An Experimental Evaluation of Garbage Collectors on Big Data Applications

Lijie Xu\textsuperscript{1}, Tian Guo\textsuperscript{2}, Wensheng Dou\textsuperscript{1}, Wei Wang\textsuperscript{1}, Jun Wei\textsuperscript{1}

\textsuperscript{1} Institute of Software, Chinese Academy of Sciences
\textsuperscript{2} Worcester Polytechnic Institute

August 2019
Popular big data frameworks rely on garbage-collected languages to manage in-memory objects.
GC inefficiency

Big data applications suffer from **heavy GC overhead**
- GC time takes up to ~30% of Spark application execution time [StackOverflow][1]

Q: What are the causes of heavy GC overhead?
The causes of heavy GC overhead

Example: Spark Join Application

Application dataflow:

User code
Large intermediate computing results

Shuffle write/read
Large intermediate data

Cached data
Large cached data

In-memory objects

Objects managed by JVM

Cause 1:
Big data application generates massive data objects
⇒ GC is time-consuming
The causes of heavy GC overhead

1. User space for user code
2. Execution space for shuffle write/read
3. Storage space for cached data

Memory space managed by the framework

Cause 2:
The framework only manages the intermediate and cached data in a logical space

⇒ Rely on garbage collectors to manage the data objects

Objects managed by JVM

In-memory objects

Example: Spark Join Application

Shuffle write in-memory partition

Shuffle read in-memory aggregation

if cached

1, a
2, b
3, c
1, d
1, A
2, B
3, C
2, D

map()

1, a
2, b
3, c
1, d
1, A
2, B
3, C
2, D

reduce()

1, (a, A)
1, (d, A)
3, (c, C)
2, (b, B)
2, (b, D)

1 a
2 b
3 c
1 d
1 A
2 B
3 C
2 D

map()
The causes of heavy GC overhead

1. User space for user code
2. Execution space for shuffle write/read
3. Storage space for cached data

Memory space managed by the framework

Example: Spark Join Application

Shuffle write in-memory partition

1 a 2 b
3 c 1 d
1 A 2 B
3 C 2 D
map() map() map() map()

Shuffle read in-memory aggregation

1 2 3
1, a a d A
1, b b
2, c c C
3, d 1, d, A
1, c, C
1, (a, A)
1, (d, A)
3, (c, C)
reduce() reduce() reduce()

if cached

1, (a, A)
1, (d, A)
3, (c, C)

Causes of heavy GC overhead:

1. Fragmentation
2. Inefficiency of current garbage collectors
3. Current garbage collectors are not designed for big data applications
   (not aware of the characteristics of big data objects)
Three popular garbage collectors

JVM has three popular garbage collectors

- **Parallel**, **CMS**, and **G1** collectors
- One JVM uses only one collector at runtime
Three popular garbage collectors

**GC process:** *Mark unused objects* → *Sweep unused objects* → *Compact the space (optional)*

**Different GC algorithms**

**Parallel GC**
- Stop-the-world GC
- App threads
  - Stop the world
  - GC threads for marking
  - GC threads for sweeping

**CMS/G1 GC**
- Concurrent GC
- App threads
  - Stop the world
  - Concurrent marking
  - Concurrent sweeping
Research questions

Q1: Why are current garbage collectors inefficient for big data applications? Root causes?

Q2: Are there any GC optimization methods?
Methodology – Experimental evaluation

1. Select representative big data applications with different memory usage patterns

   SQL  
   Graph  
   Machine Learning

2. Run applications on different garbage collectors to identify GC patterns

   (Parallel, CMS, G1 collector)

3. Analyze the correlation between memory usage patterns and GC patterns to identify the causes of GC inefficiency
Application selection – memory usage patterns

1. GroupBy (from BigSQLBench)

Map Stage
- map()

Reduce Stage
- Memory usage pattern: Long-lived accumulated records

- GroupByKey
- spill()
- merge()

2. Join (from BigSQLBench)

Map Stage
- map()

Reduce Stage
- join()

- Massive temporary records
- Cartesian product

3. SVM (from Spark MLlib)

Map Stage
- Features
- Label
- Input matrix
- Long-lived cached records

Reduce Stage
- Humongous data objects
- gradient vector
- loss value
- compute(w_new)

- broadcast the new hyperplane vector w^T

4. PageRank (from Spark Graph library)

Map Stage
- Cashed data
- 1st Iterative Stage
- join()

Reduce Stage
- 2nd Iterative Stage
- 3rd Iterative Stage
- Long-lived accumulated records
Application selection – memory usage patterns

1. GroupBy (SQL)

Map Stage
- `map()`
- `1.1 a
  2.3 b`
- `3.5 c
  1.2 d`
- `1.6 A
  2.8 B`
- `3.7 C
  4.9 D`

Reduce Stage
- `groupByKey()`
- `spill()`
- `merge()`
- `1, [a, d, A]`
- `3, [c, C]`

Memory usage pattern:
- Long-lived accumulated records

2. Join (SQL)

Map Stage
- `map()`
- `1 a
  2 b`
- `3 c
  1 d`
- `1 A
  2 B`
- `3 C
  2 D`

Reduce Stage
- `join()`
- `Cartesian product`
- `massive temporary records`
- `1, [(a,d), A]`
- `3, [c, C]`

JVM heap
- Young Gen
- Old Gen

Shuffled records are accumulated in memory
⇒ Long-lived objects
⇒ Stored in Old Gen

P1: Long-lived accumulated records
Application selection – memory usage patterns

1. GroupBy (SQL)

Map Stage

<table>
<thead>
<tr>
<th>Map Stage</th>
<th>Reduce Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>map()</td>
<td>Memory usage pattern: Long-lived accumulated records</td>
</tr>
<tr>
<td>1.1 a</td>
<td>1, [a, d, A]</td>
</tr>
<tr>
<td>2.3 b</td>
<td>3, [c, C]</td>
</tr>
<tr>
<td>3.5 c</td>
<td>3, [c, C]</td>
</tr>
<tr>
<td>1.2 d</td>
<td>1, [a, d, A]</td>
</tr>
<tr>
<td>1.6 A</td>
<td>3, [c, C]</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2, [b, B]</td>
</tr>
<tr>
<td>3.7 C</td>
<td>4, [D]</td>
</tr>
<tr>
<td>4.9 D</td>
<td>2, [b, B]</td>
</tr>
</tbody>
</table>

Shuffled records are accumulated in memory
⇒ Long-lived objects
⇒ Stored in Old Gen

1, [a, d, A]
2, [b, B]
3, [c, C]
1, d
1, A
2, B
3, C
4, D

GroupByKey()

2. Join (SQL)

Map Stage

<table>
<thead>
<tr>
<th>Map Stage</th>
<th>Reduce Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>map()</td>
<td>Long-lived accumulated records</td>
</tr>
<tr>
<td>1 a</td>
<td>1, a</td>
</tr>
<tr>
<td>2 b</td>
<td>2, b</td>
</tr>
<tr>
<td>3 c</td>
<td>3, c</td>
</tr>
<tr>
<td>1 d</td>
<td>1, d</td>
</tr>
<tr>
<td>1 A</td>
<td>1, A</td>
</tr>
<tr>
<td>2 B</td>
<td>2, B</td>
</tr>
<tr>
<td>3 C</td>
<td>3, C</td>
</tr>
<tr>
<td>2 D</td>
<td>3, C</td>
</tr>
</tbody>
</table>

Cartesian product

Massive temporary records

P1: Massive temporary records

JVM heap

Young Gen

P2: Massive temporary records

Old Gen

P1: Long-lived accumulated records

Temporary results are generated in user code (e.g., cartesian())
⇒ short-lived objects
⇒ Stored in Young Gen
Application selection – memory usage patterns

Machine learning app: SVM with cached data

Input matrix
Long-lived cached records
Humongous data objects

Features
Label

Map Stage

Reduce Stage

Driver

\[ w^T \]

broadcast the new hyperplane vector \( w^T \)

JVM heap

Young Gen

Old Gen:

P1: Long-lived cached records

SVM stores training data in memory
⇒ Long-lived cached records
⇒ Sorted in Old Gen
Application selection – memory usage patterns

Machine learning app: SVM with cached data

SVM stores training data in memory
⇒ Long-lived cached records
⇒ Sorted in Old Gen

SVM generates large vectors (large arrays)
⇒ A vector achieves 345MB
⇒ Humongous data object
⇒ Stored in Old Gen

JVM heap

Young Gen

Old Gen:

P1: Long-lived cached records
P2: Humongous data objects
Application selection – memory usage patterns

Iterative Graph App: PageRank with cached data

1. Map Stage
   - Young Gen
   - Old Gen

2. 1st Iterative Stage
   - Cache data
   - 1, [2] (6, [3, 7])
   - 2, [1]
   - 3, [5, 6] (7, [4])
   - 4, [1]
   - 6, [3, 7]
   - 7, [4]

3. 2nd Iterative Stage
   - ReduceByKey
   - 1, [2] (6, [3, 7])
   - 2, [1]
   - 3, [5, 6] (7, [4])
   - 4, [1]
   - 6, [3, 7]
   - 7, [4]

4. 3rd Iterative Stage
   - ReduceByKey
   - 1, [2] (6, [3, 7])
   - 2, [1]
   - 3, [5, 6] (7, [4])
   - 4, [1]
   - 6, [3, 7]
   - 7, [4]

JVM heap

- Young Gen
- Old Gen

P1: Iterative long-lived accumulated records

PageRank generates shuffled records in each iteration
⇒ Iterative long-lived accumulated records
⇒ Similar memory usage pattern in each iteration
Application selection – memory usage patterns

Iterative Graph App: PageRank with cached data

PageRank generates shuffled records in each iteration
⇒ Iterative long-lived accumulated records
⇒ Similar memory usage pattern in each iteration
Application selection – memory usage patterns

Iterative Graph App: PageRank with cached data

1. Map Stage
2. Cached data
3. 1st Iterative Stage
   - join()
4.reduceByKey()
5. 2nd Iterative Stage
6. reduceByKey()
7. 3rd Iterative Stage
8. reduceByKey()

JVM heap

Young Gen

P1: Iterative long-lived accumulated records

Old Gen

P2: Long-lived cached records

Similar memory usage pattern in each iteration

(a) PageRank-0.5-Parallel-task

Graph showing memory usage over time with different patterns before and after garbage collection.
Methodology – Input data variation

Vary input data size to simulate different memory pressures

<table>
<thead>
<tr>
<th>Application</th>
<th>Data-1.0 (100%)</th>
<th>Data-0.5 (50%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupBy</td>
<td>200GB Uservisits (1.2B rows) [1]</td>
<td>50% rows (100GB)</td>
</tr>
<tr>
<td>Join</td>
<td>200GB Uservisits (1.2B rows), 40GB Rankings (600M rows) [1]</td>
<td>50% rows (100GB, 20GB)</td>
</tr>
<tr>
<td>SVM</td>
<td>21GB KDD2012 matrix [2] (149M rows, 54M features)</td>
<td>50% columns (11.2GB, 27M features)</td>
</tr>
<tr>
<td>PageRank</td>
<td>25GB Twitter graph [3] (476M edges, 17M nodes)</td>
<td>50% edges (12.2GB, 238M edges)</td>
</tr>
</tbody>
</table>

Testbed

Alibaba Cloud 9 nodes, Java 8, Spark 2.1.2

Each node runs 4 JVMs (tasks)

⇒ Observe the memory usage and GC activities of each task
Observation 1: Applications with long-lived accumulated records suffer from high GC overhead (1-40 mins)
Where are long-lived accumulated records from?

Shuffled records are aggregated in memory

Massive long-lived accumulated records

e.g., ~40M records (5.5GB)
Memory usage pattern of long-lived accumulated records

Shuffled records are aggregated in memory

Massive long-lived accumulated records
e.g., ~40M records (5.5GB)

JVM

Young Gen: for storing short-lived objects
Old Gen: for storing long-lived objects

Require large old generation to store the increasing accumulated records
Memory usage pattern of long-lived accumulated records

GroupBy
Join
PageRank
Application

Map Stage

Reduce Stage

Require large old generation to store the increasing accumulated records

Shuffled records are aggregated in memory

Massive long-lived accumulated records
e.g., ~40M records (5.5GB)

JVM

Young Gen: for storing short-lived objects

If old generation is limited => frequent GCs

Time

memory size

frequent and long GC pauses

memory usage
Findings about the garbage collectors

Different heap sizing policies lead to different GC frequencies

(a) GroupBy-1.0-Slowest-Parallel-Task

(b) GroupBy-1.0-Slowest-CMS-Task

(c) GroupBy-1.0-Slowest-G1-Task

Parallel GC
Allocate the smallest old gen
& shrink with memory usage
⇒ Suffer from frequent GC

CMS GC
Allocate the largest old gen
& does not shrink
⇒ The fewest GC pauses

G1 GC
Allocate large old gen
& shrink with memory usage
⇒ Less GCs than Parallel
Findings about GC inefficiency

**Finding 1:** Current collectors allocate limited old generation based on historical and current memory usage => frequent GC pauses

<table>
<thead>
<tr>
<th></th>
<th>Old Gen (GB)</th>
<th>BeforeGC</th>
<th>AfterGC</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel GC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CMS GC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>G1 GC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A long (11 s) full GC pause due to concurrent mode failure.
Implication about the heap sizing policy

(a) GroupBy-1.0-Slowest-Parallel-Task

(b) GroupBy-1.0-Slowest-CMS-Task

(c) GroupBy-1.0-Slowest-G1-Task

Implication:

Design more intelligent heap sizing policies for accommodating big data objects (especially for long-lived accumulated records)
Experimental results – Observation 2

App execution time and GC time comparison

GroupBy-1.0 (20.1%)
- App Execution Time (min):
  - Parallel: 45.4
  - CMS: 36.3
  - G1: 39.4
- GC Time (min):
  - Parallel: 19
  - CMS: 0.9
  - G1: 1.2

Join-1.0 (30.5%)
- App Execution Time (min):
  - Parallel: 78.7
  - CMS: 54.7
  - G1: 57.1
- GC Time (min):
  - Parallel: 41
  - CMS: 0.7
  - G1: 2.6

SVM-0.5 (3.2%)
- App Execution Time (min):
  - Parallel: 6.2
  - CMS: 6
  - G1: 6
- GC Time (min):
  - Parallel: 0.4
  - CMS: 0.3
  - G1: 0.1

PageRank-0.5 (49.1%)
- App Execution Time (min):
  - Parallel: 26.1
  - CMS: 19.5
  - G1: 38.3
- GC Time (min):
  - Parallel: 11.3
  - CMS: 3.5
  - G1: 3.3

Observation 2: Applications with **Parallel GC** suffer from longer app execution and GC time than applications with **CMS/G1 GC**
GC process

1. Mark live objects
2. Sweep unused objects
Big data applications have too many objects to mark and reclaim in each GC activity
⇒ Long individual GC pause (10-20s)
Why Parallel GC is slower than CMS/G1 GC?

Finding 2:
Parallel collector’s stop-the-world marking/sweeping algorithm leads to more pause time (10-20s)

Parallel GC
Stop-the-world GC
App threads
Stop the world
Stop-the-world
App threads

CMS/G1 GC
Concurrent marking/sweeping
App threads
Stop the world
Stop the world
Stop the world
App threads

Stop the app threads for GC (10-20s pause)
Concurrent marking & sweeping (less than 1s pause for remark)
Concurrent garbage collectors

Question:
Are concurrent collectors good enough for big data applications?
Finding about marking & sweeping algorithm

Finding 3: CMS and G1 collectors’ concurrent marking algorithms suffer from CPU contentions with CPU-intensive data operators.

CMS/G1 GC
Concurrent marking/sweeping

App threads

Stop the world

Stop the world

Stop the world

GC thread has CPU contention with App threads
Finding about marking & sweeping algorithm

Finding 3: CMS and G1 collectors’ concurrent marking algorithms suffer from CPU contentions with CPU-intensive data operators

CMS/G1 GC
Concurrent marking/sweeping

App threads

Stop the world

Stop the world

Stop the world

App threads

GC thread has CPU contention with App threads

Leads to high CPU usage during concurrent GC activities
Implication about GC algorithm

**CMS/G1 GC**

Concurrent marking/sweeping

App threads

Stop the world

Stop the world

Stop the world

App threads

**Implication:**

Design more efficient marking & sweeping algorithms for big data objects to reduce individual GC pause time.
Observation 3: Applications with G1 collector suffers from OutOfMemory error while processing humongous objects in SVM application.
Root cause of the OOM error

Humongous data object (~500 MB vector array)
Root cause of the OOM error

Humongous data object
(~500 MB vector array)

Root cause:
Not enough contiguous space for keeping this humongous object
Root cause of the OOM error

Humongous data object
(~500 MB vector array)

G1 JVM

Implication
Redesign the object allocation algorithm to balance the trade-off between memory utilization and reliability
The other findings

(see our paper for details)

- 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.

- 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.

- 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.

- 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.

- G1 tasks suffer from 1.1-1.2x higher physical memory usage than ParallelGC and CMS tasks.

- Compared to CMS and G1 collectors, Parallel collector’s inappropriate generation resizing timing mechanism leads to 38% more full GC pauses.
GC optimization methods

Propose three GC optimization methods

• Prediction-based dynamic heap sizing policies

• Label-based object marking algorithms
  ⇒ Explicitly label the long-lived data objects based on lifecycles to avoid unnecessary marking

• Overriding-based object reclamation algorithms
Related work

Performance studies on big data applications
• PerformanceStudyOnSpark [NSDI 2015], StudyOnMemoryBloat [ISMM 2013], OutOfMemoryStudy [ISSRE 2015]

Framework memory management optimization
• MemTune [IPDPS 2016], Broom [HotOS 2015], Façade [ASPLOS 2015], Deca [VLDB 2016], Tungsten [SparkSQL]

GC optimization for big data applications
• Yak [OSDI 2016], NG2C [ISMM 2017]
Conclusions

• Summarize the unique memory usage patterns of big data applications

• Experimental evaluation on three collectors and identify the root causes of GC inefficiencies

• Propose GC optimization methods
Q & A
Thanks!