中国科学院软件研究所学术年会'2019
暨计算机科学国家重点实验室开放周

学术论文

An Experimental Evaluation of Garbage Collectors on Big Data Applications

Lijie Xu[1], Tian Guo[2], Wensheng Dou[1], Wei Wang[1], Jun Wei[1]

[1] Institute of Software, Chinese Academy of Sciences [2] Worcester Polytechnic Institute

The 45th International Conference on Very Large Data Bases (VLDB 2019), pages 570-583

xulijie@iscas.ac.cn

Introduction

Big data frameworks, such as Hadoop MapReduce and Spark, rely on garbage-collected languages. Big data applications usually process a large volume of data that lead to heavy GC overhead (up to 50% of the application execution time).

Three Key Research Questions

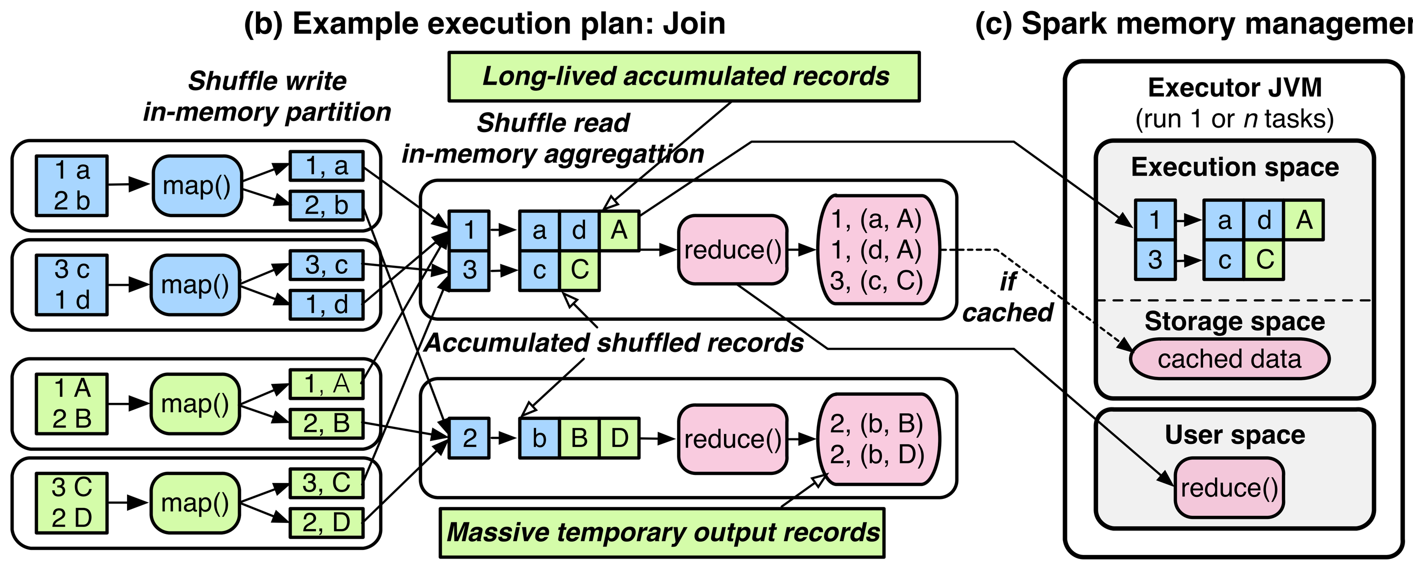
- RQ1: What are the typical memory usage patterns of big data applications?
- RQ2: Are current garbage collectors sufficient for big data applications? If not, why?
- RQ3: What are the guidelines for application developers and insights for designing big-data-friendly garbage collectors?

Background

Spark Memory Management

The memory usage of a Spark application consists of three parts:

1. **Execution space:** for storing shuffled data
2. **Storage space:** for storing cached data
3. **User space:** for storing operator-generated data

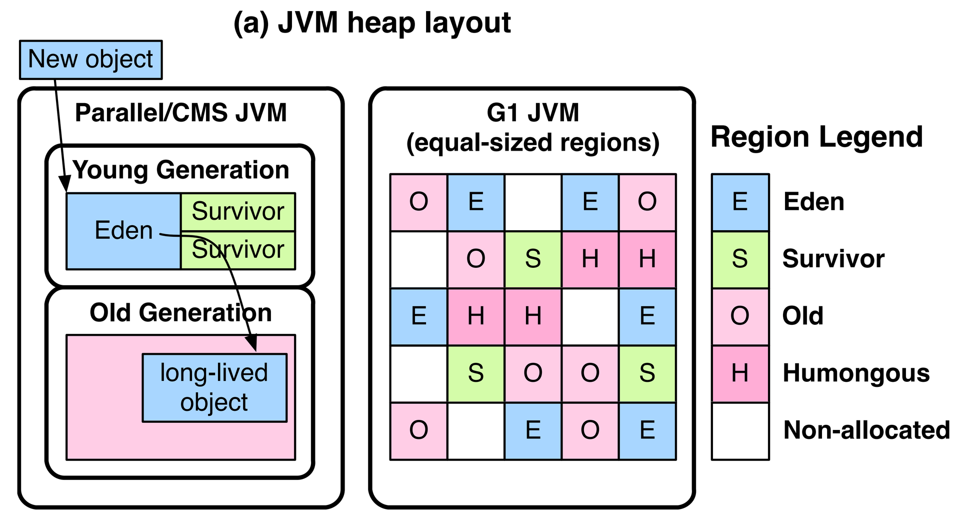


(b) Example execution plan: Join (c) Spark memory management

Different Garbage Collectors

1. Heap Layout Differences:

Garbage collectors divide the heap into two generations:
Young generation: for keeping short-lived objects
Old generation: for keeping long-lived objects



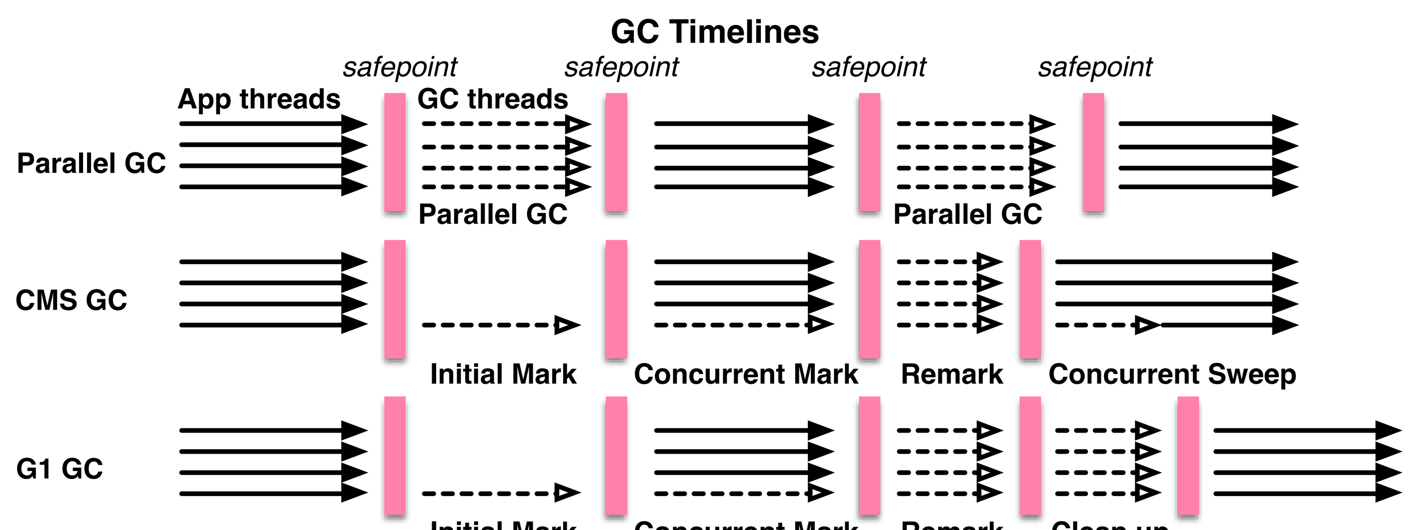
(a) JVM heap layout

GC	Heap Layout Differences
Parallel, CMS	Contiguous generations with an explicit boundary
G1	Dividing heap space into equal-sized regions

2. GC Algorithm Differences:

GC	Young GC	Full GC
Parallel	Mark-copy (STW)	Mark-sweep-compact (STW)
CMS	Mark-copy (STW)	Concurrent mark-sweep (mostly concurrent, non-compacting)
G1	Mark-copy (STW)	Concurrent mark + mixed evacuation (mostly concurrent,incremental compact)

GC Timelines

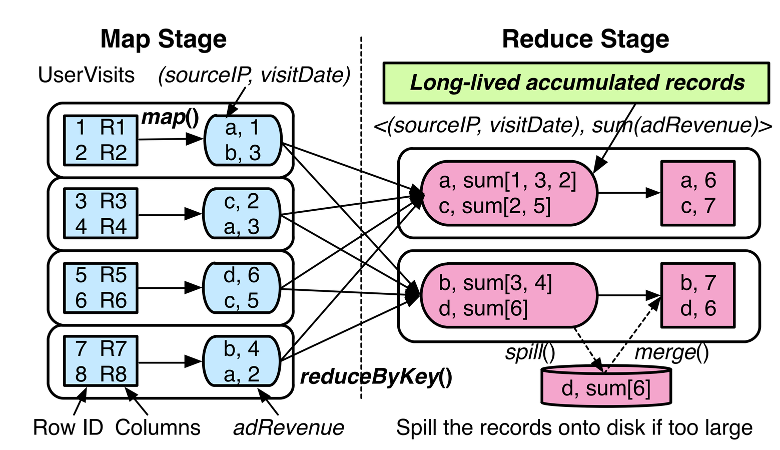


Methodology

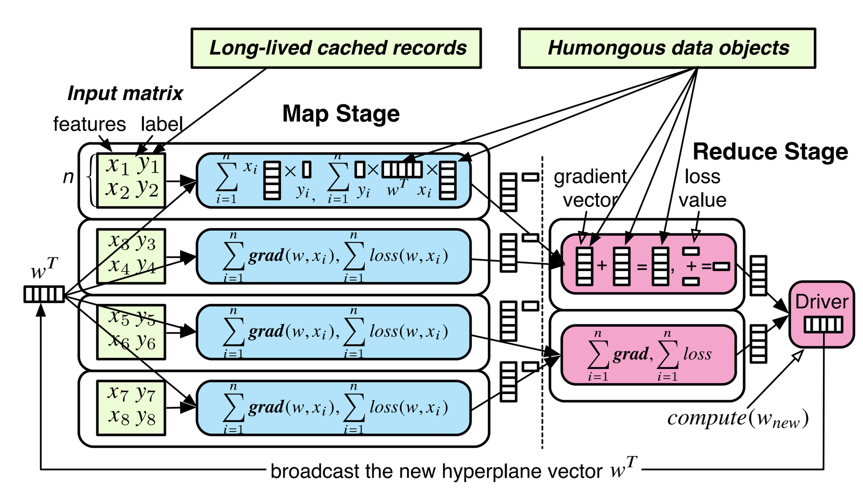
We summarize the computation features and memory usage patterns of four representative Spark applications:

Application	Type	Application features: #Cached data	#Shuffled records	Space complexity	Memory usage patterns
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: $O(N_{map\_task})$	$reduce() : O(\lvert x \rvert)$	*Humongous data objects*, ***Cached records***
PageRank	Graph	$O(N_{edges})$	Medium: $O(N_{edges})$	$join() : O(m+n)$	*Iterative* ***accumulated records***, ***Cached records***

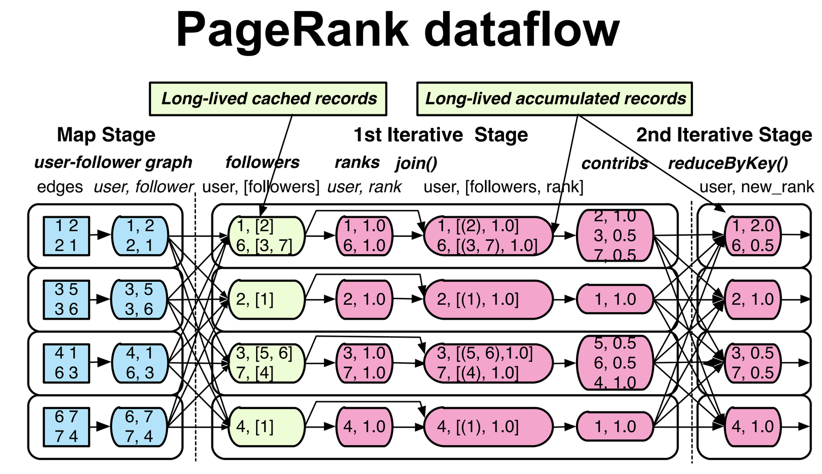
GroupBy dataflow



SVM dataflow



PageRank dataflow



Experimental datasets

App	Data-1.0 (100%)	Data-0.5 (50%)
GroupBy	200GB Uservisits	50% rows
Join	200GB Uservisits, 40GB Rankings	50% rows
SVM	21GB KDD2012 matrix	50% columns
PageRank	25GB Twitter graph	50% edges

We perform the four applications on representative datasets with different sizes that can lead to different memory pressures.

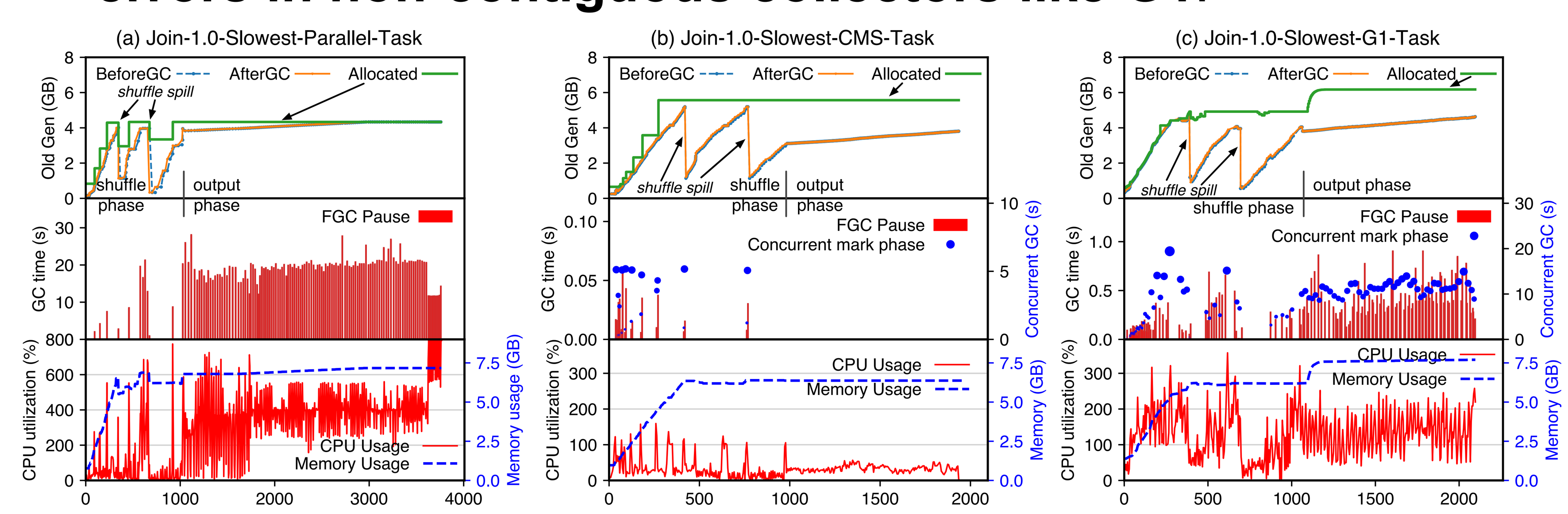
Results

Overall Results and Key Findings

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 Parallel	CMS	G1	Comparison	Data-1.0 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)	×

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.**



(a) Join-1.0-Slowest-Parallel-Task (b) Join-1.0-Slowest-CMS-Task (c) Join-1.0-Slowest-G1-Task

Proposed Optimizations

1. Prediction-based dynamic heap sizing policies.
2. Label-based object marking algorithms.
3. Overriding-based object reclamation algorithms.