# An Experimental Evaluation of Garbage Collectors on Big Data Applications

Lijie Xu[1], Tian Guo[3], Wensheng Dou[1,2], Wei Wang[1,2,*], Jun Wei[1,2]
[1]State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[3]Worcester Polytechnic Institute
[1]{xulijie, wsdou, wangwei, wj}@otcaix.iscas.ac.cn, [3]tian@wpi.edu

## ABSTRACT

Popular big data frameworks, ranging from Hadoop MapReduce to Spark, rely on garbage-collected languages, such as Java and Scala. Big data applications are especially sensitive to the effectiveness of garbage collection (i.e., GC), because they usually process a large volume of data objects that lead to heavy GC overhead. Lacking in-depth understanding of GC performance has impeded performance improvement in big data applications. In this paper, we conduct the first comprehensive evaluation on three popular garbage collectors, i.e., Parallel, CMS, and G1, using four representative Spark applications. By thoroughly investigating the correlation between these big data applications' memory usage patterns and the collectors' GC patterns, we obtain many findings about GC inefficiencies. We further propose empirical guidelines for application developers, and insightful optimization strategies for designing big-data-friendly garbage collectors.

**PVLDB Reference Format:**
Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. An Experimental Evaluation of Garbage Collectors on Big Data Applications. *PVLDB*, 12(5): 570-583, 2019.
DOI: https://doi.org/10.14778/3303753.3303762

## 1. INTRODUCTION

Big data frameworks, e.g., Hadoop MapReduce [1] and Apache Spark [2], are developed using garbage-collected languages, such as Java and Scala. These languages speed up the framework and application development because garbage collection (GC) liberates developers from complicated memory management and reduces the chance of memory leaks. However, automatic memory management has great impacts on application performance when the garbage collector has too many objects to manage.

Different from traditional Java/Scala applications, big data applications are both *data-intensive* and *memory-intensive*. These applications not only store a large amount of data in memory but also generate a lot of intermediate computing results in memory. Aside from the large volume, these in-memory data objects have different lifecycles that further complicate garbage collection. As a result, big data applications often have significant memory pressure that leads to heavy GC overhead such as frequent GC cycles and long GC pauses. For example, existing studies reported that GC activities can take up to 50% of big data application execution time [21, 33], and thus inevitably lead to poor application performance.

The ineffectiveness of GC for big data applications can be attributed to three intertwined reasons. First, the large volume of input data and intermediate computing results generated by user-defined data operators lead to substantial memory usage. Moreover, some reusable intermediate data are cached in memory for better performance [3], and thus further increase the memory pressure. Second, the current frameworks' memory management mechanisms only tackle data-level memory management. For example, Spark splits the memory space into two logical parts: the first part for data caching & shuffling, and the second part for storing intermediate computing results. This coarse-grained partition is inefficient in memory management: Spark cannot utilize empty space in the second part for data caching & shuffling. Third, the current object-level memory management approaches in garbage collectors do not take into account object characteristics of big data applications when determining *how* to allocate new objects and *how* to reclaim the unused objects.

Existing works have explored GC inefficiency from different aspects. MemTune [52] dynamically tunes the configuration for data cache to optimize the memory utilization. Facade [45], Deca [42], and Broom [37] propose region-based and lifetime-based memory managers to reduce the GC overhead. Recently, Yak [44] and NG2C [32] optimize garbage collection mechanisms, such as heap layout and GC algorithms. However, none of these works explore GC inefficiency by studying the correlation among the abovementioned three aspects. To thoroughly understand GC inefficiency for big data applications, we investigate three key research questions.

- **RQ1:** What are the typical memory usage patterns of big data applications?
- **RQ2:** Are current garbage collectors sufficient for big data applications? If not, why?
- **RQ3:** What are the guidelines for application developers and insights for designing big-data-friendly garbage collectors?

To answer the above research questions, we first analyze the computation features and memory usage patterns of four widely-used Spark applications for SQL, machine learning, and graph computation. Then, we conduct a comprehensive evaluation on the four representative Spark applications using three popular garbage collectors, i.e., Parallel, CMS (Concurrent Mark Sweep), and G1 (Garbage First). By analyzing the correlation between applications' memory usage patterns and the GC patterns, we identify the

*Corresponding author.

*Proceedings of the VLDB Endowment,* Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303762

root causes of performance differences among the three garbage collectors. Based on the above analysis, we obtain ten findings on the GC inefficiencies for big data applications. We further propose several guidelines for application developers and GC optimization strategies for researchers. Our main findings and optimization approaches are summarized as follows.

**Key Findings.** (1) Big data applications' unique memory usage patterns (e.g., long-lived shuffled data and humongous data objects), and computation features (e.g., iterative computation and CPU-intensive data operators) contribute to the substantial performance differences among garbage collectors. (2) The concurrent collectors, such as CMS and G1, can reduce the GC pause time while reclaiming the long-lived shuffled data. However, they hinder CPU-intensive data operators due to serious CPU contention. (3) All three collectors are inefficient for managing humongous data objects, which lead to frequent GC cycles and even OOM errors in non-contiguous collectors like G1.

**Proposed optimizations.** (1) All three collectors cannot allocate proper heap space to accommodate long-lived shuffled data. To optimize object allocation, we propose a new heap resizing policy through memory usage prediction and dynamic heap space adjustment. (2) All three collectors suffer from unnecessary continuous GC while reclaiming the long-lived shuffled and cached data. By leveraging data lifecycles, we propose a new object marking algorithm to reduce GC frequency. (3) All three collectors are inefficient for iterative applications that need to reclaim large volume of shuffled data in each iteration. By leveraging the distinctive lifecycles and fixed size of these data, we propose a new object sweeping algorithm for achieving no GC pause for iterative applications.

In addition, we identify the root causes of two OOM errors, namely Spark framework's memory leak in handling consecutive shuffle spills and G1's heap fragmentation problem. Spark and OpenJDK communities have confirmed our identified causes [19, 10]. In summary, our main contributions are as follows.

- We summarize the typical memory usage patterns of big data applications, and empirically evaluate three widely-used garbage collectors on four Spark applications. Our in-depth study reveals the inefficiencies of current garbage collectors.
- Based on our findings on GC inefficiencies, we propose four guidelines for application developers and three GC optimization strategies for researchers.
- Our findings and implications can open up new research directions for designing big-data-friendly garbage collectors.

## 2. BACKGROUND

### 2.1 Spark memory management

The key part of a Spark application is the user-defined driver program. It contains a series of data operators manipulating key-value data. In Spark, the input, output, and intermediate data are modeled as *Resilient Distributed Dataset* (RDD) [54], which is a data structure that represents a partitioned collection of data as shown in Figure 1. Spark framework automatically transforms the driver program into a DAG-based dataflow graph according to the data dependencies among RDDs. The dataflow graph is further transformed into a MapReduce-like execution plan, including several stages split by shuffle dependencies. In this paper, map stages refer to the stages that do not need data shuffling and are usually the first stages, while reduce stages refer to the stages that need to shuffle data from previous stages. At runtime, Spark framework allocates multiple executors (i.e., JVMs) that run map/reduce tasks to perform the data operations in each stage.

The memory usage of a Spark application consists of three parts. (1) *Cached data*. Big data applications, especially iterative ones, usually cache reusable data in memory to reduce disk I/O. The cached data are usually *long-lived* objects and span multiple stages. Spark allocates a logical *storage space* to store the cached data as shown in Figure 1c. (2) *Shuffled data*. As shown in Figure 1a, if the application contains $N$-to-$N$ shuffle dependencies (e.g., dependencies between *ParallelRDD* and *CoGroupedRDD*), Spark performs shuffle write/read to transfer the data between map and reduce stages as shown in Figure 1b. In the shuffle write phase, each map task outputs key-value $\langle k, v\rangle$ records into different partitions according to the *key k*. In the shuffle read phase, reduce tasks fetch $\langle k, v\rangle$ records from map outputs and perform in-memory aggregation. *Aggregation* means that the reduce task uses HashMap-like data structures to accumulate the shuffled $\langle k, v\rangle$ records into $\langle k, \mathit{list}(v)\rangle$ or directly combine the $\mathit{list}(v)$ into new $v'$ using data operators such as $\mathit{sum}(\mathit{list}(v))$. The aggregated records are usually *long-lived* data objects since they need to be kept in memory until the task ends. As shown in Figure 1c, Spark allocates an *execution space* to keep the shuffled records. The execution space and storage space share the default 60% of the JVM heap space. (3) *Operator-generated data*. User-defined data operators may generate intermediate computing results during data processing. Since these operators process the data records one by one, the in-memory computing results can be *short-lived* objects (e.g., temporary output records) or *long-lived* records (e.g., buffered intermediate results). Spark allocates a logical *user space* with the default 40% of the heap space for storing these intermediate computing results.

### 2.2 JVM memory management

All the cached, shuffled, operator-generated data are stored as objects in JVM heap and managed by garbage collectors. In this section, we explain the key differences of heap layouts and GC algorithms used by three popular garbage collectors, i.e., Parallel, CMS and G1. We focus on the important aspects of garbage collectors' designs and their implications.

The *weak generational hypothesis* [11] states that most objects survive for a short time and only a few objects live long. Based on this hypothesis, garbage collectors divide the heap into two generations: *young generation* for keeping short-lived objects, and *old generation* for keeping long-lived objects. The young generation consists of an Eden space and two Survivor spaces. New objects are initially allocated in Eden space and further copied to Survivor space when GC occurs. If the objects in Survivor space survive a few GC cycles, they will be promoted to old generation. In Parallel and CMS, the young and old generations are both contiguous space with an explicit boundary. In contrast, G1 logically separates young and old generations in a non-contiguous way, by dividing heap space into a large number of equal-sized regions. Each region can be Eden, Survivor, or Old space. Specially, G1 stores humongous objects (larger than 50% of the region size) into humongous regions, which span multiple regions in old generation.

**GC algorithms.** A typical GC cycle involves two tasks: (1) *object marking* that identifies the live (reachable) objects through traversing the object reference graph; (2) *object sweeping* that reclaims the memory space occupied by the unused objects. To obtain consistent results, garbage collectors may suspend application threads while performing object marking and sweeping. Such scenarios are referred to Stop-The-World (STW) GC pauses.

For garbage collection in young generation, namely minor or young GC, all three collectors use the STW *mark-copy* algorithm. This algorithm first marks the live objects in Eden space, and then copies the live objects to one of the Survivor spaces. The other

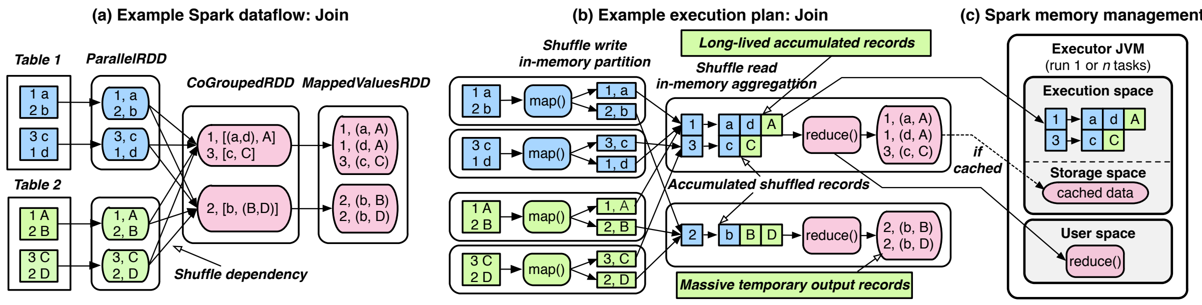

**Figure 1:** Illustrations of an example Spark application from dataflow graph, execution plan, to memory management.

**Table 1:** Representative Spark applications with different computation features and memory usage patterns. The **bold** ones are long-lived data objects.

| Application | Type | Application features | | | Memory usage patterns |
|---|---|---|---|---|---|
| | | #Cached data | #Shuffled records | Space complexity | |
| GroupBy | SQL | None | Medium: $O(N_{rows})$ | *reduceByKey*(*sum*) : $O(1)$ | ***Accumulated records*** |
| Join | SQL | None | Heavy: $O(N_{rows\ of\ R\&U})$ | *join*() : $O(m+n)$ | ***Accumulated records****, Temporary output records* |
| SVM | ML | $O(N_{matrix\_rows})$ | Light: $O(N_{map\_task})$ | *reduce*() : $O(\|x\|)$ | *Humongous data objects,* ***Cached records*** |
| PageRank | Graph | $O(N_{edges})$ | Medium: $O(N_{edges})$ | *join*() : $O(m+n)$ | *Iterative* ***accumulated records****,* ***Cached records*** |

Survivor space is used for swapping the live objects. The live objects that survive several young GCs in Survivor space are finally promoted to old generation.

For garbage collection in old generation, namely major or full GC, the three collectors use different GC algorithms. **Parallel** collector is designed for high throughput by launching multi-threads to perform full GC. Its object marking and sweeping phases are stop-the-world, and performed together in a single full GC. So, Parallel collector suffers from long full GC pause when there are too many objects to mark and sweep. Parallel collector also compacts the free space in full GC cycle to eliminate heap fragmentation. **CMS** is designed for low latency by offloading much object marking and sweeping work to background threads that do not require stopping application threads. Its object marking phase is divided into three phases: the *concurrent mark phase* is for marking all the live objects, while the other two STW phases (*initial mark* and *remark* phases) are for identifying the GC roots and updating the marking results. Its object sweeping phase also runs concurrently with application threads. CMS does not have compacting step that may lead to heap fragmentation. **G1** is designed to balance the throughput and latency, through profiling the percentage of live objects in each region and incrementally reclaiming the regions that are filled with unused objects. Its object marking phase is similar to that of CMS. However, to balance the GC frequency and memory utilization, it performs object sweeping incrementally in two phases. One is at the end of each object marking phase, which only sweeps the regions without any live objects and selects the old regions with high occupancy of unused objects as candidate regions. When the total unused objects in these candidate regions reach a threshold, G1 launches a STW *mixed collection* phase to reclaim them. This mechanism reduces the GC frequency at the cost of high memory consumption. In this paper, we focus on full GC performance since full GC is usually more time-consuming than young GC.

## 3. METHODOLOGY

We first explain how we select representative applications to trigger diverse memory usage patterns in Spark. We then introduce how we setup the experiments with different input data sizes and configurations. We finally present how we perform GC analysis by scrutinizing the correlation between the applications' memory usage patterns and the collectors' GC patterns.

### 3.1 Application selection

The memory usage of Spark applications is affected by data features (e.g., cached data, shuffled data, and operator-generated data) and computation features (e.g., iterative computation). We select four Spark applications in Table 1 as our experimental subjects. The four applications are representative. (1) They are from different domains, e.g., SQL query, machine learning, and graph computation; (2) They have different computation patterns, e.g., heavy/-light data shuffle, data aggregation with different space complexities, iterative computation with data cache; (3) They have diverse memory usage patterns, e.g., *long-lived accumulated results*, *temporary output records*, and *humongous data objects*. We explain these four applications as follows.

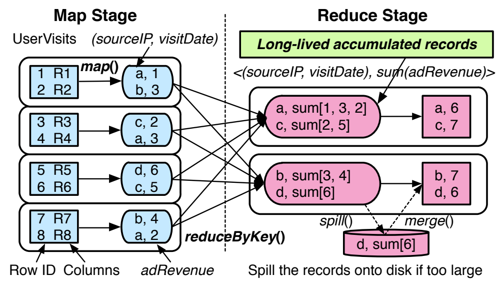

**Figure 2:** The dataflow of GroupBy application.

**(1) GroupBy** is a SQL application simplified from the aggregation query in Spark's BigSQL benchmark [20, 47]. Figure 2 illustrates the GroupBy dataflow, where *sourceIP*, *visitDate*, and *adRevenue* are three columns of table *UserVisits*.

```
SELECT sourceIP, visitDate, SUM(adRevenue)
FROM UserVisits GROUP BY sourceIP, visitDate;
```

This application is implemented with basic RDD APIs. In map stage, the map tasks transform each row of table *UserVisits* to be $\langle(sourceIP, visitDate), adRevenue\rangle$ record. In reduce stage, each reduce task performs *reduceByKey*() to group shuffled records with the same key to $\langle(sourceIP, visitDate), list(adRevenue)\rangle$ records, and simultaneously performs the aggregation function *sum*() on each *list*(*adRevenue*). Although the space complexity of *sum*() in *reduceByKey*() is $O(1)$, these aggregated records are memory-consuming. These records are accumulated in a HashMap-like data

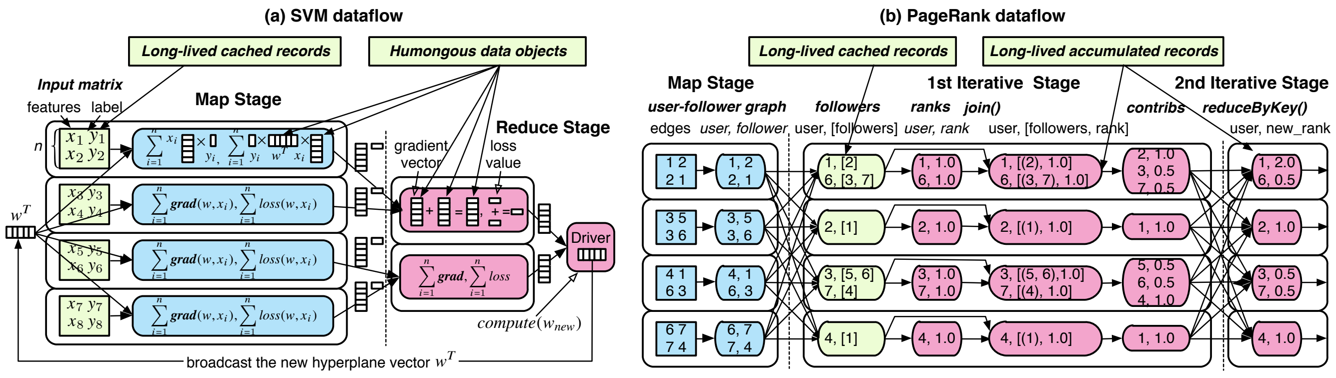

**Figure 3:** The dataflow of SVM and PageRank applications.

structure with different ⟨*sourceIP*, *visitDate*⟩ keys and remain in memory until either being spilled onto disk or the completion of reduce task. Therefore, *reduceByKey*() exhibits memory usage pattern of *long-lived accumulated records*.

**(2) Join** is a SQL application simplified from the join query in the benchmark [20, 47]. Figure 1b shows the dataflow of Join.

```
SELECT URL, pageRank, adRevenue
FROM Rankings As R, UserVisits As U
WHERE R.URL = U.URL;
```

This application is implemented with basic RDD APIs. The map tasks transform each row of table *Rankings* to ⟨*URL*, *pageRank*⟩ record and transform each row of *UserVisits* to ⟨*URL*, *adRevenue*⟩ record. In shuffle phase, each reduce task performs *join*() operator to group the two tables' rows with the same key *URL* as ⟨*URL*, *list*(*pageRanks*, *adRevenues*)⟩ records. These grouped shuffle records are kept in memory until being spilled onto disk or the completion of reduce task, so they are referred to *long-lived accumulated records*. In output phase, the *join*() operator calculates the Cartesian product of the two sets *pageRanks* and *adRevenues*, and output new records ⟨*URL*, *pageRank*, *adRevenue*⟩ one by one. Since these records are directly output into HDFS, they are regarded as *massive temporary output records*. The space complexity of *join*() is $O(m + n)$, where $m$ and $n$ denote the length of set *pageRanks* and set *adRevenues*. Join application suffers from *heavy* shuffle, because the number of its shuffled records equals the number of rows from both *Rankings* and *UserVisits* tables.

**(3) Support Vector Machine (SVM)** is an iterative machine learning application from Spark MLlib [15] for large-scale data classification. The training data is a large matrix that contains a large number of data points. Each data point contains a feature vector $x$ and a class label $y$. SVM uses gradient descent algorithm to iteratively compute the best hyperplane vector $w$ to separate data points into two classes by minimizing a *loss* function. Figure 3a shows the dataflow of SVM, which uses a linear kernel with L2 regularization. The bold variables in the code denote vectors.

```
gradient = matrix.map(x =>(grad(w,x), loss(w,x)))
                 .reduce(sum(grad), sum(loss))
w = w - stepSize * gradient
```

At the beginning of each iteration, the initial hyperplane $w$ is broadcasted to each map task. Map tasks then perform *map*() to compute the vector *grad*$(w, x)$ and value *loss*$(w, x)$ of each data point $x$ and sums the ⟨*grad*, *loss*⟩ together. The space complexity of *map*() is $O(|x|)$, where $|x|$ represents the dimension of data point $x$. Since $|x|$ is usually huge (∼60 millions in our experiments), the *grad* vector and hyperplane vector $w$ are *humongous data objects* (large double array). Different from GroupBy and Join, SVM has *light* shuffle because each map task only outputs one record and only $N_{map\_task}$ records are shuffled to subsequent reduce tasks. Each reduce task does not accumulate the shuffled records but perform *reduce*() to aggregate them into one ⟨$\sum$ *grad*, $\sum$ *loss*⟩ record. The space complexity of *reduce*() is also $O(|x|)$. Finally, the driver program collects *grad* vectors from all the reduce tasks, sums these vectors, and updates the hyperplane $w$. The training data are regarded as *long-lived cached records*, because they are cached in memory and serve as the input data for each iteration.

**(4) PageRank** is an iterative graph application for measuring the importance of each vertex according to the linked edges. Here, PageRank is used to compute the *rank* of each user in Twitter's user-followers graph [23].

```
contribs = followers.join(ranks).flatMap{
           (user, (followers, rank)) =>
           followers.map(f => (f,rank/|followers|))
}
ranks = contribs.reduceByKey(sum(contrib))
                .map(rank => 0.15 + 0.85*rank)
```

As shown in Figure 3b, map tasks perform *map*() to transform each edge to be ⟨*user*, *follower*⟩ record. In the first iterative stage, each reduce task groups the shuffled ⟨*user*, *follower*⟩ records into ⟨*user*, *list*(*followers*)⟩, which are further cached in memory as the input data for the following iterations. Therefore, these records are *long-lived cached records*. Next, reduce tasks *join* these records with users' *ranks* as ⟨*user*, *list*(*followers*, *rank*)⟩, and compute the Cartesian product on each *list*(*followers*, *rank*). This *join* operation does not require additional data shuffling because the *followers* and *ranks* RDDs are co-partitioned. Join is performed in each iteration and generates *massive temporary records*. In the second iterative stage, reduce tasks perform *reduceByKey*() to aggregate the shuffled ⟨*user*, *rank*⟩ records into ⟨*user*, *sum*(*list*(*rank*))⟩. These shuffled records occupy $O(N_{edges})$ space and remain in memory until the iteration ends, so they are *long-lived accumulated records*. Different from GroupBy and Join, these *long-lived accumulated records* are generated and reclaimed in each iteration. Finally, the reduce tasks perform *map*() to compute the new *rank* of each user. The rest of iterative stages are the same as the second iterative stage.

## 3.2 Experimental setup

### 3.2.1 Input data variation

Input data size is the key to decide the volume of shuffled data and affects the memory usage of user-defined data operators. We

**Table 2:** Input data selection and variation. Table *Uservisits* and table *Rankings* are generated by HiBench [9].

| Application | Data-1.0 (100%) dataset | Data-0.5 (50%) dataset |
|---|---|---|
| GroupBy | 200GB Uservisits (1.2B rows) | 50% rows (100GB) |
| Join | 200GB Uservisits (1.2B rows), 40GB Rankings (600M rows) | 50% rows (100GB, 20GB) |
| SVM | 21GB KDD2012 matrix [12] (149M rows, 54M features) | 50% columns (11.2GB, 27M features) |
| PageRank | 25GB Twitter graph [23] (476M edges, 17M nodes) | 50% edges (12.2 GB, 238M edges) |

select different data sizes that can lead to different memory pressures. As shown in Table 2, we select four widely-used datasets with two different sizes: **data-1.0** (100% of the input data) and **data-0.5** that consists of 50% of the input data. Note that data-0.5 is sampled from data-1.0 based on individual dataset's characteristics. For dataset in GroupBy and Join, we select the first 50% rows of each table. For SVM, we randomly select 50% of the matrix columns because the memory usage of data operators is related to the matrix dimension. For dataset in PageRank, we select the first 50% of the edges because the memory usage of data operator is related to the number of edges.

### *3.2.2 Dataflow and GC configurations*

Spark applications contain dataflow-related configurations such as *input block size* and *partition number*. These configurations determine the task parallelism. The number of map tasks is equal to *input data size* divided by *input block size* (default at 128MB). The number of reduce tasks equals *partition number*. To fully use the cluster resource, we set the *partition number* to be the number of CPU cores in the cluster. For other configurations, we adopt the default values to reduce test space. We enable Parallel collector using *-XX:+UseParallelGC*, CMS using *-XX:+UseConcMarkSweepGC*, and G1 using *-XX:+UseG1GC*.

### *3.2.3 Experimental environments*

We perform the evaluation on a cluster of nine *mn4.2xlarge* nodes on Alibaba Cloud. One node serves as the master, and the others serve as workers. Each node has 4 physical cores (8 virtual cores) and 32 GB RAM that concurrently runs 4 executor JVMs. To avoid memory contention, each JVM is configured to run only one task. Therefore, each JVM has one physical core with 6.5 GB heap, and the remaining 6 GB memory of each node is used for off-heap Java NIO buffers, operating system, and Hadoop DataNode process. For the master node, the driver program is configured to use 16 GB memory to accommodate the large (12.8 GB) parameter vectors in SVM. We use Spark 2.1.2 standalone version with Hadoop HDFS 2.7.1, running on Ubuntu 16.04 and Oracle HotSpot JVM 64-Bit 1.8.0, to perform all the experiments. We use Spark standalone version instead of YARN-based version to eliminate the memory effects of YARN containers [18]. Before each run, we clear OS buffer caches and restart the workers to eliminate the cache effects.

## 3.3 Analytical approaches

For each application, we run it with two input data sizes and three garbage collectors. We repeat each run five times and report the average application execution time. We also select the application with the medium execution time from the five repeated experiments for comparing the stage and task execution time.

### *3.3.1 Application profiling*

We design and implement three profilers to collect various performance metrics. (1) The *execution time profiler* measures the execution time of each application and each map/reduce task. (2) The *dataflow profiler* collects the number and size of the records in each data processing phase. We extend the Spark log system to record the spilled data size and spill time. (3) The *resource profiler* collects the CPU, memory usage, and GC metrics of each task. All three profilers are open at our github project *SparkProfiler* [22].

### *3.3.2 Performance comparison and analysis*

For each application, we quantify the performance difference among three garbage collectors through comparing the following metrics. (1) *Application execution time* identifies whether the three collectors have different impacts on the application performance. (2) *Task execution time* identifies where the performance difference occurs since map tasks and reduce tasks have different memory usage patterns. (3) *Fine-grained task execution time* identifies the potential causes of the performance difference. Task execution time is decomposed to three parts: *data computation time*, *shuffle spill time*, and *GC time*. If GC time dominates the performance difference, we further perform *GC pattern comparison*.

**GC pattern comparison.** This comparison aims to identify the root causes of the GC time difference. The GC patterns we study include (1) *Memory allocation pattern*. The three collectors may allocate different young/old generation sizes to accommodate the data objects with different lifecycles. We analyze *how* different allocation policies affect the GC time. (2) *GC time and GC frequency pattern*. The three collectors adopt different GC algorithms that may lead to different GC time. For concurrent collectors such as CMS and G1, the GC time consists of *young GC time*, *full GC time*, and *concurrent GC time*, as shown in Figure 4. *Full GC time* only refers to the duration of stop-the-world full GC phases (e.g., *initial mark* and *remark* phases), while *concurrent GC time* denotes the duration of concurrent GC phases like *concurrent marking*. Parallel collector's full GC is completely stop-the-world, so its concurrent GC time is 0. We compare these types of GC time across three collectors and identify the most time-consuming GC phases.

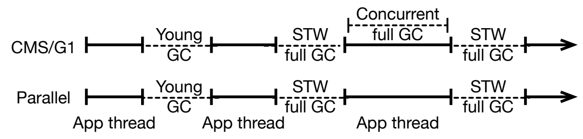

**Figure 4:** The timeline of application thread with GC cycles.

# 4. EXPERIMENTAL RESULTS

## 4.1 Overall results

We compare the applications' average execution time under different data sizes and garbage collectors in Table 3. The key experimental results and key findings are summarized as follows.

**(1) Applications that generate *long-lived accumulated records* are more prone to garbage collectors' inherent inefficiency, with over 10.8% increased execution time.** *Long-lived accumulated records* refer to the large number of shuffled records, which are accumulated in memory by data aggregation operators such as *reduceByKey*() and *join*(). As shown in the grey cells in Table 3, the applications with *long-lived accumulated records* demonstrate over 10.8% and up to 49.1% relative execution time difference. The smallest execution time difference (3.2-4.6%) happens in SVM application, which has *light* shuffle and does not generate *long-lived accumulated records*. Each SVM reduce task only shuffles 12∼14 records ($\sqrt{N_{map\_task}}$) that are further combined into only one record in memory.

**Table 3:** The average application execution time comparison with different data sizes. ×(OOM) means that the applications failed with OOM errors.

| Application | Data-0.5 | | | | Data-1.0 | | | |
|---|---|---|---|---|---|---|---|---|
| | **Parallel** | **CMS** | **G1** | **Comparison** | **Parallel** | **CMS** | **G1** | **Comparison** |
| **GroupBy** | $20.4_{(1.1)}$ | $18.2_{(0.2)}$ | $18.4_{(0.4)}$ | $C < G1 < P(10.8\%)$ | $45.4_{(19)}$ | $36.3_{(0.9)}$ | $39.4_{(1.2)}$ | $C < G1 < P(20.1\%)$ |
| **Join** | $31.8_{(5.7)}$ | $28.3_{(0.3)}$ | $28.4_{(0.8)}$ | $C < G1 < P(11.3\%)$ | $78.7_{(41)}$ | $54.7_{(0.7)}$ | $57.1_{(2.6)}$ | $C < G1 \ll P(30.5\%)$ |
| **SVM** | $6.2_{(0.4)}$ | $6.0_{(0.3)}$ | $6.0_{(0.1)}$ | $C = G1 < P(3.2\%)$ | $15.2_{(1.2)}$ | $14.5_{(1.1)}$ | × (OOM) | $C < P(4.6\%)$ |
| **PageRank** | $26.1_{(11.3)}$ | $19.5_{(3.5)}$ | $38.3_{(3.3)}$ | $C \ll P \ll G1(49.1\%)$ | × (OOM) | × (OOM) | × (OOM) | × |

* This table compares the application's average execution time ($t$) with different data sizes and garbage collectors. $P, C, G1$ denotes *Parallel*, *CMS*, and $G1$. Take the first cell $20.4_{(1.1)}$ for example, the average time it takes to run GroupBy application with half of the data size using Parallel GC is 20.4 min, and the corresponding GC time is 1.1 min. $C < G1$ means that the relative execution time difference between the application with CMS GC and the application with G1 GC is $(t_{G1} - t_C)/t_{G1} < 20\%$. $\ll$ means that the relative execution time difference is 20%+. (10.8%) means that the relative execution time difference between the fastest and the slowest applications with different GCs is 10.8%.

**(2) Applications with concurrent collectors achieve 8.3%-94% shorter GC time than applications with Parallel collector.** CMS and G1 are concurrent collectors that perform most of the full GC concurrently with application threads. Aside from the applications with OOM errors, concurrent collectors achieve 8.3-94% shorter GC time than Parallel collector in all the applications.

**(3) G1 is the only collector that suffers from OOM errors while processing the humongous objects in SVM-1.0 application**. Although G1 is designed for managing memory larger than 6 GB [7], its region-based heap management is not reliable for managing the humongous objects.

### 4.1.1 *Key contributors to the performance differences*

The root causes of the performance differences include big data applications' memory usage patterns, computation features, and the garbage collectors' different GC algorithms. **Among these memory usage patterns, the patterns of *long-lived accumulated records* and *humongous data objects* are the key contributors to the substantial performance differences.**

***Long-lived accumulated records*** refer to the shuffled records that are kept in memory for both shuffle and output phases until being spilled to disk. Since big data applications usually generate millions of *long-lived accumulated records*, these records lead to frequent and long full GC pauses due to the mismatched requirement with current GC algorithms. (1) The *long-lived accumulated records* require large old generation to accommodate. Therefore, inappropriate young/old generation sizing polices will lead to frequent full GC pauses (Finding 2, 3). (2) It is time-consuming and CPU-intensive to reclaim the *long-lived accumulated records*. Current object marking/sweeping algorithms need to traverse the whole object graph to identify the live referenced objects. Since *long-lived accumulated records* are numerous and all referenced, the object marking/sweeping phase is both time-consuming and CPU-intensive. Stop-the-world marking/sweeping algorithm performs sequential object marking and sweeping that leads to long individual full GC pause. Concurrent marking/sweeping algorithms can reduce full GC time by performing object marking and sweeping in parallel with application threads (Finding 4, 10). However, they suffer from *concurrent mode failures* problem (Finding 4) and degrade the CPU-intensive data operators like *join*() (Finding 8). (3) Without awareness of the data object lifecycles, current GC algorithms leverage static GC triggering thresholds that lead to repetitive work during each GC cycle (Finding 7).

***Humongous data objects*** refer to objects that are larger than G1's maximum region size (32 MB). It is common for big data applications to generate humongous objects, such as large vector (big double array) in SVM. The non-contiguous region-based collectors like G1 may not have large enough ***contiguous*** space to accommodate these humongous objects, and therefore suffers from heap fragmentation that eventually leads to OOM errors (Finding 9).

## 4.2 GroupBy results

We explore the impact of *long-lived accumulated records* on the application performance using GroupBy-1.0 as an example.

### 4.2.1 *Performance comparison results*

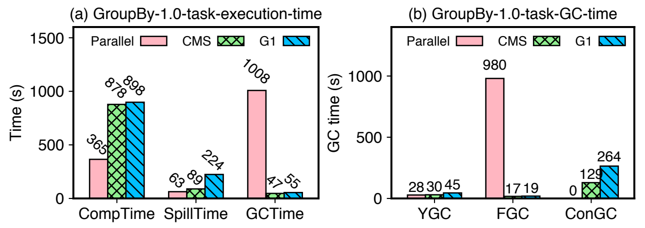

**Figure 5:** The execution time comparison among GroupBy-1.0 reduce tasks. ParallelGC task is the slowest one due to its longest full GC time.

GroupBy-1.0 application contains a map stage (1680 map tasks) and a reduce stage (32 reduce tasks). The performance difference was only observed in reduce stage, where the memory space is dominated by *long-lived accumulated records*. Since the slowest reduce task dominates the execution time of reduce stage, we compare the execution time of the slowest reduce tasks with different garbage collectors and obtain $\mathit{CMS}_{17m} < \mathit{G1}_{20m} < \mathit{Parallel}_{24m}$. As shown in Figure 5a, we further break down the task execution time and group them logically into data computation time (*CompTime*), shuffle spill time (*SpillTime*), and GC time to pinpoint potential performance bottlenecks. Data computation time refers to the time that the task spends on data processing. Figure 5a shows that ParallelGC task achieves 29-72% shorter shuffle spill time than CMS and G1 tasks. This is caused by the three collectors' different heap layouts (Finding 1). To understand GC time differences, we further decompose the GC time into young GC (*YGC*) time, full GC (*FGC*) time, and concurrent GC (*ConGC*) time in Figure 5b. This figure shows that ParallelGC task suffers from ~50x longer full GC time than CMS and G1 tasks. The root causes are due to the three collectors different young/old generation sizing polices and different object marking algorithms (Finding 2, 3, 4).

### 4.2.2 *Findings and their implications*

**Finding 1: ParallelGC tasks trigger 1.5x more shuffle spills than CMS and G1 tasks. The root cause is that Parallel collector has the smallest available heap size that leads to the lowest spill threshold of ParallelGC tasks.** By default, Spark allocates 60% of the JVM heap to store the shuffled data and cached data. In shuffle phase, the reduce task launches a *reduceByKey*() operator to aggregate all the shuffled $\langle k, v\rangle$ records with the same $k$ into $\langle k, \mathit{list}(v)\rangle$ records. These records are accumulated in memory and will trigger shuffle spill if their size exceeds the spill threshold

($\sim$60% of the heap size). Figure 6 illustrates the distribution of the accumulated shuffled records in the 32 reduce tasks. It shows that ParallelGC tasks totally spill 60 times and the spill threshold is 3.29 GB, while CMS/G1 tasks totally spill 39 times and the spill threshold is 3.69/3.70 GB. We find the root cause is that Parallel collector has smaller available heap size than CMS and G1 collectors under the same heap size configuration. Take the 6.5 GB JVM for example, the runtime available heap size comparison is $\mathit{Parallel}_{5.78G} < \mathit{CMS}_{6.44G} < \mathit{G1}_{6.50G}$. In Parallel collector, the missing 0.72 GB (6.50-5.78) heap space is used as a Survivor space for swapping the survival objects during young GC [24]. This Survivor space is not used for storing new objects, so it is not included in the available heap space. CMS collector has the same problem but only 0.06 GB (6.50-6.44) missing space due to its smaller Survivor space. In contrast, G1 adopts region-based heap layout and is able to use the Survivor space as available heap space. Since Spark does not change the spill threshold at runtime, ParallelGC tasks achieve the shortest spill time due to the smallest size of spilled data (3.29GB). However, ParallelGC tasks suffer from more shuffle spills and disk I/O.

**Implication:** We need to design dynamic spill threshold to balance the spill time and spill frequency according to the available runtime heap size.

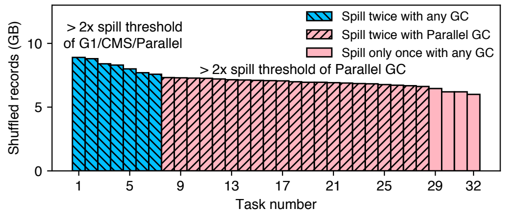

**Figure 6:** The distribution of accumulated shuffled records in 32 GroupBy reduce tasks. Due to data skew, some tasks spilled twice and others spilled only once. The spill threshold is $\mathit{Parallel}_{5.78G} < \mathit{CMS}_{6.44G} < \mathit{G1}_{6.50G}$.

**Finding 2: Different young/old generation sizing polices lead to different full GC frequencies, because the *long-lived accumulated records* require large old space to accommodate. By allocating large old space without shrinkage, CMS tasks achieve $\sim$48% less full GC pauses than ParallelGC and G1 tasks.** As described in Finding 1, the *reduceByKey*() operator constantly accumulates shuffled records into memory (about 5.5GB in the slowest reduce task). These *long-lived accumulated records* span from shuffle phase to output phase. Even when they are spilled onto disk, they will be gradually read back into memory to merge with the unspilled records in output phase. In JVM, these *long-lived* records are constantly transferred from young generation to old generation, and will trigger full GC when the old generation becomes full. This indicates that the size of the young/old generation has impacts on the GC performance. Fortunately, the three collectors have *adaptive generation sizing polices*, which can dynamically adjust the young/old heap size according to the statistics of GC pause time and heap occupancy (known as *GC Ergonomics* [8]). However, we find that the three collectors demonstrate inefficient generation sizing patterns that lead to high young or full GC frequencies. (1) *Parallel GC prefers to expand and shrink the old space according to the heap occupancy.* As shown in "Allocated" line in old generation in Figure 7a, Parallel GC constantly enlarges the old space to accommodate the increasing shuffled records. However, its allocated old space grows up to the smallest size (4.33GB) compared to that of CMS/G1 GC ($\sim$6GB) in Figure 7b/c. The reason is that Parallel GC limits its old space to be 66.6% of the heap size. When the memory usage drops down after shuffle spill, Parallel GC also shrinks the old space to a small size ($\sim$2.6GB). Small old space leads to frequent full GC pauses. (2) *CMS prefers to expand the old space without shrinkage.* Since CMS allocates large old space (about 90% of the heap size) and does not shrink at the spill time, it has enough old space to keep the *long-lived accumulated records* in both shuffle and output phases. As a result, CMS task only triggers 27 full GC cycles that are 48-68% less than that of Parallel/G1 task. However, CMS task suffers from 2x more young GC pauses due to its smallest young space. (3) *G1 prefers to balance the size of young/old space according to the statistics of GC pause time and heap usage.* As shown in Figure 7c, G1 allocates large old space to accommodate the increasing shuffled records in shuffle phase. However, after shuffle spill, it tries to enlarge the young space and shrink the old space to accommodate the read-back spilled records. This policy leads to 10 more full GC cycles than CMS tasks since the read-back shuffled records still require large old space.

**Implication:** Current young/old generation sizing polices are inefficient for accommodating the *long-lived accumulated records*. We need to design more intelligent heap sizing polices with awareness of the memory usage in each data processing phase.

**Finding 3: Compared to CMS and G1 collectors, Parallel collector's inappropriate generation resizing timing mechanism leads to 38% more full GC pauses.** Finding 2 shows that all the three collectors can resize the old space to accommodate the *long-lived accumulated records* in shuffle phase. However, their different generation resizing timing mechanisms (i.e., *when* to resize the old generation) also result in different GC frequencies. Parallel collector can only resize the old generation at full GC pauses. As a result, all its full GC pauses in shuffle phase are caused by this resizing timing requirement. In contrast, CMS and G1 collectors can resize the old generation during lightweight young GC pauses, which reduces the full GC pauses.

**Implication:** We not only need to solve *how* to resize the young/old generation but also *when* to take the resizing action.

**Finding 4: For reclaiming the *long-lived accumulated records*, Parallel collector's stop-the-world object marking algorithm is 10x slower than CMS and G1 collectors' concurrent marking algorithms.** As shown in Figure 7, ParallelGC tasks suffer from 10x longer individual full GC pause than CMS and G1 tasks. The reason is that Parallel GC uses stop-the-world object marking algorithm named *mark-sweep-compact*, which needs to suspend application threads to mark the live objects and sweep the unused objects. Since the *long-lived accumulated records* are numerous ($\sim$38 millions), this stop-the-world marking is time-consuming that leads to up to 10-20s individual full GC pause. In contrast, CMS and G1 collectors use concurrent marking algorithms, which perform most of the object marking work concurrently with application threads. Therefore, their average full GC pauses drop down to $\sim$1s. However, these concurrent algorithms may suffer from long full GC pause when the object reclamation cannot catch up with the need of object allocation. Figure 7b shows that CMS task suffers from a long (11 s) full GC pause caused by *concurrent mode failure*. The failure means that the concurrent marking phase cannot finish before the old space becomes full (when the spilled records are read back into memory). In this occasion, CMS falls back to launch a stop-the-world full GC pause, similar to Parallel collector.

**Implication:** Concurrent object marking algorithm can reduce GC pause time while reclaiming the *long-lived accumulated records*. However, they may suffer from unexpected *concurrent mode failure* when the object reclamation is slower than object allocation.

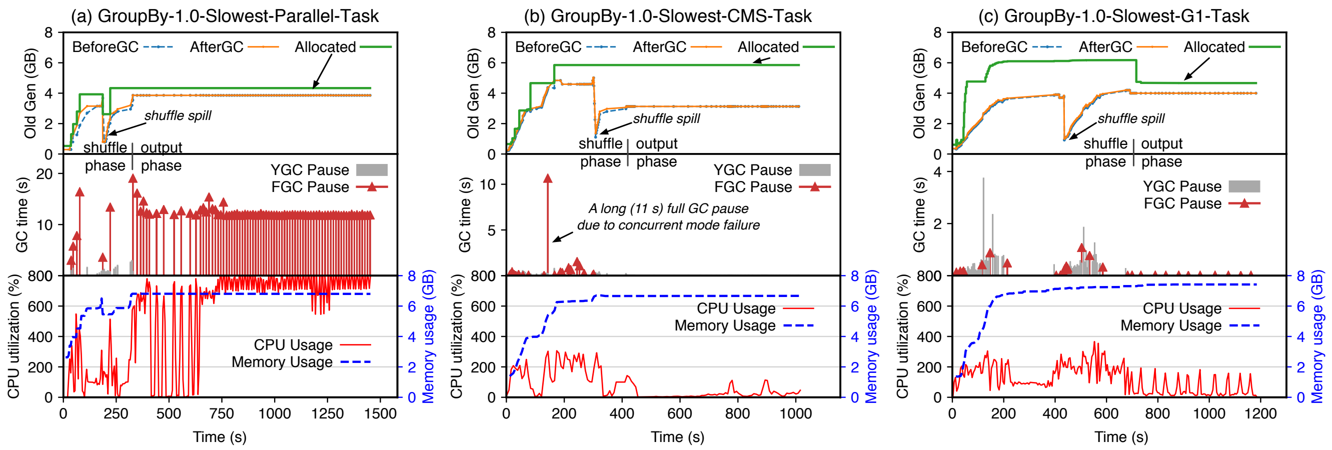

**Figure 7:** The performance comparison among the GroupBy-1.0 slowest tasks with different GCs. *BeforeGC* denotes the size of live objects in old generation before each young or full GC. *AfterGC* denotes the size of live objects in old generation after each young/full GC. *Allocated* denotes the allocated old space.

**Finding 5: ParallelGC tasks suffer from 2.5-7.6x higher CPU usage than CMS and G1 tasks, due to 1.7-12x more full GC pauses and 10x longer individual full GC pause.** The implication is that we need to reduce full GC frequency and individual full GC pause to lower the tasks' CPU consumption.

**Finding 6: G1 tasks suffer from 1.1-1.2x higher physical memory usage than ParallelGC and CMS tasks.** The reason is that G1 collector allocates a large native data structure *remembered sets* for keeping object information used for GC [6]. The implication is that we need to allocate more physical memory for G1 or design more memory-efficient object storing structure for G1.

## 4.3 Join results

In this section, we explore the combined impact of *long-lived accumulated records* and *massive temporary records* on the application performance while using Join-1.0 as an example.

### 4.3.1 Performance comparison results

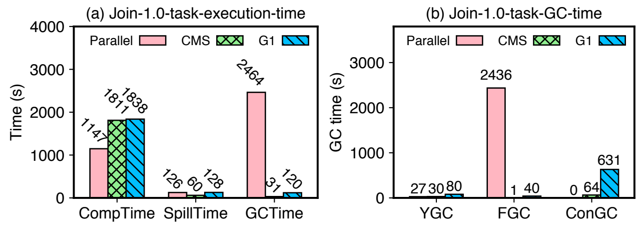

**Figure 8:** The execution time comparison among Join-1.0 reduce tasks is $CMS_{32m} < G1_{35m} < Parallel_{62m}$. ParallelGC task is 1.8x slower than CMS and G1 tasks, due to its extremely long full GC time. The data computation of CMS and G1 tasks are 1.6x slower than ParallelGC task.

Join-1.0 application has two map stages (1680/320 map tasks) and a reduce stage (32 reduce tasks). The performance differences only happen in reduce stage, where the memory usage consists of *long-lived accumulated records* and *massive temporary output records*. As described in Section 3, the *massive temporary output records* refer to the records generated by the Cartesian product operation in *join*(). We compare the execution time of the slowest reduce tasks with different garbage collectors, and obtain $CMS_{32m} < G1_{35m} < Parallel_{63m}$. We further break down the tasks' execution time and GC time in Figure 8. (1) Figure 8b shows that ParallelGC tasks suffer from 60x more full GC time than CMS and G1 tasks. This is mainly caused by the different full GC triggering conditions (Finding 7). (2) Figure 8a shows that CMS and G1 tasks suffer from 1.6x longer data computation time than ParallelGC task. This is caused by the CMS and G1 collectors' CPU-intensive object marking algorithms (Finding 8).

### 4.3.2 Findings and their implications

**Finding 7: Threshold-based full GC triggering conditions lead to frequent, but unnecessary full GC pauses towards the *long-lived accumulated records*. Due to different full GC triggering thresholds, ParallelGC suffers from 1.7x more full GC pauses than G1, and G1 suffers from 7x more full GC pauses than CMS.** Figure 9 shows that the three collectors demonstrate different GC patterns in output phase, where the *long-lived accumulated records* are kept in memory and *massive temporary output records* are constantly generated. ParallelGC task triggers 136 full GC pauses that leads to ~40 min GC time. In contrast, G1 task triggers 81 full GC pauses, while CMS task does not trigger any full GC pauses in output phase. The ***first*** root cause is that the three collectors have different *generation sizing policies*. Parallel collector's generation sizing policy limits the old generation to be a small size (default 2/3 of the heap space), while CMS and G1 collectors allocate 1.2x more old space. The ***second*** root cause is that the three collectors have different full GC triggering conditions. Parallel GC uses a lazy triggering condition that launches full GC when the old space becomes full. Since the *long-lived accumulated records* have occupied 98% of the old space as shown in Figure 9a, Parallel GC constantly launches full GCs to perform object reclamation. However, these full GCs are **unnecessary** because the *long-lived accumulated records* cannot be reclaimed until the output phase finishes. In contrast, CMS and G1 use *aggressive* triggering conditions that start the GC cycle before the old space is exhausted. G1 GC starts a full GC cycle when the heap usage reaches a certain threshold (default 45% of the heap space). CMS GC starts a full GC cycle at a higher threshold (default 92% of the old space) and according to the runtime estimation of when the old generation will be exhausted. Since the *long-lived accumulated records* exceed the 45% threshold but has not reached the 92% threshold, G1 task suffers from consecutive full GC cycles while CMS task does not trigger full GC cycles in output phase.

**Implication:** Current threshold-based full GC triggering conditions tend to trigger unnecessary full GC pauses without being aware of the data objects' characteristics, e.g., sizes and lifecycles.

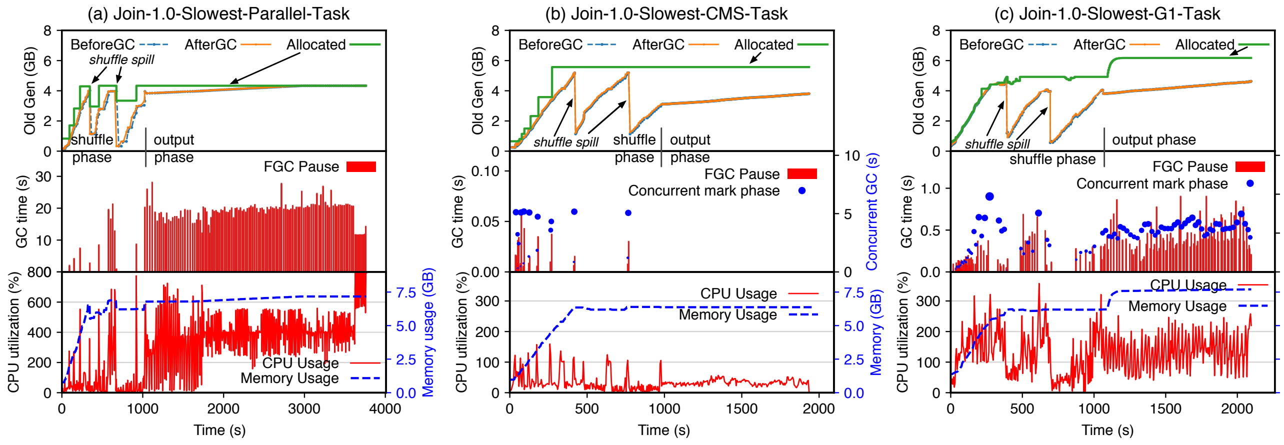

**Figure 9:** The memory usage and GC time comparison among the Join-1.0 slowest tasks. *FGC Pause* only illustrates the time of stop-the-world phases in each full GC cycle, including *initial-mark* and *remark* phases. The span time of each *concurrent mark* phase is illustrated by the diameter of the blue circle.

**Finding 8: Concurrent object marking algorithms used in CMS and G1 collectors are inefficient for handling *long-lived accumulated records* due to CPU contentions with CPU-intensive data operators.** As shown in Figure 8, CMS and G1 tasks have ∼95% shorter GC time but 1.6x longer data computation time than ParallelGC task. The root cause is that the concurrent marking algorithms in CMS and G1 have CPU contention with the data processing threads. Parallel collector uses stop-the-world object marking algorithm that pauses the data processing thread during each full GC. In contrast, both CMS and G1 collectors use concurrent marking algorithms that perform object marking in parallel with the data processing threads. While reclaiming the *long-lived accumulated records*, concurrent marking algorithms are CPU-intensive that degrade the simultaneous CPU-intensive data operators like *join()*. To mark the live objects, the concurrent marking algorithm needs to traverse the whole object graph. This marking step is CPU-intensive because the *long-lived accumulated records* are numerous (∼57 millions) and living in both shuffle and output phases. Unfortunately, the data operator *join*() is also CPU-intensive. As described in Section 3, *join*() operator needs to compute the Cartesian product of the rows with the same key from two tables, which has $O(n^2)$ time complexity and finally processes a large number of (∼38 millions) output records. Due to CPU contention, the concurrent marking algorithms slow down this CPU-intensive data computation in CMS and G1 tasks. Moreover, as interpreted in Finding 7 and shown in blue circles in Figure 9c, G1 task suffers from more full GC cycles (i.e., concurrent mark phases) than CMS task in output phase. Therefore, the CPU usage of G1 task is much higher than CMS task and G1 task's output phase is 1.2x longer than that of CMS task. Given that many Spark applications are CPU-intensive [46], such CPU contention between GC activities and Spark applications is common.

**Implication:** Concurrent marking algorithms reduce the GC pause time at the cost of degraded CPU-intensive Spark applications' performance. Given the prevalence of CPU-intensive big data applications, we need to design new object marking algorithm to balance GC pause and CPU usage of object marking.

## 4.4 SVM results

In this section, we explore the combined impact of *long-lived cached records* and *humongous data objects* while using SVM-0.5 as an example application.

### *4.4.1 Performance comparison results*

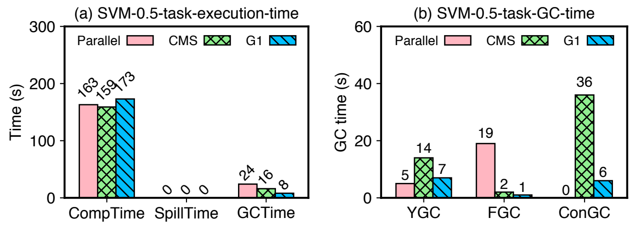

**Figure 10:** The execution time comparison among SVM-0.5 iterative reduce tasks is $CMS_{175s} < G1_{181s} < Parallel_{187s}$. These tasks have 6.4% execution time difference but up to 66% GC time difference.

The SVM-0.5 application has 10 iterations. Each iteration has a map stage (89 map tasks) and a reduce stage (8 reduce tasks). In both map and reduce tasks, the memory space is dominated by the *long-lived cached records* and *humongous data objects*. The performance differences only happen in reduce stages, so we pick the slowest reduce task in each reduce stage and sum their execution time together as the execution time of iterative reduce tasks. As shown in Figure 10, the iterative reduce tasks with different garbage collectors exhibit zero spill time and small (8%) data computation time difference. The reason is that SVM application has *lightweight* shuffle. Each SVM reduce task only needs to shuffle 12 records of $\langle grad\ vector, loss\ value\rangle$, compared to millions of shuffled records in GroupBy and Join. Moreover, these shuffled records are combined into only one $\langle \sum grad, \sum loss\rangle$ record in memory. So, most of the shuffled records are *short-lived* and do not lead to shuffle spill. However, each shuffled record is *humongous object* (about 200MB), which causes up to 95% full GC time difference as shown in Figure 10b. The root cause is that the *humongous objects* are directly allocated into old generation if the young generation does not have enough space [14]. Given that CMS has the smallest young space (only 600MB) and ∼20% of the old space is occupied by *long-lived cached records*, this direct allocation has led to a total of 24 full GC cycles. Parallel GC also suffers from frequent (in total 19) full GC pauses, because its smallest old generation cannot hold all the directly allocated humongous objects. However, G1 tasks only trigger 8 full GC pauses owing to its *eager humongous object reclamation mechanism* introduced in Java 8 [4]. This mechanism allows G1 to reclaim humongous objects at young GC instead of full GC. As a result, G1 tasks trigger 75% less full GCs than CMS tasks, and therefore leads to 83.3% shorter concurrent GC time. However, G1's humongous object management mechanism can lead to OOM error (Finding 9).

### 4.4.2 Findings and their implications

**Finding 9: For the applications with *humongous data objects*, G1's non-contiguous region-based heap management has heap fragmentation problem that leads to OOM errors.** We observe an unexpected OOM error while running SVM with data-1.0 using G1. The error occurs when there is a sufficiently large amount of free heap space left (about 3GB). We find the root causes are due to SVM application's humongous data objects and G1's heap fragmentation problem. (1) *Humongous objects*. After analyzing the OOM heap dump by Eclipse MAT [5], we find four humongous data objects of 417MB, including a parameter double array, a gradient double array, and two byte arrays. For G1 collector, these objects are *humongous objects* because their size exceeds the region size (default 1MB and up to 32MB). (2) *Heap fragmentation in G1*. Different from Parallel and CMS collectors' continuous heap layout, G1 uses region-based heap management that splits the heap space into equal-sized regions. The *humongous object* is directly allocated into humongous regions that contain a set of contiguous old regions [7]. However, G1 may not find enough contiguous old regions to accommodate SVM's many humongous objects even though there are many non-contiguous free regions. Although the *eager humongous object reclamation mechanism* can improve the efficiency of humongous object reclamation, it cannot solve this fragmentation issue. We have submitted this issue to OpenJDK community [16], and the developers plan to fix this defect in JDK 11 by a brute-force approach of launching additional full GC to move humongous objects [10]. According to JVM developer's suggestions, one way to decrease the chance of humongous object fragmentation is to increase G1's region size. We experiment with various region sizes and find that the OOM error disappears when the region size is enlarged to 16MB+. However, determining the right region size is challenging due to the unpredictable humongous object memory requirements exhibited by different big data applications and the intricacies of region-based heap management.

**Implication:** Region-based heap management uses fine-grained memory management that improves memory utilization. However, it is not suitable for managing humongous objects that require large and contiguous space. Increasing the region size can reduce the chance of OOM errors but also decrease the memory utilization. Since humongous object is a common pattern in big data applications, we should redesign the object allocation algorithm to balance the trade-off between memory utilization and reliability.

## 4.5 PageRank results

In this section, we explore the combined impact of *iterative long-lived accumulated records* and *long-lived cached records* while using PageRank-0.5 as an example application. We also find that the OOM errors in PageRank-1.0 applications are due to the memory leak in handling consecutive shuffle spills [19].

### 4.5.1 Performance comparison results

PageRank-0.5 application has a map stage (98 map tasks) and 10 iterative reduce stages (32 reduce tasks in each iteration). We only observe performance differences in reduce tasks, where the memory usage is dominated by *iterative long-lived accumulated records* and *long-lived cached records*. As described in Section 3, *iterative long-lived accumulated records* refer to the shuffled records that are accumulated in memory in each iteration. We merge the slowest task in each iteration as an iterative task, and compare the task execution time in Figure 11. Figure 11a shows that CMS and G1 tasks has 1.1∼2x longer data computation time than ParallelGC tasks. The root cause is due to the CPU-intensive concurrent object marking as interpreted in Finding 8. The decomposed GC time in Figure 11b shows that ParallelGC tasks suffer up to 142x longer full GC time than CMS and G1 tasks. The root causes are due to the stop-the-world object marking and the full GC triggering conditions interpreted in Finding 4 and 7. Finally, we observe that G1 tasks suffer from 16x longer full GC time than CMS tasks, due to the different object sweeping algorithms (Finding 10).

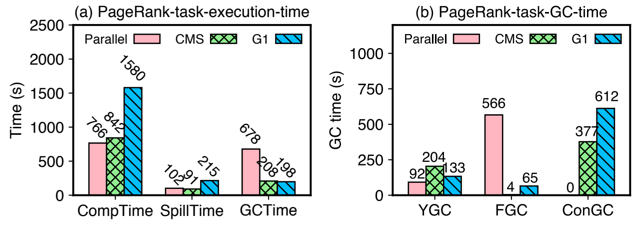

**Figure 11:** The execution time comparison among the PageRank-0.5 tasks is $\mathit{CMS}_{19m} < \mathit{Parallel}_{26m} < \mathit{G1}_{33m}$. G1 task is 1.7x slower than ParallelGC task due to its longest data computation time.

### 4.5.2 Findings and their implications

**Finding 10: For iterative applications that require to reclaim massive *long-lived accumulated records* in each iteration, CMS collector's concurrent sweeping algorithm achieves 16x shorter full GC time than G1's incremental sweeping algorithm.** Different from GroupBy and Join, PageRank application generates new *long-lived accumulated records* in each iteration during shuffle phase. Since these *long-lived accumulated records* are unused in the next iteration, the collector needs to reclaim a large (∼12 millions) number of records totaling 3.7GB after each iteration. This leads to heavy pressure on object marking and sweeping. Due to concurrent object marking algorithms, CMS and G1 tasks achieve 89-99% shorter full GC time than ParallelGC tasks. However, we find that G1 tasks suffer from 16x longer full GC time than CMS tasks. The root cause is that G1 uses a semi-concurrent *incremental sweeping algorithm* that is less efficient than CMS's concurrent sweeping algorithms for reclaiming *long-lived accumulated records* in iterative applications. In order to achieve predictable pause time, G1's sweeping algorithm reclaims the garbage regions incrementally in two phases according to the live object occupancy. (1) *A partly stop-the-world cleanup phase*. This phase occurs at the end of each full GC cycle for reclaiming the old regions without any live objects. It also selects the old regions that have fewer live objects than 85% percentage as candidate old regions for further collection. This phase totally leads to 4.5s full GC pause. (2) *A stop-the-world mixed collection phase*. This phase aims to reclaim the accumulated candidate old regions. Unlike non-iterative applications where most (∼95%) old regions are filled with non-reclaimable *long-lived accumulated records*, iterative applications' old regions are frequently allocated and reclaimed in each iteration. Therefore, many old regions will not be completely empty and selected as candidate old regions in previous *cleanup phase*. The PageRank task's GC log shows that the number of candidate old regions reaches 1,296 in the 8*th* iteration and the total reclaimable space of these regions achieves 7.34%. In this occasion, to keep the reclaimable space at a low level (<5% by default), G1 launches a *mixed collection* to reclaim both candidate old regions and young

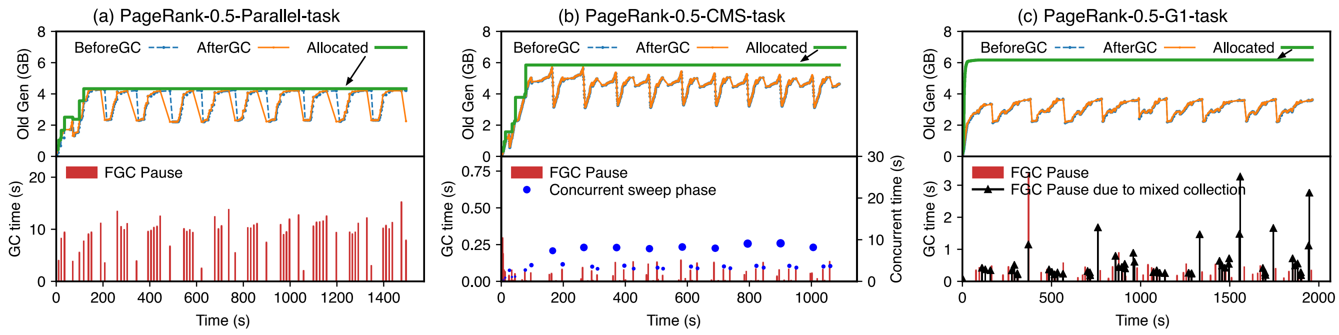

**Figure 12:** The memory usage and GC time comparison among PageRank-0.5 slowest tasks. ParallelGC tasks suffer from frequent and long full GC pauses due to STW object marking and sweeping. CMS tasks suffer from consecutive concurrent sweeping, while G1 tasks suffer from long *mixed collection* pauses.

regions. This *mixed collection* is stop-the-world that leads to long individual full GC pauses totaling 31s as shown in the triangle lines in Figure 12c. In contrast, as illustrated by the blue circles in Figure 12b, CMS sweeps unused objects concurrently with application threads and therefore does not incur full GC pause time.

**Implication:** Concurrent marking/sweeping algorithms are more efficient than stop-the-world or semi-concurrent ones for frequent memory reclamation of a large number of data objects, as commonly observed in iterative applications that have *long-lived accumulated records* to reclaim after each iteration.

## 5. LESSONS AND INSIGHTS

**To application developers:** Our key insights are to reduce *long-lived accumulated records* and *humongous objects* via memory-aware design, and choose proper garbage collectors according to the application features. (1) Finding 2 and 4 reveal that we should reduce memory usage of *long-lived accumulated objects* such as accumulated shuffled records. The solution is to use more memory-efficient aggregate data structures like *Compressed Buffer Tree* [25], or lower the space complexity of user-defined functions in data aggregation operators. We can simultaneously increase the *partition number* to reduce the shuffled records of each task. (2) Finding 9 reveals that we should avoid creating humongous objects such as big arrays. The solution is to split the large object into small objects or enlarge the region size of region-based collectors like G1. (3) The applications with data aggregation operators such as *groupByKey*() and *reduceByKey*() usually have *long-lived accumulated records* to reclaim. Therefore, we can choose concurrent collectors like CMS or G1 to reduce the time-consuming full GC pause. (4) Finding 8 reveals that the CPU-intensive data operators can be degraded by concurrent object marking/sweeping threads. The solution is to allocate more CPU cores for each task to alleviate the CPU contention between application threads and GC threads.

**To researchers:** Our findings reveal that the GC inefficiency is caused by the mismatch between big data object features (types, sizes, and lifecycles) and current GC algorithms. To minimize this mismatch, our insight is to co-design garbage collectors and frameworks' memory management mechanism, through estimating data objects' sizes and leveraging the relatively fixed lifecycles of data objects. In detail, we propose three new optimization methods.

**(1) Reduce the GC frequency via *prediction-based* heap sizing policy.** Current heap sizing policies are *history-based*. That is, the size of young/old generation is adjusted according to the historical GC statistics, such as average pause time and heap occupancy. However, as Spark applications exhibit vastly different memory usage patterns among different data processing phases, history-based adjustment cannot effectively match the applications' memory demands. For example, the memory usage of GroupBy's reduce tasks grows up quickly at the beginning of the shuffle phase, drops down at the time of shuffle spill, and then goes up again in merge output phase. Our solution is to provide a *prediction-based* heap sizing policy that adjusts the size of young/old generation based on the predicted memory usage. Since Spark framework manages the lifecycles of the shuffled/cached records, we can improve it by estimating the runtime sizes of big data objects. For example in shuffle phase, to estimate the total size of accumulated shuffled records, the framework can perform online linear regression to quantify the correlation between runtime memory usage in old space and current processed number of shuffled records. Since the total number of shuffled records is calculable, the framework can predict the future memory usage of accumulated records. Thus, garbage collectors can allocate proper old space to accommodate these long-lived objects and determine the GC triggering threshold.

**(2) Minimize the GC work via *lifecycle-aware* object marking algorithm.** Current object marking algorithms are *traverse-based* that need to traverse the whole object graph to mark the live objects. This traversing is unnecessary and time-consuming for big data applications with a large number of long-lived objects. Our insight is to reduce this object traverse through explicitly labeling the liveness of data objects. The concrete method is to let Spark framework explicitly tell the garbage collectors *which* data objects are long-lived shuffled/cached records and *when* they are unused. Once the shuffled records are spilled onto disk or the cached data are evacuated, the framework can signal the collector to perform the garbage collection on these data objects. Therefore, the collector can avoid periodically tracing these objects at each young/-full GC. This method is similar to but more lightweight than the lifetime-based memory management proposed in Deca [42], which uses multiple self-defined data containers (byte arrays) to manage the objects with different lifetimes. The key advantage of our proposed approach is that it does not need to determine the number of allocated data containers, merge the data objects into different lifetimes, nor manually reclaim the objects.

**(3) Minimize GC work in iterative applications via *overriding-based* object sweeping algorithm.** For iterative big data applications that have *long-lived accumulated objects* to reclaim in each iteration, current object sweeping algorithm is time-consuming because it needs to figure out each unreachable object and reclaim them one by one. However, data objects in iterative applications exhibit fixed lifecycles and fixed size. The *long-lived accumulated records* are allocated at the beginning of the iteration and can be reclaimed after each iteration. The exact size of these records can also be calculated after the first iteration, since iterative applications usually have similar memory usage pattern in each iteration. Our insight is to provide an *overriding-based* object sweeping algorithm using region-based reclamation. Unlike previous region-based memory management mechanisms [33, 45] that still need to explicitly track and sweep the data objects with different lifecycles, our algorithm allocates a fixed contiguous large region to accommodate the *long-lived accumulated records* generated in each iteration. When these records become unused at the end of current iteration, we do not reclaim these records one by one but treat the whole region as an empty region. In the next iteration, the new *long-lived accumulated records* are directly allocated in this region through overriding the old records. Therefore, we do not need to mark and sweep the old records in each iteration.

## 6. DISCUSSION

**DataFrames vs. RDDs.** Instead of RDDs, Spark SQL applications use DataFrames [29], whose intermediate data are managed by an optimized memory manager named Tungsten [17]. We compare the performance differences between DataFrames and RDDs on GroupBy and Join. These additional experimental results are available through our extended experimental report [13]. Our finding is that the three collectors' GC time drops from 1-41 min to $\sim$3s and the individual full GC pause time of ParallelGC tasks drops from $\sim$10s to 0.3s. The reason is that Tungsten performs SQL operations directly on binary data rather than Java objects. In other words, Tungsten stores the shuffled records in a serialized binary form and performs aggregation functions directly on the serialized objects. As a result, the number of in-memory Java objects is greatly reduced, which reduces the GC frequency and the object marking/sweeping. However, Tungsten is currently only applicable for specific SQL operators with some limitations. For example, it requires the operated data types are fixed-width types such as *Int*, *Double*, and *Date*. This technique can improve GC performance when it is applicable for RDD-based Spark applications.

**CPU/memory size variation** may have effects on GC behaviors, so we perform experiments on Join application with CPU and memory variation [13]. (1) We double CPU cores of each task and find that Parallel tasks' average full GC pause drops $\sim$36% due to more parallel GC threads. However, Parallel tasks still suffer from $\sim$10x longer individual full GC pause than CMS/G1 tasks due to SWT object marking/sweeping algorithm (as Finding 4). The computation time of CMS and G1 tasks drops $\sim$30% due to alleviated CPU contention with GC threads. However, CPU contention still exists and their computation time is still 1.5x longer than Parallel tasks (as Finding 8). (2) We lower 15% of the task's memory size. We find that Parallel tasks suffer from OOM errors due to the smallest old space allocated by heap sizing policy (as Finding 2). CMS and G1 tasks suffer from 1.4x more GC cycles due to higher memory pressure and static GC triggering conditions (as Finding 7).

**The generality of our findings.** Our current findings are specific to big data applications. Readers should interpret them under our experimental settings. However, if applications in other environments have similar memory usage patterns, some findings may be applicable. For example, Finding 2, 3, 4 may be applicable for other applications that generate a large number of long-lived objects.

## 7. RELATED WORK

**Performance studies on big data applications.** Many researchers have studied the performance of MapReduce and Spark applications [39, 38, 46, 49, 28]. They found the application performance can be affected by configuration parameters [38], I/O mode [39], data caching [26], straggler tasks [40, 27], scheduling [55], network traffic [34], checkpoint interval [48], etc. The study conducted by Ousterhout *et al.* [46] found that many Spark applications are CPU-bound. This supports the generality of Finding 8. Memory-related studies [33, 35, 51] show that big data applications suffer from serious memory pressure, such as memory bloat, heavy GC overhead, and even OOM errors. Bu *et al.* [33] found the memory bloat is caused by the large volume of data objects and the object metadata. Xu *et al.* [52] found that improper data caching policy and cache size can lead to performance degradation and GC overhead. Xu *et al.* [51] found the OOM root causes are due to the improper memory configurations, data skew, and memory-consuming user code. Our work leverages prior work but focuses on investigating the mismatch between the memory usage patterns of big data applications and current GC algorithms.

**Framework memory management optimization.** Researchers have proposed memory configuration tuning strategies [52], region-based or lifetime-based memory management [33, 37, 43, 45, 42] for improving memory utilization and GC optimization. MemTune [52] dynamically adjusts the data cache size and cache policy based on memory usage statistics. Bu *et al.* [33] proposed a region-based memory manager to reduce the object overhead, through merging small data objects into few large objects and manipulating them at the binary level. Facade [45] proposes a compiler and runtime system to bound the number of in-memory data objects, through storing data in an off-heap region and manipulating the data with control interfaces. Deca [42] proposes a lifetime-based memory manager to reduce GC overhead, through analyzing the lifetimes of the data objects in user-defined functions and grouping the objects with similar lifetimes into byte arrays.

**Garbage collection optimization for big data applications.** As mentioned in the survey [31], researchers begin to optimize existing garbage collectors [53, 41, 50, 30, 36] or design new garbage collectors [44, 32] for big data applications. For example, Suo *et al.* [50] proposed dynamic GC load balancing strategy to optimize the concurrency of Parallel GC in multicore environment. NG2C [32] extends G1 collector to reduce the time-consuming object copy between generations. Yak [44] divides the heap into a control space for managing the objects in the control path and a data space for managing the objects in the data path. The control space uses regular GC algorithms, while the data space uses region-based algorithms to reduce GC overhead. Our proposed GC optimizations are measurement-driven and can complement existing efforts.

## 8. CONCLUSION

Big data applications usually suffer from heavy GC overhead due to the mismatches between high memory requirements of these applications and current GC algorithms. In this work, we perform an in-depth study on three garbage collectors using four representative Spark applications. Specially, we investigate unique memory usage patterns of big data applications and GC inefficiencies in handling these patterns. Our study reveals many interesting findings and implications, as well as provides useful guidelines for application developers and insightful GC optimization strategies for designing big-data-friendly garbage collectors.

**Acknowledgements.** We thank Xingtong Ye for his contributions to profiler implementation and experimental setup, and Feng Zhu for his valuable feedback. This work is supported by the National Key R&D Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 61802377), Frontier Science Project of CAS (QYZDJ-SSW-JSC036), National Science Foundation (1755659, 1815619), Google Cloud Platform Research credits, and Youth Innovation Promotion Association at CAS.

## 9. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Apache Spark. http://spark.apache.org/.

[3] Caching in Spark. https://spark.apache.org/docs/latest/quick-start.html#caching.

[4] Early reclamation of large objects in G1. https://bugs.openjdk.java.net/browse/JDK-8027959.

[5] Eclispe Memory Analyzer (MAT). https://www.eclipse.org/mat/.

[6] G1 native memory consumption. http://mail.openjdk.java.net/pipermail/hotspot-gc-use/2017-February/002609.html.

[7] Garbage-First Garbage Collector. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html.

[8] GC Ergonomics. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html#ergonomics.

[9] HiBench: A big data benchmark suite. https://github.com/intel-hadoop/HiBench.

[10] [JDK-8191565] Last-ditch Full GC should also move humongous objects. https://bugs.openjdk.java.net/browse/JDK-8191565.

[11] JVM Generations. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html.

[12] KDD Cup 2012 dataset. https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html.

[13] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei. An Extended Experimental Report of Garbage Collectors on Big Data Applications. https://github.com/JerryLead/TR/blob/master/GC-Study-Extended-Experimental-Report.pdf.

[14] Memory Management in the Java HotSpot Virtual Machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

[15] MLlib: Apache Spark's scalable machine learning library. https://spark.apache.org/mllib/.

[16] OOM error caused by large array allocation in G1. http://mail.openjdk.java.net/pipermail/hotspot-gc-use/2017-November/002725.html.

[17] Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[18] Running Spark on YARN. https://spark.apache.org/docs/preview/running-on-yarn.html.

[19] [SPARK-22713] OOM errors caused by the memory contention and memory leak in TaskMemoryManager. https://issues.apache.org/jira/browse/SPARK-22713.

[20] Spark BigSQL Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[21] Spark executor GC taking long. http://stackoverflow.com/questions/38965787/spark-executor-gc-taking-long.

[22] SparkProfiler: Profiling Spark Applications for Performance Comparison and Diagnosis. https://github.com/JerryLead/SparkProfiler.

[23] Twitter social graph. http://an.kaist.ac.kr/traces/WWW2010.html.

[24] Why does my JVM have access to less memory than -Xmx specifies? https://plumbr.io/blog/memory-leaks/less-memory-than-xmx.

[25] H. Amur, W. Richter, D. G. Andersen, M. Kaminsky, K. Schwan, A. Balachandran, and E. Zawadzki. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, pages 18:1–18:16, 2013.

[26] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 267–280, 2012.

[27] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, pages 265–278, 2010.

[28] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling spark in the real world: Performance and usability. *PVLDB*, 8(12):1840–1843, 2015.

[29] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1383–1394, 2015.

[30] R. Bruno and P. Ferreira. POLM2: Automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware)*, pages 147–160, 2017.

[31] R. Bruno and P. Ferreira. A study on garbage collection algorithms for big data environments. *ACM Comput. Surv.*, 51(1):20:1–20:35, 2018.

[32] R. Bruno, L. P. Oliveira, and P. Ferreira. Ng2c: Pretenuring garbage collection with dynamic generations for hotspot big data applications. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 2–13, 2017.

[33] Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 119–130, 2013.

[34] P. Costa, A. Donnelly, A. I. T. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, 2012.

[35] L. Fang, K. Nguyen, G. H. Xu, B. Demsky, and S. Lu. Interruptible tasks: treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles, (SOSP)*, pages 394–409, 2015.

[36] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 661–673, 2015.

[37] 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 *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[38] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[39] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[40] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. SkewTune: Mitigating skew in mapreduce applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, 2012.

[41] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your JVM: understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 383–400, 2016.

[42] 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. *PVLDB*, 9(12):936–947, 2016.

[43] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[44] K. Nguyen, L. Fang, G. H. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 349–365, 2016.

[45] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. H. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 675–690, 2015.

[46] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–307, 2015.

[47] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 165–178, 2009.

[48] P. Sharma, T. Guo, X. He, D. E. Irwin, and P. J. Shenoy. Flint: batch-interactive data-intensive processing on transient servers. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 6:1–6:15, 2016.

[49] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB*, 8(13):2110–2121, 2015.

[50] K. Suo, J. Rao, H. Jiang, and W. Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the 13th EuroSys Conference (EuroSys)*, pages 35:1–35:15, 2018.

[51] L. Xu, W. Dou, F. Zhu, C. Gao, J. Liu, H. Zhong, and J. Wei. Experience report: A characteristic study on out of memory errors in distributed data-parallel applications. In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 518–529, 2015.

[52] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. MEMTUNE: Dynamic memory management for in-memory data analytic platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 383–392, 2016.

[53] Y. Yu, T. Lei, W. Zhang, H. Chen, and B. Zang. Performance analysis and optimization of full garbage collection in memory-hungry environments. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 123–130, 2016.

[54] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, 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 Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.

[55] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–42, 2008.