Towards Automatic Memory Tuning for In-Memory Big Data Analytics in Clusters

Aris-Kyriakos Koliopoulos, Paraskevas Yiapanis, Firat Tekiner, Goran Nenadic, John Keane
School of Computer Science, University of Manchester[1]
aris.koliopoulos@drivetribe.com, {paraskevas.yiapanis, firat.tekiner, gnenadic, john.keane}@manchester.ac.uk

***Abstract*—Hadoop provides a scalable solution on traditional cluster-based Big Data platforms but imposes performance overheads due to only supporting on-disk data. Data Analytic algorithms usually require multiple iterations over a dataset and thus, multiple, slow, disk accesses. In contrast, modern clusters possess increasing amounts of main memory that can provide performance benefits by efficiently using main memory caching mechanisms.**

Apache Spark is an innovative distributed computing framework that supports in-memory computations. Even though this type of computations is very fast, memory is a scarce resource and this can cause bottlenecks to execution or, even worse, lead to failures. Spark offers various choices for memory tuning but this requires in-depth systems-level knowledge and the choices will be different across various workloads and cluster settings. Generally, the optimal choice is achieved by adopting a trial and error approach.

This work describes a first step towards an automated selection mechanism for memory optimization that assesses workload and cluster characteristics and selects an appropriate caching scheme. The proposed caching mechanism decreases execution times by up to 25% compared to the default strategy and reduces the risk of main memory exceptions.

***Keywords*-Spark; Distributed Systems; Data Analytics; Big Data; Memory**

I. Introduction

Distributed systems provide an infrastructure that can enable efficient and scalable Big Data Analytics [1], [2], [3]. Such systems, made up of organized clusters of commodity hardware, process large volumes of data in a distributed fashion. Hadoop [4], an open source implementation of MapReduce [5], is the most widely-used platform for large-scale distributed data processing. Hadoop processes data from disk which makes it inefficient for data analytics applications that usually require iteration. Spark [6] is a more recent distributed framework that works with Hadoop and provides in-memory computation allowing iterative jobs to be processed much faster making it a more suitable base for data analytics.

The in-memory nature of Spark allows the framework to process data up to 100x faster than Hadoop's disk-based MapReduce paradigm on occasions, especially when involving data analytics [7]. Spark takes advantage of an abstraction called Resilient Distributed Datasets (RDDs) that allows data to be cached in memory and reused when required. This enables significant performance improvements compared to reading and writing data from disk. Despite the obvious benefits of performance, memory remains a limited resource and must be handled with care as the user cannot precisely predict how much memory will be needed for an application for the following reasons: (1) an RDD can consume a factor of 2x-5x more space than the raw data inside their fields, (2) the number of objects generated during the program's execution is data-dependent.

Spark's default caching mechanism is to store data only in memory. This mechanism is usually the most efficient as long as all data can fit into main memory. Even if all data can fit into memory, task failures from out-of-memory exceptions may still occur when memory is allocated faster than can be handled by garbage collection. For these reasons Spark offers various storage level options. Unfortunately, the choice of the optimal storage level requires in-depth systems-level knowledge, can be found only through experimentation, and will be different across various workloads and cluster settings. Here we address this problem by proposing an automatic storage level selection mechanism based on problem size and cluster characteristics. This paper makes the following contributions:

- It explores the memory impact of various workloads using different storage levels for in-memory distributed systems;
- It describes a framework that automatically selects the optimal storage level based on data and cluster characteristics;
- It applies the automated framework on previously unseen data decreasing execution times by up to 25% as well as reducing failures compared to the default storage level.

Although this work utilizes Spark as its target platform, the techniques described are applicable to any in-memory distributed platform that offers similar memory storage characteristics to Spark.

[1]Koliopoulos' current address: DriveTribe, London, UK; Yiapanis' current address: School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, UK.

II. Motivation for an automated caching mechanism

A. Caching Overheads

An important feature of Spark is the ability to cache datasets in memory across operations. RDDs [7] are a distributed main memory abstraction, implemented in Spark, that enable users to perform in-memory computations in large systems.

Spark RDDs are represented in memory as distributed Java objects. These objects are very fast to access and process, but they may consume up to 5x more memory than the raw data of their attributes. This overhead can be attributed to the meta-data that Java stores alongside the objects and the memory consumed by the object's internal pointers. For example, a `String` class introduces 40 bytes of pure overhead associated with storing characters separately as `Character` classes and keeping properties such as string length. Consequently, a 10 character String requires 60 bytes of main memory.

Spark's default caching option is to store Java objects directly in memory. With this option a fraction of the memory is used for RDDs with the rest used to store any objects created during execution. The default caching option loads the cache fraction of the executors until saturation and then recomputes additional partitions on demand as and when required. In our experiments (Section V), this method was found to be unsuitable in practice. In many cases, new RDD partitions are allocated memory faster than the Garbage Collector (GC) is able to discard older partitions. Consequently, memory leaks and the tasks fail on main memory exceptions.

B. Spark Storage Options

Spark offers a series of storage configurations to tackle such overheads. These configurations define whether Spark should cache objects in-memory, on-disk or use a combination of the two. These levels can be further customized by the use of different serialization and compression libraries: Spark can be configured to use either the built-in Java serialization or the Kryo [8] serialization libraries. Finally, Spark's codecs can further reduce the memory footprint of RDD objects by compressing serialized byte arrays.

The storage configurations that Spark provides introduce a trade-off between memory usage and CPU efficiency. The best option in terms of CPU efficiency is to store the data decompressed in main memory. If the data does not fit in memory then serialization and compression can be used to minimize the space requirement. These options, however, may introduce performance penalties. If all else fails the data can be stored on disk, as with Hadoop. However, this is likely to be the least CPU efficient option and should be avoided where possible.

Given the above, it is clear that in order to find the best storage configuration for their dataset, a user must utilize a trial and error experimental methodology. This is largely impractical as the result will vary across different datasets and clusters as well as requiring systems-level evaluation knowledge. In this work we experiment with the different storage configurations that Spark offers and propose an automated mechanism for the most appropriate storage level based on a few parameters such as cluster memory size and data size.

III. Exploring the effectiveness of Serialization and Compression

This section provides experimental results on the space overheads of RDDs and the effectiveness of serialization and compression. According to [6], Kryo offers performance improvements over Java serialization.

A number of different dataset samples from the UCI Machine Learning Repository [9] and Stanford SNAP [10] were evaluated in order to measure memory overheads for different categories of datasets. Tables I and II display the RDD size using memory only, Java serialization, Kryo serialization, Java serialization and compression, and Kryo serialization and compression as a percentage of the original on-disk value.

Dataset	Data Type	Memory Only	Java Serialization	Kryo Serialization
Susy	Structured Numeric	198.00%	91.60%	91.20%
Higgs	Structured Numeric	204.00%	96.00%	95.60%
Generated	Structured String Sparse	208.16%	95.92%	95.92%
USCensus	Structured Numeric	234.88%	100.87%	101.45%
Record Linkage	Structured String	330.00%	105.00%	103.00%
KDD data '10	Structured String	250.00%	103.00%	102.00%
adult	Structured String	242.11%	100.00%	99.47%
supermarket	Structured String Sparse	221.05%	104.21%	103.16%
Million Song	Structured Numeric	206.25%	102.50%	102.50%
Wiki articles	Text	220.59%	104.71%	141.12%
Social circles: Twitter	Graph	500.00%	107.27%	101.82%
Epinions social network	Graph	629.63%	107.41%	100.00%
Google Web Graph	Graph	527.27%	100.00%	93.64%

Table I
RDD size using various caching methods as a percentage of the original on-disk value

Dataset	Data Type	Memory Only	Java Ser. & Compression	Kryo Ser. & Compression
Susy	Structured Numeric	198.00%	52.00%	51.60%
Higgs	Structured Numeric	204.00%	50.80%	50.80%
Generated	Structured String Sparse	208.16%	22.45%	22.45%
USCensus	Structured Numeric	234.88%	38.08%	38.08%
Record Linkage	Structured String	330.00%	44.00%	43.00%
KDD data '10	Structured String	250.00%	47.00%	47.00%
adult	Structured String	242.11%	53.16%	53.16%
supermarket	Structured String Sparse	221.05%	26.32%	26.32%
Million Song	Structured Numeric	206.25%	55.00%	54.38%
Wiki articles	Text	220.59%	44.12%	44.12%
Social circles: Twitter	Graph	500.00%	44.55%	43.64%
Epinions social network	Graph	629.63%	48.15%	44.44%
Google Web Graph	Graph	527.27%	40.00%	38.18%

Table II
RDD size using various caching methods as a percentage of the original on-disk value

As the results demonstrate, uncompressed main memory footprints vary greatly and can reach up to 600% of the original dataset. However, serialized objects demonstrate footprints close to the on-disk values in all cases. Compression demonstrates an additional 50% and 75% reduction for dense and sparse datasets respectively. In order to further

assess the benefits of caching, a performance analysis of different caching strategies is required.

Figure 1 presents the average execution time overhead of each caching strategy. Serialization and compression mechanisms present significant memory footprint reductions offset by a small performance penalty.

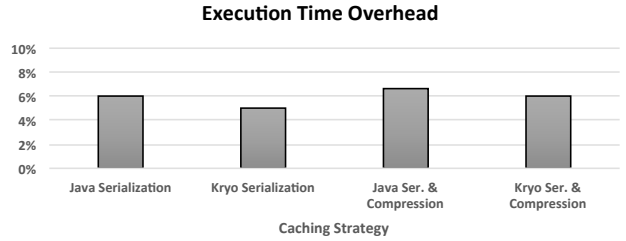

Figure 1. Execution time overhead for different caching strategies in Spark

IV. Automatic caching strategy selection

Spark offers various caching options and enables a user to implement custom caching strategies. However, this practice demands expert knowledge of the underlying platform and extensive experimental evaluation of the different options. In order to tackle this issue, a custom strategy has been implemented based on the insights obtained from Section III. This strategy is triggered in cases where the user does not explicitly specify a caching mechanism. The selection process is illustrated in Figure 2.

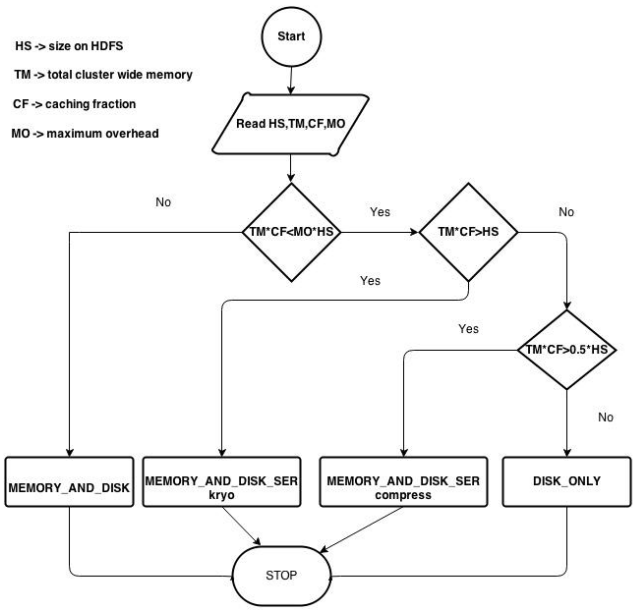

Figure 2. Caching configuration selection process

This process uses the file's size on the Hadoop Distributed File System (HDFS), the total cluster wide memory, the caching fraction of the executors and the maximum overhead as input parameters. In large text-based files the overhead was experimentally computed and in the worst case scenario it approaches 500%. The Apache Spark documentation mentions the same worst-case overhead without specifying a dataset type. Consequently, the algorithm uses this value as default, but allows a user to specify the overhead parameter that better relates to their data.

If the cluster-wide executor cache memory is enough to absorb the dataset in the worst case, default caching is used. Uncompressed objects are faster to access and the CPU overhead of serialization is avoided. In the case where the cache approaches the dataset size, serialized objects are preferred as they demonstrate memory footprints which are equivalent to the original file size. Kryo serialization proved to be more efficient and it is used as the default option. This process introduces serialization overhead but decreases GC overhead and enables up to 5x more data to be stored in-memory. Compression is additionally used to tackle cases where at least 50% of the on-disk data can be cached. Compression introduces an additional CPU overhead but further reduces memory footprints by 50% compared to serialization. Finally, if the dataset is twice as large as the available cache (and thus the compression mechanisms cannot ensure that full caching is possible) then disk caching is used.

V. Experimental evaluation

Our automatic caching strategy has been implemented as a single Scala class and integrated into Spark. When a task is submitted, the input parameters are read from the application context and the algorithm selects a Storage Level and decides on the use of serialization and compression automatically.

The proposed strategy was tested against the default caching strategy in a number of experiments. The workload consisted of a 20GB dataset and the FP-Growth algorithm [11] on an 8-core cluster (two 4-core m3.xlarge Amazon EC2 instances). The experiment was repeated using one thousand (1K) and four thousand (4K) partitions on 15GB and 2GB (to account for multi-tenant environments [12], where memory is often limited) of cache memory.

Table III displays the elapsed execution times of Spark's default caching strategy in comparison to our custom caching strategy.

Table IV shows the percentage of tasks that failed across the experiments when using Spark's default caching strategy in comparison to our strategy.

The automatic strategy decreases execution times by up to 25% and, in these experiments, eliminates failures caused by insufficient main memory.

In the default strategy, the objects are cached deserialized (uncompressed Java object structures). This forces the GC to recursively traverse the object hierarchy before evicting

Partitions	Cache Size	Default Strategy	Custom Strategy
4K	15GB	1365	1290
1K	15GB	1620	1405
4K	2GB	1919	1448
1K	2GB	Failed	1865

Table III
Execution times (seconds) for default cache strategy vs. automatic strategy

Partitions	Cache Size	Default Strategy	Custom Strategy
4K	15GB	0.00%	0.00%
1K	15GB	17.00%	0.00%
4K	2GB	6.25%	0.00%
1K	2GB	100.00%	0.00%

Table IV
Failed tasks for default cache strategy vs. automatic strategy

unreferenced objects. Each time memory runs out, a set of old partitions is garbage collected and a new set is fetched from disk. As the size of cache memory decreases, this procedure is triggered more frequently. If this process is slower than memory allocation, tasks fail due to main memory exceptions. Larger partitions contain larger object hierarchies and thus increase GC overhead (GC has a larger workload when it is triggered). This is the reason why larger partitions demonstrated increased fail rates in the experiments that used the default strategy.

The automatic strategy achieves better performance by decreasing both the GC overhead and the frequency of the procedure. The selection algorithm assesses the available memory and whether the partition replacement mechanism would be triggered by the given dataset. If this is the case, the algorithm activates and configures serialization and compression mechanisms.

Serialized/compressed objects are represented as an array of bytes. The GC discards these as a single entity, regardless of the number of objects they encapsulate. This avoids the cost of searching object hierarchies for unused objects. Additionally, these objects consume up to 10x less memory, thus freeing up to 10x extra in-memory storage and hence significantly decreasing the frequency of the partition replacement mechanism. However, this process, in turn, introduces CPU overhead due to serialization/compression.

The results demonstrate that the cost of serialization is lower than the cost of partition replacement. Additionally, in the experiments, reducing the GC overhead achieves 100% task completion even in cases where the default strategy fails repeatedly and aborts the execution.

VI. Conclusions & Future work

This work has considered automation of the trial and error method to select the best caching configuration for big data workloads running on cache-based systems such as Spark. For this purpose multiple caching strategies have been experimentally evaluated. The analysis demonstrates that serialization and compression mechanisms are able to significantly decrease memory footprints with small performance penalties.

The default caching strategy of Spark was found to be inefficient in cases where data does not fit into memory. Analysis of results produced insights about the trade-offs between different caching strategies, and an automatic caching strategy selection algorithm was proposed, based on these insights. Evaluation of this algorithm showed improved performance of up to 25% compared to the default strategy as well as reduced risk of main-memory exceptions. This behavior is attributed to decreasing both the garbage collection overhead and the garbage collection frequency.

Our results are preliminary but positive. Future work aims to perform in-depth investigation of the GC behavior and experiment with a greater variety of workloads and configurations.

Acknowledgment

The work was supported by an IBM Faculty Award in Big Data Engineering. The authors wish to thank Dr Mark Hall at the University of Waikato for his advice and encouragement.

References

[1] "Apache Mahout," http://mahout.apache.org/.

[2] "MLib," https://spark.apache.org/mllib/.

[3] A. K. Koliopoulos, P. Yiapanis, F. Tekiner, G. Nenadic, and J. Keane, "A Parallel Distributed Weka Framework for Big Data Mining Using Spark," in *In IEEE Intl. Congress on Big Data (BigData Congress)*, 2015.

[4] "Apache Hadoop," http://hadoop.apache.org/.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Com ACM*, pp. 107–113, 2008.

[6] "Apache Spark," https://spark.apache.org/.

[7] 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 *NSDI*, 2012.

[8] "Kryo," https://github.com/EsotericSoftware/kryo/.

[9] "Machine Learning Repository," https://archive.ics.uci.edu/ml/datasets.html.

[10] "Stanford Network Analysis Project," http://snap.stanford.edu/.

[11] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns Without Candidate Generation," in *In ACM SIGMOD Intl Conference on Management of Data*, 2000, pp. 1–12.

[12] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, "A Framework for Native Multi-Tenancy Application Development and Management," in *In Intl Conference on Enterprise Computing, E-Commerce, and E-Services*, 2007, pp. 551–558.