user policy. If a condition is met, then corresponding elasticity actions are executed through the state manager. The state manager provides the following set of system primitives that are also depicted in Figure 3.

- `add (resource-spec)`: Allocate a new container as specified in the `resource-spec` that represents the resource of the container such as CPUs and memory.
- `delete (container-id)`: Release the container whose id is `container-id`.
- `resize (container-id, resource-spec)`: Resize the container whose id is `container-id` to the resources specified in `resource-spec`.
- `move (set<unit>, src-id, dst-id)`: Move `set<unit>` from the container whose id is `src-id` to the container whose id is `dst-id`.
- `checkpoint (container-id)`: Persist the state of the container whose id is `container-id` into stable storage.

Note that when releasing containers via `delete` or shrinking containers via `resize`, state in the slaves can be lost. To address this problem, the state manager can decide to either `checkpoint` such state, `move` it to other slaves or simply discard it depending on its recoverability and the system's status.

2.3 User Policy

The user policy is a list of rules, each of which consists of a condition and elasticity actions. A user can define a policy using the policy definition language in Figure 4.

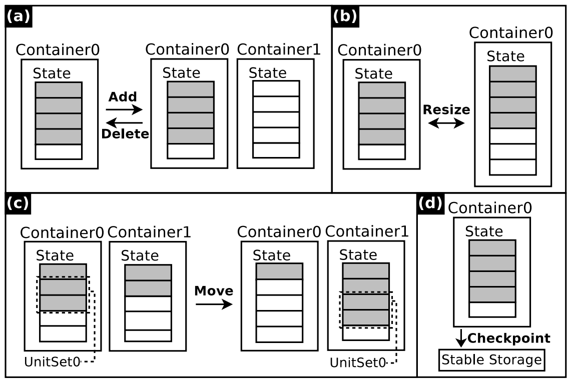

Figure 3: System primitives for handling state.
(a) `add (resource-spec)` / `delete(container1)`
(b) `resize(container0, resource-spec)`
(c) `move(unitset0, container0, container1)`
(d) `checkpoint(container0)`

Conditions are represented as conjunctions and disjunctions of predicates on metrics processed by the metric manager. When a condition is satisfied, the policy engine triggers the corresponding actions.

Actions are translated into lower-level system primitives provided by the state manager in Section 2.2.2. The following examples show how each action can be defined using the language in Figure 4.

- `Add` *resource-spec*: Allocate a container whose amount is specified in *resource-spec* by calling `add`.
- `Delete` ($c \Rightarrow c.idle\_time > 1$ min): Remove containers that have been idle for longer than one minute by executing `delete` on all such containers.
- `Resize` ($c \Rightarrow c.idle\_time > 1$ min) *resource-spec*: Change the memory size of idle containers to the amount as specified in *resource-spec* by calling `resize` on all such containers.
- `Merge` ($c \Rightarrow c.idle\_time > 1$ min) 2: Find idle containers and merge every two of them into one. Call `move` to transfer data units between containers. During the process, EM may decide to `delete` containers that have become empty.
- `Split` ($c \Rightarrow c.load > 0.8$) 2: Choose overloaded containers and split the state of each container into two containers by calling `move` to relocate data. EM may decide to `add` new containers in the process.
- `Migrate` ($c \Rightarrow c.load > 0.8$) ($c \Rightarrow c.load < 0.2$): Migrate data from busy containers to idle containers by calling `move`.

⟨*Policy*⟩ ::= list⟨*Rule*⟩

⟨*Rule*⟩ ::= ⟨*Condition*⟩, sequence⟨*Action*⟩

⟨*Condition*⟩ ::= ⟨*Predicate*⟩
  | '(' ⟨*Condition*⟩ ')'
  | ⟨*Condition*⟩ ∨ ⟨*Condition*⟩
  | ⟨*Condition*⟩ ∧ ⟨*Condition*⟩

⟨*Action*⟩ ::= `Add` ⟨*ResourceSpec*⟩
  | `Delete` ⟨*SelectFunc*⟩
  | `Resize` ⟨*SelectFunc*⟩ ⟨*ResourceSpec*⟩
  | `Merge` ⟨*SelectFunc*⟩ `factor`
  | `Split` ⟨*SelectFunc*⟩ `factor`
  | `Migrate` ⟨*SelectFunc*⟩ ⟨*SelectFunc*⟩

⟨*Predicate*⟩ is a statement that is true or false depending on the system state.

⟨*ResourceSpec*⟩ represents the amount of resources (e.g., CPU cores, memory, etc).

⟨*SelectFunc*⟩ selects containers that return true when *SelectFunc* is applied.

Figure 4: Policy definition language.

3 Elastic Interactive Query

When processing interactive queries, allocating too little memory increases query latency, while allocating too much causes excessive use of scarce memory resource. A naive approach would estimate the load and configure the memory resource accordingly before running the query. However, interactive queries run in iterations and produce intermediate data to be used and possibly cached for subsequent queries. This makes it very difficult to predict how much memory to use for each query. We show how EM obviates such prediction with dynamic state reconfiguration.

3.1 Representing Schema

In distributed in-memory query processing frameworks, a table is partitioned and cached in containers in a columnar format to exploit data locality. Since in this case each value of a row forms an atomic unit, we can naturally define it as the base unit of reconfiguration in EM.

3.2 Profiling Query Execution

The following are a few examples of metrics an interactive query processing framework can profile and use through EM.

Load. To ensure low latency, an in-memory framework should have enough memory resource capacity to keep data in memory. But load on slaves fluctuates for every query. EM can keep track of such load in the following way. First, individual metric trackers can track metrics such as container size, data size and requests for data per second. Second, the metric manager can use these metrics to compute an aggregate load for each slave.

Idle time. Holding onto resources even when one is not making good use of them hampers efficient use of cluster resources. In interactive query processing, this can happen when the user does not submit queries. We can address these issues using EM by collecting and aggregating metrics like last task execution time and last state access time of each slave.

3.3 User Policies For Query Execution

The following shows an example policy to improve the performance and resource utilization in the interactive query processing. *load* and *idle-time* are measured with the profiled metrics in Section 3.2. *top*(*metric*) denotes the container that has the largest value for the *metric*.

- Rule 1
  `Condition`: *average*(*load*) > 0.8
  `Action`: `Add` (*resource-spec*)

- Rule 2
  `Condition`: *idle-time* > 1 min
  `Action`: `Delete` (*top*(*idle-time*))

Rule 1 is applied when the average load is bigger than a threshold. To improve performance, EM expands its resource capacity by allocating new containers. Rule 2 is applied to containers who have been idle for longer than one minute. For efficient use of cluster resources, EM releases the container with the highest *idle-time*.

4 Elastic Machine Learning

Most distributed machine learning jobs start by loading data from disk, which is later on accessed in memory throughout the remaining job. Slaves run the algorithm independently on its portion of the data. The master aggregates the computation results and calculates a model. This model is broadcast to the slaves, and then the whole process is repeated; hence an iterative job.

Thus we can say that ML can be portrayed by a few aspects: slaves perform identical operations during each iteration and the training dataset never changes as the job progresses.

The execution logic does not drastically change as the job continues, which means even a single dynamic reconfiguration, if carried out correctly, can speed up execution. The benefit grows even more if many iterations remain, or in other words the reconfiguration is done early in the life span of the job. We illustrate how EM captures such insights and applies an optimized configuration to the currently running ML application.

4.1 Representing ML Data

ML algorithms accept a big training data set as input, and use it to build a model. The data consists of independent observations represented as rows of a table, where a row can be a single float, vector, or even a matrix depending on the algorithm itself. Each observation is atomic and thus we can define it as the base unit. In case each worker maintains its own version of the model such as in asynchronous systems [7], model partitions can also be defined as units.

4.2 Profiling ML Execution

The iterative characteristic of ML applications allows various metrics to be collected after each iteration. Several instances are shown below.

Iteration time. The overall performance of a distributed job is dependent on the slave with the worst performance, i.e. the slowest slave. To identify that slave, the EM system must be provided with the running time per iteration of each slave, which indicates their performance.

Computation and communication overheads. Even without a significantly slow slave, a machine learning application may have room for performance improvement by changing the number of slaves. EM can check the overheads of computation and communication to make a decision to balance out the two and find a number of slaves that improves performance.

4.3 User Policies for Machine Learning

The timing for the policy engine to check conditions specified in policies may differ depending on the ML execution model: every synchronization barrier between iterations is the ideal point for synchronous models such as BSP [12, 17], whereas for asynchronous models condition checking and actions are done in the background.

The example policy below shows how EM can contribute to optimizing the job configuration of ML applications. *iter-time*, *comp-time*, and *comm-time* are computed using the profiled metrics in Section 4.2. *big_outlier*(*metric*) computes whether there exists a value in the *metric* that outstands from the other values. *topK*(*metric*) and *bottomK*(*metric*) choose the containers that have the largest *K* and smallest *K* values for the *metric*, respectively.

- Rule 1
  `Condition`: *big_outlier*(*iter-time*) = true
  `Action`: `Migrate` (*top1*(*iter-time*), *bottom1*(*iter-time*))

- Rule 2
  `Condition`: *average*(*comp-time*) / *average*(*comm-*

time) > 5
`Action`: `Split` (*top1*(*comp-time* / *comm-time*), 2)

- Rule 3
  `Condition`: *average*(*comp-time*) / *average*(*comm-time*) < 0.2
  `Action`: `Merge` (*bottom2*(*comp-time* / *comm-time*), 2)

Rule 1 can detect stragglers that are taking a considerably longer time to finish iterations compared to other containers. The total job running time can be reduced by migrating data from the straggler to a faster container. Rules 2 and 3 cover cases in which computation and communication overheads are unbalanced. When computation is intense, Rule 2 kicks in and splits the data of the container with the most computation, lowering the global overhead. On the other hand, if communication is the bottleneck then Rule 3 is invoked and the distributed workloads of the two most communication-heavy containers are merged to one to diminish the network overhead.

Resource constraints. The assumption of limited resources is actually a common scenario in distributed computing, either by hardware specifications or because of other users sharing the same resources. Under this situation, the metric of available resources must be taken into account when deciding resource-related actions to be invoked. Although it is debatable whether to give precedence to user-defined policies over resource constraint policies, there exist circumstances where user-defined policies are impossible to carry out due to the lack of additional resources. A job might be running with only two slaves and need to execute `Add` to achieve optimal performance, but the cluster may not have more slaves available because other jobs are using many nodes.

5 Related Work

Discardable Distributed Memory (DDM) [15] proposes the use of memory, obtained from a resource manager, as fast external storage for running jobs. EM instead reconfigures state directly within running jobs. In addition to supporting memory elasticity, this abstraction provides the means to tune computation parallelism and communication overheads.

Commercial cloud offerings [1, 3, 4] allow developers to configure custom rules for triggering expansion or contraction of the number of VMs for serving systems. Escape Capsule [13] proposes a mechanism to capture and automatically migrate per-session state that spans all layers of the software stack. FreeFlow [14] splits and rebalances flow-specific state among virtual middlebox replicas. While also enabling elasticity, the main goal of these systems is load balancing, whereas EM can improve the runtime performance of in-memory big data processing frameworks using mechanisms and policies.

A different line of work explores techniques to achieve elasticity in a distributed computing environment. ElasticOS [9] enables a process to stretch its associated resource boundaries across multiple machines, obviating the need to accommodate cluster programming models. The motivation is clearly different from that of EM, which aims to enable elastic use of in-memory data for distributed frameworks.

6 Conclusion

We propose Elastic Memory, an abstraction that provides mechanisms and policies for dynamically expanding and shrinking memory resources and splitting and merging memory state. We believe that the Elastic Memory architecture enables new exciting execution modes that bring elasticity back to in-memory big data analytics and offers interesting research opportunities to consider elasticity as a first-class citizen in in-memory big data analytics.

Acknowledgments

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea and Microsoft Research, under ICT/SW creative research program supervised by the NIPA (National ICT Industry Promotion Agency) (NIPA-2014-H0510-14-1022). Brian Cho was supported by the Institute of New Media and Communications, Seoul National University, grants No. 0423-20140007 and No. 0423-20140067.

References

[1] Amazon web services. `http://aws.amazon.com`.

[2] Cloudera impala. `http://impala.io`.

[3] Google cloud. `http://cloud.google.com`.

[4] Microsoft azure. `http://azure.microsoft.com`.

[5] Sparkmllib. `https://spark.apache.org/mllib/`.

[6] Sparksql. `https://spark.apache.org/sql/`.

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[9] A. Gupta, E. Ababneh, R. Han, and E. Keller. Towards elastic operating systems. In *HotOS*, 2013.

[10] 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 *NSDI*, 2011.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys*, 2007.

[12] F. Loulergue, F. Gava, and D. Billiet. Bulk synchronous parallel ml: modular implementation and performance prediction. In *ICCS*. 2005.

[13] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Escape capsule: Explicit state is robust and scalable. In *HotOS*, 2013.

[14] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[15] C. N. Sanjay Radia, Arpit Agarwal. Support memory as a storage medium, 2014. `https://issues.apache.org/jira/browse/HDFS-5851`.

[16] The Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org`.

[17] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[18] 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 *SOCC*, 2013.