Code and Data Transformations to Address Garbage Collector Performance in Big Data Processing


Published in:
25TH IEEE International Conference on High-Performance Computing, Data and Analytics

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright 2018 IEEE. This work is made available online in accordance with the publisher’s policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person’s rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Open Access
This research has been made openly available by Queen's academics and its Open Research team. We would love to hear how access to this research benefits you. – Share your feedback with us: http://go.qub.ac.uk/oa-feedback
Code and Data Transformations to Address Garbage Collector Performance in Big Data Processing

Damon Fenacci, Hans Vandierendonck and Dimitrios S. Nikolopoulos
Centre for Data Science and Scalable Computing
Queen's University Belfast
{d.fenacci, h.vandierendonck, d.nikolopoulos}@qub.ac.uk

Abstract—Java, with its dynamic runtime environment and garbage collected GC memory management, is a very popular choice for big data processing engines. Its runtime provides convenient mechanisms to implement workload distribution without having to worry about direct memory allocation and deallocation. However, efficient memory usage is a recurring issue. In particular, our evaluation shows that garbage collection has huge drawbacks when handling a large number of data objects and more than 60% of execution time can be consumed by garbage collection. We present a set of unconventional strategies to counter this issue that rely on data layout transformations to drastically reduce the number of objects, and on changing the code structure to reduce the lifetime of objects. We encapsulate the implementation in Apache Spark making it transparent for software developers. Our preliminary results show an average speedup of 1.54 and a highest of 8.23 over a range of applications, datasets and GC types. In practice, this can provide a substantial reduction in execution time or allow a sizeable reduction in the amount of compute power needed for the same task.

Keywords—garbage collection; spark; big data; java

I. INTRODUCTION

Modern “Big Data” computing frameworks are developed using managed languages. Systems such as Hadoop [1], Spark [2], Hyracks [3], Dryad [4] or Naiad [5] are written in Java, C#, Scala or Python and run on top of the Java Virtual Machine (JVM) or the .NET Common Language Runtime (CLR). These languages provide a number of advantages, i.e. garbage collected memory management, strong typing and platform independence, that contribute to improving the productivity of software developers and data analysts.

Garbage collectors are highly complex algorithms that have been tuned experimentally to meet the memory allocation patterns of common applications [6]. Applications in data analytics, however, present a thoroughly different memory allocation pattern where established garbage collectors do not perform well. For instance, Nguyen et al [7] have shown that there is a cyclic memory allocation pattern, where large amounts of data are incrementally allocated and subsequently freed en masse. As such, for large datasets, garbage collection can easily exceed 20% of the execution time and in excess of 50% for some applications [8]–[11].

The literature presents several solutions to this problem. We find, however, that none of them are satisfactory as they are fundamentally inconsistent with the goals of managed languages. One approach is to explicitly allocate large amounts of data outside of the garbage collected heap [7], [10], [12]. Software developers are then tasked with explicit memory deallocation. These techniques, while performant, are prone to the same issues of low-level, C-style memory allocators (memory leaks, double frees, dangling references) and lose the key benefits of managed languages. Another approach is to merge small objects in buffers and access the data in a low-level, binary format [8]. This technique furthermore requires code changes to access the merged objects. These techniques significantly reduce the overhead on the Garbage Collector (GC) but they involve code changes that cause overhead for the developer. Some approaches, moreover, require modification of the virtual machine and are not applicable to off-the-shelf environments.

This paper proposes a different approach that retains the benefits of managed languages and does not require changes in the virtual machine. The key observation behind this work is that garbage collection issues are an immediate consequence of the method and timing in which data structures are laid out in memory. Languages like Java and Scala represent data in an object-oriented format, where each object is accessible through a reference. Composite data types, recursively, hold references to their components. As such, composite data types are almost invariably stored as linked data structures. Every component in these linked data structures appears as an individual heap object to the garbage collector. We will demonstrate that this linked data representation is the root cause of high garbage collection overhead.

In order to alleviate the pressure on the garbage collector, we aim at storing large data sets in a minimal number of heap objects as well as shorten the lifespan of small temporary objects. The proposed solution is twofold: store large data sets in consecutive memory locations, using a non-linked layout and iteratively examine a small amount of data at a time, discarding it as soon as it has been processed.

We observe that data types used in data analytics typically
allow consecutive layout, also in object-oriented languages. For most data analytics problems, data sets have homogeneous types, i.e., all data elements are integers, or they are all vectors of floating-point numbers, etc. We discuss how a consecutive layout of homogeneous data sets can be achieved using regular Scala programming and show its application in the context of Spark.

We moreover observe that in many applications data does not need to be retained in memory for long periods of time. Iterators can be used to read data on demand to avoid loading whole data files in memory. Although most Big Data platforms are built based on this principle, often this doesn’t apply when reading data from multiple files. We show how the use of iterators to process blocks of data can shorten their lifespan and dramatically reduce the time taken by the GC to process them.

Our implementation completely hides the complexity of the data layout in re-usable libraries (MLlib and GraphX, to be precise) and allows developers to take advantage of its benefits with minimal changes in their code. Our results show that we can achieve an average speedup of 1.95 with the highest speedup exceeding 4x. These speedups are comparable to prior work but do not require specialized virtual machines or unconventional memory models [10], [12].

II. RELATED WORK

GC have been the subject of extensive research for decades [13]. Over time, GC have become concurrent, parallel and more performant [14]–[19]. Overall, modern memory managed garbage collectors are tracing GC [20]. They trace live objects in the heap by following references starting from a set of root references. In a second step they collect all non-marked objects. In addition, they divide the heap into two separate regions for young and old generations [21] and traverse them separately. Minor GC runs (young generation) happen more often and collect most objects. The few remaining long-lived objects are moved to the old generation. Only when minor GC runs cannot recycle dead objects, a full GC run is performed, which scans the old generation as well. The current implementation of Oracle’s JVM provides four types of GC: generational GC, SerialGC, ParallelGC, Concurrent Mark and Sweep and G1 GC [22], [23] (see Table I).

The time complexity of tracing garbage collection is linear in the number of live heap objects [24]. GC overhead also depends on how often the GC is invoked, which is mostly a property of the workload. [8], [11] show that Big Data applications create a very large number of long-living objects. In these circumstances, generational GCs frequently trigger more expensive passes over the “old” generation without freeing much memory [7].

A wide variety of computational models and systems for big data are implemented in a managed language [1], [3]–[5], [27]–[29], making GC performance a recurring issue. Prior work has attempted to address GC performance issues by storing large data sets in special memory areas, out of reach of the GC [7], [10], [12]. Allocation in these special memory areas is controlled by the programmer, e.g., using Java annotations. JVM extensions are required to manage these memory areas and to make them cooperate with garbage collected memory. Nguyen et al [10] observe that many objects share the same lifetime and use region-based memory management to control allocation and deallocation of these objects while keeping them out of the garbage collector’s scope. Broom [12] distinguishes “data” objects that hold the data set from “control” objects that are created as a side effect of the operation of JVM-based languages. While GC are well-optimized for control objects, data objects are stored in native memory areas to avoid garbage collection overhead. Similarly, Yak [7] introduces the concept of epochs to manage storage of data objects. Following the same guided memory-management idea, Kedia et al [30] introduce a keyword to explicitly release objects in type-safe languages such as Java or C# and take advantage of the exception mechanism to handle references of objects that have been declared dead.

These solutions achieve good GC performance improvements but require the developer to explicitly manage non-GC memory using some programming interface to the JVM, which in some proposals may introduce memory management issues typical of machine-level programming languages. These solutions moreover depend on core JVM modifications. In contrast, our solution achieves comparable results that are fully interoperable with off-the-shelf JVMs and do not diminish the strong safety and security benefits provided by the JVM.

Bu et al [8] minimize the number of live objects by merging them in byte buffers either manually or by means of annotations. Their implementation resembles Spark’s in-memory serialization [2]. As such, the approach by Bu et al ties in nicely with existing functionality. However, as with serialization, an important processing overhead has to be taken into account. Our solution achieves similar performance gains without the serialization overhead and negligible code modifications.

Our proposal pursues similar goals as object inlining [31], [32]. However, it differs from object inlining in several salient aspects. Object inlining attempts to reduce the overhead caused by indirections by converting linked objects into a single object: objects referenced only once may be inlined in the object that holds the reference to them. To keep its memory model simple, the JVM does not natively
support object inlining. Moreover, object inlining requires complex runtime infrastructure and is triggered heuristically. Our approach overcomes these problems by changing the data layout to mimic inlining of complex data structures such as array of objects, which are too application-specific to be handled by generic compilers, while still avoiding modifications of the JVM and its memory model. Furthermore our solution does not require the programmer to review one’s code as in [33] but merely switch to another library.

Finally, layout transformations involving structure-of-arrays vs array-of-structures has been extensively investigated in the past, in particular in relation to improving cache efficiency at compilation level [34] [35] and for GPU programming [36] [37]. We take advantage of the same kind of transformations for a different purpose, namely reducing the number of references to ease the work of the GC.

III. PROBLEM DESCRIPTION

We explain the issues that incur in garbage collection for Spark using the MLlib implementation of term frequency-inverse document frequency (TF-IDF), an important information retrieval and text mining algorithm [38]. Given a set of documents (corpus), TF-IDF computes a value for each word in the corpus that reflects how important the word is within the document it appears in. The value increases proportionally to the number of times a word shows up in a document but is adjusted by how often the same word is found in all documents.

Memory-related performance issues have previously been flagged for the MLlib/Apache Spark TF-IDF algorithm [39]. We analyze these issues using datasets of varying document and term counts (see Table II, Section VI). Details of our evaluation platform are described in Section VI-A. We use the JVM’s built-in GC log functionalities to examine the GC throughput, defined as the percentage of time spent processing real application transactions vs. the time spent in GC activities. As a rule of thumb, throughput should exceed 95% as applications typically freeze up during GC runs [40]. What stands out (Figure 1) is a very low throughput, in particular for old-generation GC runs, for corpuses with large files like Artificial (18.9% throughput) or WestburyLab (40.5% throughput). A closer look at the former shows that the average time spent by a GC run is 34.6 s with a worst case of 177 s, an unacceptable amount by any standard. Furthermore, we observe from Figure 2 that GC runs with very long duration are scattered over the execution with a slight accumulation at the beginning and at the end.

In order to gain insight into what is happening in the heap and the reason for such a bad GC performance, we analyze the number of live objects in the heap over time. We use hprof [41], a Java heap profiling tool that provides information about the number and type of objects stored in the heap. It works by interrupting the JVM and scanning the heap. We run hprof at regular intervals of 1 second for the duration of TF-IDF with the Artificial dataset. Figure 3 shows the number of instantiated objects in the heap (blue) and total heap size (yellow) during the execution of the application. These measurements are always taken after a full GC run, i.e. they correspond to the heap usage of live data. These graphs show that a large number of relatively small objects are created which takes much of the available heap space for most of the run. While it is expected that the objects created by the 2 GB corpus of the Artificial dataset do not fill the 80 GB of available heap space, it is surprising that the total heap size grows to almost 20 times the size of the data set. This effect is explained by the handling of Strings in Java, which creates two objects: a descriptor and the actual array of characters. The descriptor stores a sizable amount of extra data and the characters are stored as wide, two-byte characters. As such, an average word in the English language counting 5 characters takes 48 bytes in memory, almost a factor of 10 more. This, however, is not the main problem for the garbage collector. The main issue for garbage collection results from a large number of long-lived objects. If given enough memory, Spark tries to keep objects in memory during computation to avoid serializing or recomputing them at a later stage, in particular if objects have to be reused, as is very often the case. Hence objects are retained over garbage collection runs and quickly moved from the young to the old generation by the GC. As a consequence, the old generation of the heap is constantly crowded, full garbage collections run more frequently but no data is released as most of it is alive for the full run.

It is fundamental to the operation of GC that overhead grows with the number of live objects. The first step of a tracing GC run consist in the marking phase. During this phase the GC traverses the whole object graph starting from object references in the thread stack and in global variables. All reachable objects are marked [13]. The time taken by this process is proportional to the number of references in the heap. As TF-IDF stores every word in the input using separate objects, the GC needs to mark a large number of objects, which can take several seconds to complete.

The second step of the GC is to free up unused space. This can be achieved through compaction or through copying live objects to a new space. In either case, time complexity is again proportional to the number of live objects. The number of live objects grows to nearly one billion (Figure 3), while the data set size is only 1.33 GB. There is clearly inefficiency in the number of created objects. Although we selected a large heap to ensure that Spark does not spill data sets to disk, our tests and previous literature [7], [12] agree that this behavior is independent of the heap’s size.

Spark provides alternative ways to store data in memory, i.e. as serialized Java objects in the main heap and off-heap (Spark storage levels MEMORY_ONLY_SER and OFF_HEAP). This option is available mainly to increase space efficiency at the expense of performance but could
in principle have a positive impact on the GC performance. Unfortunately, our tests, performed with the ideal setup presented in VI-A, show that, even with poor base GC performance and using a fast serializer (e.g., KryoSerializer), these versions are performing significantly worse than the non-serialized ones, e.g., with average slowdowns of 1.16 for MEMORY_ONLY_SER and 1.05 for OFF_HEAP with TF-IDF with the datasets introduced earlier over the default MEMORY_ONLY that doesn’t provide any serialization. This suggests that, with standard serialization, the overhead introduced completely hampers any performance benefit gained by the reduced number of objects.

IV. APPROACH

Spark runs on top of the JVM and provides APIs for Java, Scala and Python that are centered on a common data structure called Resilient Distributed Datasets. The Resilient Distributed Datasets (RDD) is a read-only collection of objects, often backed by array-like structures, that can be partitioned and assigned for processing by specific nodes or cores in the cluster (Figure ??).

Our approach consists of three different techniques, all aimed at reducing the pressure on the GC:

- transforming arrays-of-structures into structures-of-arrays to reduce the number of objects in the heap
- making use of indexes pointing at large data structures as a replacement for object creation
- iterating over data elements to keep their lifetime short

The first and second techniques rely on the fact that the JVM provides support for 8 primitive types—byte, short, int, long, char, float, double and boolean—that are mapped directly into memory. Furthermore, when instantiating arrays of native types, data elements are stored inline, back-to-back. Our goal is to transform existing RDD structures of references to objects into RDD of native types thus avoiding the creation of objects.

Our third technique transforms the code to avoid loading big blocks of data in memory. To do so we utilize iterators, which provide an elegant way of processing data elements singularly, allowing the GC to dispose of them quickly.

Finally we also aim at encapsulating the code for each technique into Spark’s libraries thus completely hiding the code to handle this transformation from the application developer. Based on current Spark’s MLlib and GraphX libraries, we reckon that we would be able to apply one or more techniques to the vast majority of their functions. In this work we take four of them as representative sample.

A. Array-of-structures (AOS) vs. structure-of-arrays (SOA)

Our first data-layout approach exploits the interchangeability between array-of-objects and object-of-arrays for objects that are homogeneous, i.e., of the same base type and no sub-type, and made of fields of primitive types. The second constraint is not strictly necessary but facilitates the layout transformation process. Our experience and the recent Dataframe evolution taken by Spark [42] tell us that RDDs already follow these rules implicitly. In other words Spark applications start building homogeneous datasets consisting of simple, primitive-typed data and keep RDDs homogeneous throughout the computation.

We propose the transformation shown on the right-hand-side of Figure 4 (Structure-Of-Arrays). Starting from a collection of objects with fields $f_1, f_2, ..., f_n$ (left-hand side) we modify the layout to produce multiple collections $c_1, c_2, ..., c_m$, one per field (right-hand side). The advantage of this transformation lies in the reduced number of objects. If in the original structure (left) there is one object for the collection of references (RDD) and one for each RDD entry, in the modified structure (right) the number of objects is equivalent to the number of collections, which in turn is equivalent to the number of fields in an object.

Although this transformation facilitates the job of the GC
since it has fewer objects to traverse, one potential drawback
is the loss of locality caused by splitting objects into their
components.

B. Inlined homogeneous structures
To counter the locality issue we introduce another optimiza-
tion to the layout transformation shown earlier that can be
applied to objects only composed of fields of the same type.
In this case fields can be directly inlined in sequence into
one single structure as shown in the lower part of Figure 4
(In-Sequence Data Layout), thus preserving locality. At the
same time objects can be accessed using indexes pointing at
a specific place in the structure.

1) Text processing
Inlining can be commonly applied to text processing
based on the fact that applications handling text often split
documents into chunks, usually words, at some point in
their processing. The JVM treats strings as regular objects.
By splitting documents into words these applications create
an RDD containing references to (string) objects, one per
word. Therefore, processing a large number of possibly large
documents produces a vast number of objects.

To pursue our goal we move from an RDD containing
references to string objects to an RDD containing indexes
representing word positions in the original text. Figure 5a
illustrates this approach. By using integer-valued indexes in
the RDD we can avoid creating one object per word since
the JVM inlines these indexes directly in the RDD structure
instead of creating objects. As a result, only two objects are
produced: one for the full index array and another one for
the whole text.

C. Iterative processing
Text processing is also the natural target for our third
 technique, which focuses on the object’s long lifetime rather
than the volume of objects. When handling multiple dataset
files Spark loads their full content in memory for processing,
often in the form of multiple small object as mentioned in
the previous section. This data remains alive as long as the
whole processing is ongoing and cannot be discarded by the
GC. To counter this we create an iterator that runs through
the data, loading only the smallest possible portion each time
(typically a word, see Figure 5b), thus allowing the rest of
the data to be discarded by the GC. This procedure seems
to lead to better performance, even in situations where data
is reused (and thus loaded) multiple times.

D. Partitioning
Spark, like other similar cluster computing platforms,
achieves processing parallelism by partitioning the data and
distributing each partition to different nodes and cores to be
processed. Naturally our data transformation approach takes
this fact into account. Each one of the previously mentioned
transformations are applied partition-wise as exemplified in
Figure 6.

E. Limitations to our solution
1) Structure-of-arrays and indexing
As stated before, our techniques explicitly target applica-
tions that handle homogeneous objects, i.e., objects with
a common type that contains only primitive-typed fields.
These types, however, cover the majority of types used in
Spark applications such as MLib according to our analysis.

There are some composite types that can be handled by
our approach. For instance, if RDD elements are objects
containing a mixture of primitive and non-primitive types,
then we can inline all objects into their caller by replacing
the reference by the fields of the object that the reference
points to. This approach is successful if all objects refer-
cenced in all RDD elements have corresponding types, i.e.,
the data set is homogeneous.

When every RDD element is a linked list storing values
of a primitive type, we can convert the data structure first
to an RDD of (variable-length) arrays, and subsequently use
the same approach used in our handling of variable-length
text strings. If the linked lists store objects with primitive
fields, then we can create multiple RDDs, one for each field
type.

Our approach breaks down, however, when polymorphic
sub-types resulting from inheritance are used. This may
result in elements with variable sizes and variable field
types, which make it hard to lay out and to efficiently calculate indices in the arrays. Large data sets, however, have homogeneous types in all practical cases. As such, this limitation is of little practical consequence.

By reorganizing data as structure-of-arrays, memory can only be released once the whole data set has been fully processed. This may seem to be a limitation as data may stay alive longer. However, as mentioned in section III, Spark tries to keep data in memory until all processing on that data is completed. This is also confirmed by the epochal behavior of the data mentioned in [7] and [5]: data is read, used and discarded at once, independently of its layout.

V. IMPLEMENTATION

As mentioned earlier, we encapsulate our code in the MLlib and GraphX libraries, thus completely hiding the code to handle this transformation from the application developers. As a result, the amount of code that one has to modify is minimal. We will make our full source code available upon acceptance.

A. AOS vs. SOA

We implement AOS to SOA transformations on the graph algorithms Breadth-First Search (BFS) and PageRank, which are implemented in Spark’s GraphX library. The part of our code exposed in the API consists of a new function in a new class to load edges from a file into a new type of graph the details of which are hidden from the developer, e.g. GraphLoader.edgeListFile(ctx, edgeFile, ...) becomes InlinedGraphLoader.edgeListFile(ctx, edgeFile, ...) The edgeListFile method loads a graph from an edge list formatted file. In the original GraphLoader, an EdgePartitionBuilder helper class loads the graph edges into RDD partitions in the form of Edge objects (one object per edge) containing three fields, a source and a destination node IDs as 64-bit integer and an attribute associated with the edge with configurable type (see Listing 1). Our new InlinedGraphLoader implementation avoids the creation of an object per edge by inlining source and destination fields as well as edge attributes into three primitive arrays (see Listing 2). For each edge in the original file, an element is added to each of these three arrays. Therefore, at the end of the load phase, each partition contains only 3 array objects instead of one object per edge.

class InlinedEdgePartitionBuilder(size: Int) {
  val edgeSrcVector = new PrimitiveVector[size] // Source
  val edgeDstVector = new PrimitiveVector[size] // Dest.
  val edgeAttrVector = new PrimitiveVector[size] // Attributes
  ... |

Listing 2: Inlined edge representation

case class Edge(...) {
  var srcId: VertexId // Source vertex
  var dstId: VertexId // Destination vertex
  var attr: ED // Edge attribute
  ...} extends Serializable (...) |

Listing 1: Original edge object content

class InlinedKMeansDirectPartition(points: Array[Double], val dim: Int) { ... |

class KMeansDirectPartition(points: Array[Double], val dim: Int) { ... |

B. Inlined structures

We apply the inlined structure techniques to algorithms of two different families, i.e. clustering and text processing.

We use the k-means algorithm, which is part of Spark’s MLlib library, as representative for clustering. K-means groups a set of n-dimensional points in K clusters. Each point is described as a homogeneous, equal-length vector. Our implementation involves a new type of RDD, which is the sole change in the API. One needs to load data using this RDD instead of using vectors, which are the default choice. MLlib’s original k-means implementation takes an RDD of Vector elements as argument: val rdd = data.map(x => Vectors.dense(...)). In the inlined implementation, this is changed to a new type of RDD which is created using a specific helper class: val rdd = new KMeansDirectRDD(...). The implementation of KMeansDirectRDD involves a new kind of RDD partition that stores an inlined version of each vector (see Listing 3). The partition defines one helper vector and an Iterator that fills the vector’s values whenever the next vector of the partition is requested by calling its next() method.

For text processing we apply our inlining technique on tf-idf. Similarly, the implementation involves a new type of RDD which has to be used in lieu of RDDs containing strings. The new RDD keeps track of words in text using an array of indexes and defines an iterator that returns the next word as a string by extracting it from the original text (see Listing 4).

C. Iterative processing

Text processing applications, and tf-idf in particular, are again our testing target for our iterative processing technique. Central to our implementation is a word iterator that loops over words in text files belonging to a corpus of text files in a specific directory.
Spark 1.3 introduced the DataFrame API, which seeks to improve the performance and scalability of Spark (similarly Spark 2.0 introduced the DataSet API, which is a strongly-typed version of it). The DataFrame API introduces an SQL-like schema to describe the data, allowing Spark to better manage data when moving it between nodes and when serializing it. To access data in dataframes, Spark generates optimized query plans, which reduces data communication and allows some transformations on serialized data. Even so, after query plan creation and data scattering, the actual execution, though hidden from the user, is performed on RDDs (which are also the root component of dataframes), inside a single node if no serialization is happening. This is confirmed by [44] and our own tests. We base our analysis on RDDs (which are also the root component of dataframes), and we deliberately avoid serialization, as our goal is to improve the GC performance.

VI. EXPERIMENTAL EVALUATION

A. Setup

For all our experiments we run Spark 2.1.0 on OpenJDK 1.8.0 build 131. A summary of our test applications (described in section IV) together with size and description of the datasets used can be found in table II. Datasets have been chosen to represent a wide variety of input types, size and configurations.

We analyze the proposed layout transformations on a single-node multi-core platform and on a multi-node (cluster) configuration. In single-node Spark runs on one JVM, i.e. with a single heap space and running a single GC. This allows to collect precise GC information while running Spark’s multi-threaded applications on multiple cores at the same time. To ascertain how independent our transformations are from a specific GC choice on a single-node we analyze the behavior with two different GC types: Parallel GC (best performance) [25], and G1 GC (best for soft-realtime applications with large heaps) [23].

We have explored the sensitivity of GC performance to different factors following Spark’s own GC tuning advice in [59] among others. Following JVM values, gave us consistently good results: -XX:NewRatio=2, -XX:SurvivorRatio=8, -XX:MaxTenuringThreshold=5, -XX:SurvivorRatio=8, -Xms=1.5g, -XX:G1HeapRegionSize=16m as well as Spark’s default configuration for spark.memory.fraction and spark.memory.storageFraction (0.6 and 0.5 respectively). We run our single-node tests on a 1.3GHz 64-core Intel® Xeon Phi™ 7210 CPU with 4 threads per core and 96GB DRAM and our multi-node tests on a cluster of 4 nodes, each equipped with a 6-core, 2 thread per core, 2.40GHz Intel® Xeon™ E5-2620 v3 CPU and 30GB DRAM. After extensively testing our applications with a spectrum of Spark configurations to discover the best performing combination we settled on running the single-node applications using 255 threads and 80GB of memory and each cluster node using 12 threads.

Table III: Baseline execution times with Parallel GC, G1 GC and on our cluster of 4 nodes (with all JVMs running Parallel GC). Throughput (TP) before (b) and after (a) applying the layout transformation for all applications for with Parallel and G1 GC. Dashes are baseline values that are repeated from the lines above. Values are averaged over 10 runs.

### Dataset Summary Table (Table II)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th># files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artificial</td>
<td>1.35 GB</td>
<td>5</td>
<td>Artificial word count, 46358 files are 4th file repeated 488 times [45]</td>
</tr>
<tr>
<td>Gutenberg</td>
<td>20 GB</td>
<td>52,361</td>
<td>Collection of 3,036 English books by 142 authors [44]</td>
</tr>
<tr>
<td>Reuters</td>
<td>22 MB</td>
<td>22</td>
<td>Collection of documents that appeared in Reuters newswire in 1987 [47]</td>
</tr>
<tr>
<td>WestburyLab</td>
<td>10 GB</td>
<td>1,745</td>
<td>A reduced redundancy USENET corpus (2005-2011) [48]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th># files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>4.85 GB</td>
<td>68.99</td>
<td>Directed LiveJournal social network friendship [51][49]</td>
</tr>
<tr>
<td>Flickr</td>
<td>2.30 GB</td>
<td>33.14</td>
<td>Flickr users and friendship connections [52]</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>1.87 GB</td>
<td>39.95</td>
<td>Hyperlink network English Wikipedia [53]</td>
</tr>
<tr>
<td>Google</td>
<td>0.88 GB</td>
<td>5.11</td>
<td>Web pages connected by hyperlinks (2002 Google Programming Contest) [49]</td>
</tr>
<tr>
<td>Wikipedia Italian</td>
<td>1.87 GB</td>
<td>91.56</td>
<td>Wikilinks Italian Wikipedia in Italian [54]</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.07 GB</td>
<td>117.10</td>
<td>Orkut social network and ground-truth [55]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>US census</td>
<td>2.46 GB</td>
<td>68</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>43.93 GB</td>
<td>4</td>
</tr>
<tr>
<td>Gyroscope</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artificial</td>
<td>2.129 GB</td>
<td>1,394</td>
</tr>
<tr>
<td>Gutenberg</td>
<td>2,827 GB</td>
<td>3,211</td>
</tr>
<tr>
<td>Reuters</td>
<td>20 GB</td>
<td>25</td>
</tr>
<tr>
<td>WestburyLab</td>
<td>1,439 GB</td>
<td>2,164</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ParGC</th>
<th>G1GC</th>
<th>Cluster</th>
<th>TP Par.</th>
<th>TP G1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patent</td>
<td>68</td>
<td>68</td>
<td>41 64% 88% 76% 78%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LiveJournal</td>
<td>187</td>
<td>218</td>
<td>218 63% 80% 63% 64%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flickr</td>
<td>152</td>
<td>179</td>
<td>118 70% 78% 67% 68%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wikipedia</td>
<td>74</td>
<td>71</td>
<td>245 59% 89% 76% 76%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google</td>
<td>251</td>
<td>313</td>
<td>30 77% 77% 67% 67%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wikipedia Italian</td>
<td>202</td>
<td>214</td>
<td>297 56% 79% 62% 65%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orkut</td>
<td>190</td>
<td>156</td>
<td>145 49% 92% 70% 77%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patent</td>
<td>89</td>
<td>98</td>
<td>103 83% 90% 88% 90%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LiveJournal</td>
<td>189</td>
<td>139</td>
<td>280 81% 94% 66% 86%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flickr</td>
<td>80</td>
<td>87</td>
<td>147 70% 93% 82% 93%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wikipedia</td>
<td>138</td>
<td>104</td>
<td>161 74% 94% 84% 89%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google</td>
<td>64</td>
<td>76</td>
<td>29 95% 97% 89% 92%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wikipedia Italian</td>
<td>176</td>
<td>152</td>
<td>318 55% 92% 81% 81%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orkut</td>
<td>251</td>
<td>230</td>
<td>331 47% 83% 64% 68%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patent</td>
<td>90</td>
<td>84</td>
<td>25 91% 80% 83% 84%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LiveJournal</td>
<td>97</td>
<td>99</td>
<td>52 92% 97% 84% 86%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accelerometer</td>
<td>100</td>
<td>98</td>
<td>59 91% 97% 83% 85%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Household</td>
<td>109</td>
<td>125</td>
<td>24 94% 97% 84% 90%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artificial</td>
<td>2,129</td>
<td>1,394</td>
<td>3,083 85% 92% 82% 83%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gutenberg</td>
<td>2,827</td>
<td>3,211</td>
<td>3,083 85% 92% 82% 83%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reuters</td>
<td>20</td>
<td>25</td>
<td>11 89% 90% 80% 81%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WestburyLab</td>
<td>1,439</td>
<td>2,164</td>
<td>558 41% 79% 43% 68%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
and 30GB of memory.

In producing the code to handle the data layout transformations we try to adhere to the original code as much as possible. Nevertheless in most cases a slight overhead has to be accounted for when processing these transformations. We use the unmodified Spark code as a baseline in our experiments.

B. Results

Absolute values for the baseline execution times and throughput (non-GC time / total time) with the original code for all applications are displayed in Table III. Reported times are averages of 10 runs for each application. Despite the typical, well documented, irregular performance of Java programs [60], the variability of our results never exceeds 1%.

Baseline execution times in Table III do not show a clear winner between Parallel GC, G1 GC and the cluster configuration. On the one hand, the selection of the right GC is often not an obvious one when running Spark and the discrepancy between our baseline results for the two GC types confirms it. For instance, TF-IDF executes up to 1.5 times faster with G1 GC (Artificial input), while BFS executes 25% faster with Parallel GC. The baseline results show that our cluster is not as performant as our single-node machine. The cluster nodes are not as performant as the single node. Adding the communication overhead to this explains the why the cluster does not out-perform the single node setup.

For the majority of applications the transformations show sizeable speedups when compared with the original version. We also notice that speedups for Parallel GC are generally more substantial than for G1 GC. From the baseline execution time we cannot derive any clear consistent performance winner between Parallel GC and G1 GC. G1 GC is often faster but can also be up to 1.50 times slower. The reasons for this apparent inconsistency can be found in the substantially different way in which the two GCs work. Parallel GC either performs a minor or a full collection in one go. In both cases it stops the running application and marks the objects in the young generation or in the whole heap. Then it moves objects or wipes the heap. This process is optimized for speed but it takes place all at once. G1 GC instead divides the heap into regions that can be collected separately. In addition, G1 GC first collects regions that it reckons having most garbage. By doing so, it can better avoid analyzing live objects. Overall, G1 GC garbage collection runs are shorter and recycle less memory. Nonetheless, we will show that G1 GC still benefits from reducing the number of objects originated by our transformations but not always to the same extent as Parallel GC.

C. AoS vs SoA Transformation

We implemented AOS vs. SOA on graph algorithms BFS and PageRank (see Figures 7a to 7f). We observe that the throughput for BFS is lower or comparable to throughput for PageRank. This is an immediate consequence of the nature of the algorithms: BFS has only few edges active in each iteration, while PageRank has all of them active in every iteration. As such, more work is performed per created RDD in PageRank and throughput is higher. Moreover, in the baseline case, the Parallel GC performs better on smaller graphs (by the count of edges and nodes), while G1 GC performs better on larger graphs. A similar trend occurs for BFS, however, here G1 GC performs best only for the largest graph (Orkut). These observations correlate well with the observed speedups: speedups are higher on Parallel GC than on G1 GC as longer GC intervals appear. Small graphs benefit from less or no speedup, e.g., the Google graph. In the best cases, we see a speedup of nearly 2x. Speedups are smaller in the cluster setup than in the single-node setup. This is to be expected as Spark partitions data sets across the nodes in the cluster. As such, each node loads less data and GC pressure is smaller.

D. Inlining of Homogeneous Objects

In general, larger datasets lead to higher speedups when applying inlining (see Figures 7g to 7l). In particular TF-IDF’s Artificial, which uses large text files, shows the most sizeable improvements (between 1.46 and 3.63) independently of GC type, configuration and technique used. On the other hand, Reuters, which is a small-sized text file set, shows a small slowdown. The layout transformation of the data set, which is included in the overall execution time, requires additional time, which is larger in this case than the savings made in the garbage collector.

Clustering algorithm k-means shows results in line with the other algorithms albeit with a different twist. Performance gains appear to be generally more prominent for G1 GC than for Parallel GC, with Household gaining the most from transformation. We as yet have to pinpoint the cause of this trend, as it doesn’t seem to follow neither the number of instances nor the number of attributes of the dataset. Nevertheless, it consistently shows speedups between 1.02 and 1.79. Also noteworthy is the sizeable speedup of Household despite its modest throughput improvement shown in Table III. Evidently, improved data locality also plays a non-negligible role here.

E. Iterative Processing

TF-IDF, our test algorithm for iterative processing shows the overall best performance gains, in particular for Gutenberg with a 8.23 speedup on a single node (Figures 7m to 7o). Artificial and WestburyLab display speedups between 1.46 and 4.14 for all configuration whereas Reuters doesn’t show any improvement. The cluster results are again more modest. Moreover, no gains are made for the Gutenberg input. This is due to (i) partitioning the data set which reduces GC artifacts, possibly to the point where they have no noticeable impact; (ii) Gutenberg consists of many small files, whereas
we know from the Artificial input that the GC issues are the largest for very large files. With only a small number of small files loaded, GC issues become unimportant.

VII. Conclusion

Most popular data analytics frameworks are constructed in high-level, managed programming languages that use garbage collection to manage heap memory. Due to the nature of data analytics, however, state-of-the-art garbage collectors perform poorly on data analytics workloads. We have demonstrated that workloads may spend 60% of their time in garbage collection, with a single invocation of the garbage collector peaking to 177 s. We show that this overhead grows with the number of heap objects created.

This paper presents three programming techniques to minimize garbage collector overhead. Our approach uses data layout transformations and object-inlining to reduce the count of objects in the heap. Moreover, iterative processing is used to shorten the lifespan of objects. Using these transformations, we demonstrate an average speedup of 1.54 and a maximum speedup of 8.23. Our implementation hides all details in Spark’s libraries allowing developers to utilize it with minimal changes.

Acknowledgments

This work has been supported by the European Commission’s Horizon 2020 program grant no. 687628 (VINEYARD) and by EPSRC grant EP/I017271/1 “Heterogeneous Parallel and Distributed Computing with Java (HPDCJ)”.

References
