An Experimental Evaluation of Garbage Collectors on Spark Applications

Technical Report

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

1. Experimental environments

We perform the evaluation on a cluster of nine *mn4.2xlarge* server 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 by default. 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. Before each run, we clear the OS buffer caches and restart the workers to eliminate the cache effects. In the following sections, we present the experimental results on several representative Spark applications.

2. Join-200G with CPU variation

2.1 Application description

Join is a SQL application simplified from the join query in the benchmark [1, 2] as follows.

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

This SQL application is implemented with basic RDD-based APIs. The map tasks transform each row of table *Rankings* to <*URL*, *pageRank*> record and transform each row of table *UserVisits* to <*URL*, *adRevenue*> record. In shuffle phase, each reduce task performs *join*() operator to group the two tables' rows with the same key as <*URL*, *list*(*pageRanks*, *adRevenues*)>. These grouped shuffle records are kept in memory as *long-lived accumulated records*. In output phase, the *join*() operator calculates the Cartesian product of the two sets *pageRanks* and *adRevenues*, and output <*URL*, *pageRank*, *adRevenue*> record one by one. Since these records are directly output into HDFS, they are regarded as *massive temporary output records*. This application suffers from *heavy* shuffle, since the shuffled records is the sum of the number of rows from *Rankings* and *UserVisits* tables.

2.2 Input data

The dataset is generated by HiBench [3].

Application	Input data size
Join-200G	200GB Uservisits (1.2B rows) 40GB Rankings (600M rows)

2.3 Configurations

We varied the CPU cores for each task from 1 to 2, while fixing the memory size of each executor JVM as 6.5GB.

Name	Executor CPU	Executor Memory
Join-CPU-1	1 (1 core for each task)	6.5 GB
Join-CPU-2	2 (2 cores for each task)	6.5 GB

2.4 Experimental results

2.4.1 Performance comparison results

We only observed performance differences in reduce stage and compared the performance of the slowest reduce task as shown in Table 1.

Table 1: The performance comparison among the slowest reduce tasks with different CPU configurations and garbage collectors.

Name	The performance of the slowest reduce tasks						
	GC	T_{task}	T_{comp}	T_{GC}	T_{YGC}	T_{FGC}	T_{conGC}
Join-CPU-1	Parallel	62m	19m	41m	27s	2436s	0s
Join-CPU-2	Parallel	41m	14m	25m	28s	1504s	0s
Join-CPU-1	CMS	32m	30m	31s	30s	1s	64s
Join-CPU-2	CMS	23m	21m	29s	31s	1s	77s
Join-CPU-1	G1	35m	31m	120s	80s	40s	631s
Join-CPU-2	G1	25m	23m	66s	46s	24s	395s

Figure 1 illustrates the time-series memory usage and GC pause time comparison among the slowest Join-CPU-2 reduce tasks with different garbage collectors.

2.4.2 Findings and implications

Finding 1: After doubling the CPU cores of each task in the applications with *long-lived accumulated results*, the tasks' execution time drops ~30% and GC time drops 6-45%. The root causes are that (1) *The collectors launch more GC threads to perform the GC work that reduces the individual GC pause time*. For example, the average full GC pause time of Parallel tasks drops from to 18.2s to 11.5s, due to more parallel GC threads to perform the GC work. (2) *The CPU contention between the application threads and GC threads is reduced.* For CMS and G1 tasks, the computation time drops ~30% due to alleviated CPU

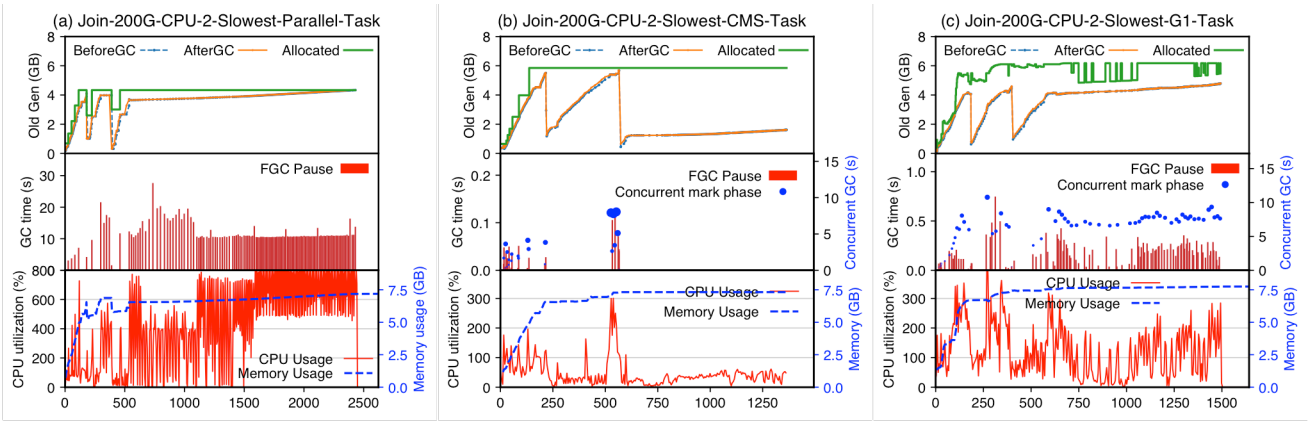

Figure 1: The memory usage and GC pause time comparison among the Join-CPU-2 slowest tasks. *FGC Pause* only illustrates the time of stop-the-world phases in each full GC cycles, including *initial-mark*, *remark*, and *cleanup* phases. The *concurrent-mark* phase is illustrated by the blue circle, and the diameter of the circle denotes the span time of the concurrent-mark phase.

contention between data operators and GC threads. However, this CPU enlargement cannot reduce the GC frequencies.

Implication: For the applications with CPU-intensive operators and *long-lived accumulated records*, it is better to allocate more CPU cores for each task to avoid long GC pause and serious CPU contention. Furthermore, we need to design intelligent policies to balance CPU utilization between the tasks and garbage collectors.

3. Join-200G with memory variation

3.1 Application and Input data

The application and datasets are the same with that in "*Join-200G with CPU variation*" in Section 2.

3.2 Configurations

We varied the memory for each task from 6.5G to 5.5G, while fixing the CPU cores of each task to be 1.

Name	Executor CPU	Executor Memory
Join-6.5GB	1 (1 core for each task)	6.5 GB
Join-5.5GB	1 (1 core for each task)	5.5 GB

3.3 Experimental results

3.3.1 Performance comparison results

We only observed performance differences in reduce stage and compared the performance of the slowest reduce tasks as shown in Table 2.

Table 2: The performance comparison among the slowest reduce tasks with different memory configurations.

Name	The performance of the slowest reduce task						
	GC	T_{task}	T_{comp}	T_{GC}	T_{YGC}	T_{FGC}	T_{conGC}
Join-6.5G	Parallel	62m	19m	41m	27s	2436s	0s
Join-5.5G	Parallel	OOM	-	-	-	-	-
Join-6.5G	CMS	32m	30m	31s	30s	1s	64s
Join-5.5G	CMS	34m	32m	38s	37s	1s	288s
Join-6.5G	G1	35m	31m	120s	80s	40s	631s
Join-5.5G	G1	36m	33m	164s	110s	54s	743s

3.3.2 Findings and implications

Finding 2: After lowering 15% of the task's memory size in the application with *long-lived accumulated results*, Parallel tasks suffer from OOM errors and CMS/G1 tasks suffer from 3-6% longer execution time and 18-27% longer GC time. The OOM root cause is that Parallel collector suffers from the smallest old space due to its heap sizing policy. Small old space cannot accommodate the accumulated shuffled records plus the extra buffers generated in shuffle spill. For CMS and G1 tasks, the performance degradation is caused by the increased GC cycles under higher memory pressure.

Implication: Given input data size, we need an intelligent memory estimator to determine the proper memory size to avoid OOM errors and reduce the GC overhead.

4. SQLGroupBy with DataFrame APIs

4.1 Application description

GroupBy is a SQL application simplified from the aggregation query in Spark's BigSQL benchmark [1, 2]. The *sourceIP*, *visitDate*, and *adRevenue* are three columns from table *UserVisits*.

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

We can use both RDD-based and DataFrame-based APIs to implement this application. For performance comparison, the RDD-based implementation is named as *RDDGroupBy*, and the DataFrame-based implementation is named as *SQLGroupBy*.

4.2 Input data

The dataset is generated by HiBench [3].

Application	Input data size
SQLGroupBy	200GB Uservisits (1.2B rows)

4.3 Configurations

Name	Executor CPU	Executor Memory
SQLGroupBy	1 (1 core for each task)	6.5 GB
RDDGroupBy	1 (1 core for each task)	6.5 GB

4.4 Experimental results

4.4.1 Performance comparison results

We only observed performance differences in reduce stage and compared the performance of the slowest reduce tasks as shown in Table 3.

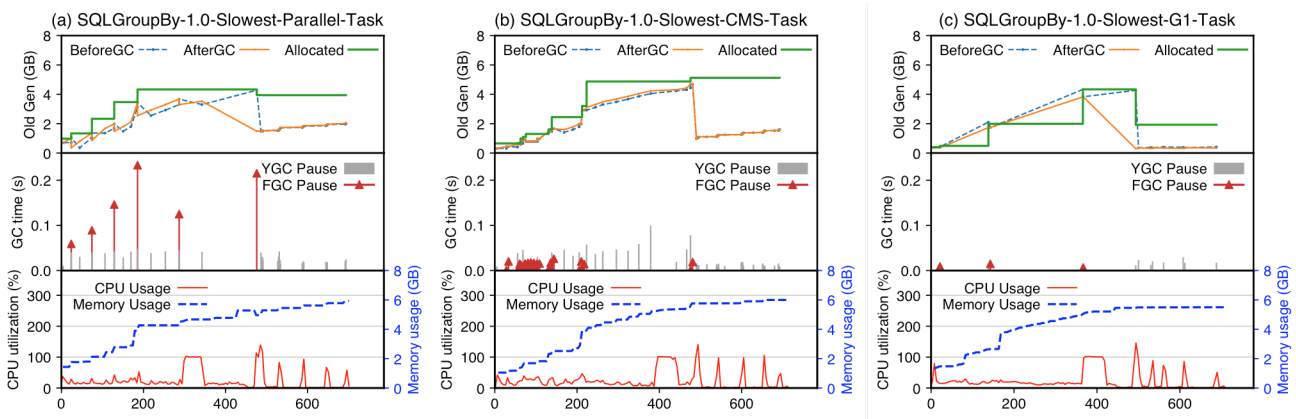

Figure 2: The memory usage and GC pause time comparison among the SQLGroupBy slowest reduce tasks on 200GB dataset.

Table 3: The performance comparison among the slowest reduce tasks with RDD and DataFrame APIs.

Name	The performance of the slowest reduce task						
	GC	T_{task}	T_{comp}	T_{GC}	T_{YGC}	T_{FGC}	T_{conGC}
RDDGroupBy	*P*	24m	6m	17m	28s	16m	0s
SQLGroupBy	*P*	12m	12m	3s	2s	1s	0s
RDDGroupBy	*C*	17m	15m	47s	30s	17s	129s
SQLGroupBy	*C*	12m	12m	4s	3s	0.4s	17s
RDDGroupBy	*G1*	20m	15m	55s	45s	19s	264s
SQLGroupBy	*G1*	12m	12m	1.2s	1s	0.2s	0.1s

* *P* refers to Parallel collector, while *C* refers to CMS collector.

Figure 2 illustrates the time-series memory usage and GC pause time comparison among the slowest SQLGroupBy reduce tasks with different garbage collectors.

4.4.2 Findings

Finding 3: Three collectors' GC time drops down from 1-17 min to ~4s, and the individual full GC pause time of ParallelGC tasks drops down from ~10 s to ~0.3 s. The reason is that Spark SQL adopts an explicit memory manager named Tungsten, which is highly optimized for SQL operations [4]. Tungsten reduced the number and size of in-memory data objects through operating many SQL operations directly against binary data rather than Java objects. It stores the shuffled records in serialized binary form, compacts them into map-like binary data structure, and performs aggregation functions directly on the serialized objects. As a result, the volume of in-memory Java objects are greatly reduced, which further reduces the GC frequency and the work of object marking and sweeping. However, Tungsten is currently only applicable for specific SQL operators with many limitations. (1) It requires the operated data types are fixed-width types such as primitive *Int*, *Double*, and *Date*. Variable-length types like *String* are not supported. (2) The aggregation functions are able to be performed on serialized data with fixed-width results. Thus, the aggregation functions do not need to deserialize the binary objects into Java objects and do not generate intermediate Java objects during aggregation. In Spark SQL, typical aggregation functions such as SUM and AVG satisfy this requirement, but many user-defined aggregation functions (UDAF) may generate complex user-defined types that cannot fit Tungsten's internal format.

5. SQLJoin with DataFrames APIs

5.1 Application description

Join is a SQL application simplified from the join query in the benchmark [1, 2]. The *pageRank* is a column from table *Rankings* and *adRevenue* is a column from table *UserVisits*.

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

We can use both RDD-based and DataFrame-based APIs to implement this application. For performance comparison, the RDD-based implementation is named as *RDDJoin*, and the DataFrame-based implementation is named as *SQLJoin*.

5.2 Input data

The dataset is generated by HiBench [3].

Application	Input data size
SQLJoin and RDDJoin	200GB Uservisits (1.2B rows) 40GB Rankings (600M rows)

5.3 Configurations

Name	Executor CPU	Executor Memory
SQLJoin	1 (1 core for each task)	6.5 GB
RDDJoin	1 (1 core for each task)	6.5 GB

5.4 Experimental results

5.4.1 Performance comparison results

We only observed performance differences in reduce stage and compared the performance of the slowest reduce task as follows.

Table 4: The performance comparison among the slowest reduce tasks with different APIs.

Name	The performance of the slowest reduce task						
	GC	T_{task}	T_{comp}	T_{GC}	T_{YGC}	T_{FGC}	T_{conGC}
RDDJoin	Parallel	62m	19m	41m	27s	2436s	0s
SQLJoin	Parallel	24m	24m	3s	2s	1s	0s
RDDJoin	CMS	32m	30m	31s	30s	1s	64s
SQLJoin	CMS	24m	24m	5s	4s	1s	51s
RDDJoin	G1	35m	31m	120s	80s	40s	631s
SQLJoin	G1	24m	24m	2s	1.5s	0.2s	0.2s

* *P* refers to Parallel collector, while *C* refers to CMS collector.

5.4.2 Findings

Finding 4: Three collectors' GC time drops down from 1~41 min to ~5s, and the individual full GC pause time of ParallelGC tasks drops down from ~10 s to ~0.3 s. The reasons are the same with that in Section 4.4.2.

6. GroupBy on small dataset (16G)

6.1 Application description

GroupBy is a SQL application simplified from the aggregation query in Spark's BigSQL benchmark [1, 2]. Figure 2 illustrates the GroupBy dataflow where *sourceIP* denotes an IP address and is a column of table *UserVisits*.

```
SELECT * FROM UserVisits
GROUP BY SUBSTR(sourceIP, 1, 7);
```

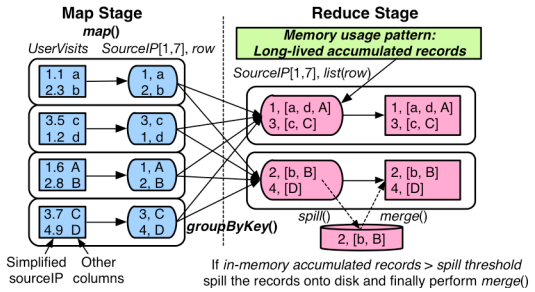

Figure 3: The dataflow of GroupBy application.

The map tasks perform *map*() that transforms each *row* of table *UserVisits* to <*sourceIP*[1,7], *row*> (the first 7 characters of *sourceIP*). The space complexity of *map*() is $O(1)$. In reduce stage, each reduce task performs *groupByKey*() that groups the shuffled records with the same key to <*sourceIP*[1,7], *list*(*row*)>. The space complexity of *groupByKey*() is $O(n)$, where n represents the length of *list*(*row*). *groupByKey*() exhibits memory usage pattern of *long-lived accumulated records* because the <*sourceIP*[1,7], *list*(*rows*)> records are accumulated in a *HashMap*-like data structure and remain in memory until either being spilled onto disk or the completion of reduce task. The number of the shuffled records of reduce tasks is equal to the number of rows in *UserVisits*.

6.2 Input data

The dataset is generated by HiBench [3].

Application	Input dataset
GroupBy-1.0	16GB Uservisits (90M rows)
GroupBy-0.5	8GB Uservisits (45M rows)

6.3 Experimental results

Table 5: The applications' execution time on different GCs.

Application	The application execution time		
	Parallel	CMS	G1
GroupBy-0.5	$3.5_{(1.1)}$	$2.8_{(0.5)}$	$2.7_{(0.3)}$
GroupBy-1.0	× (OOM)	× (OOM)	× (OOM)

In this section, we explore the impact of *long-lived accumulated records* on application's shuffle spill and GC performance using GroupBy-0.5 as an example.

6.3.1 Performance comparison results

GroupBy-0.5 application consists of a map stage (64 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 the *long-lived accumulated records*. Since the execution time of reduce stage is determined by the slowest reduce task, we compare the execution time of the slowest reduce tasks with different garbage collectors and obtain $G1_{121s} < CMS_{131s} < Parallel_{170s}$ as shown in Figure 3a. 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 4a shows that ParallelGC task achieves 5%~12% shorter shuffle spill time than CMS and G1 tasks. This is caused by the three collectors' different heap layouts that will be interpreted in Finding 5. To understand the poor GC performance demonstrated by ParallelGC and CMS compared to G1, we further decompose the GC time into young GC (*YGC*) time, full GC (*FGC*) time, and concurrent GC (*ConGC*) time in Figure 4b. This figure shows that ParallelGC task suffers from 6.6~26.5x longer full GC time than CMS and G1 tasks, while CMS task suffers from 1.3~1.6x longer young GC time than ParallelGC and G1 tasks. The root causes are due to the three collectors' different young/old generation sizing polices and full GC algorithms, which will be interpreted in Finding 6, 7, and 8.

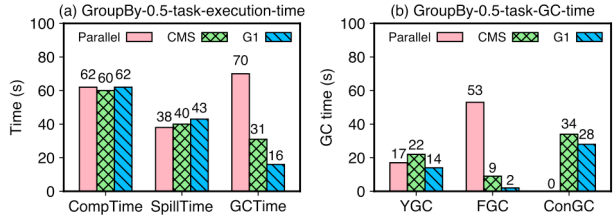

Figure 4: The execution time comparison among GroupBy-0.5 reduce tasks. ParallelGC task are the slowest one due to its longest full GC time. CMS task is slower than G1 task because of its longest young GC time and 7s longer full GC time. ParallelGC does not have concurrent GC

6.3.2 Findings and their implications

Finding 5: Compared to CMS and G1 tasks, ParallelGC tasks achieve 5%~12% shorter shuffle spill time but suffer from 10% more shuffle spills. The root cause is that the three collectors have different heap layouts that lead to different spill thresholds. 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 *groupByKey*() operator to accumulate all the shuffled records in memory. "Accumulate" means that the shuffled <*k*, *v*> records with the same *k* are aggregated into <*k*, *list*(*v*)> records. If the accumulated records exceed the 60% spill threshold, reduce tasks will trigger shuffle spill. Figure 5 illustrates the distribution of the shuffled records in the 32 reduce tasks. It shows that 4 of the 32 ParallelGC tasks trigger shuffle spills and the spill threshold is 3.29G, while only one CMS/G1 task trigger shuffle spill and the spill threshold is 3.69/3.70G. We found 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 6.5GB executor JVM for example, the runtime available heap size comparison is $Parallel_{5.78G} < CMS_{6.44G} < G1_{6.50G}$. Since Spark allocates (*heap size* - 300MB) *

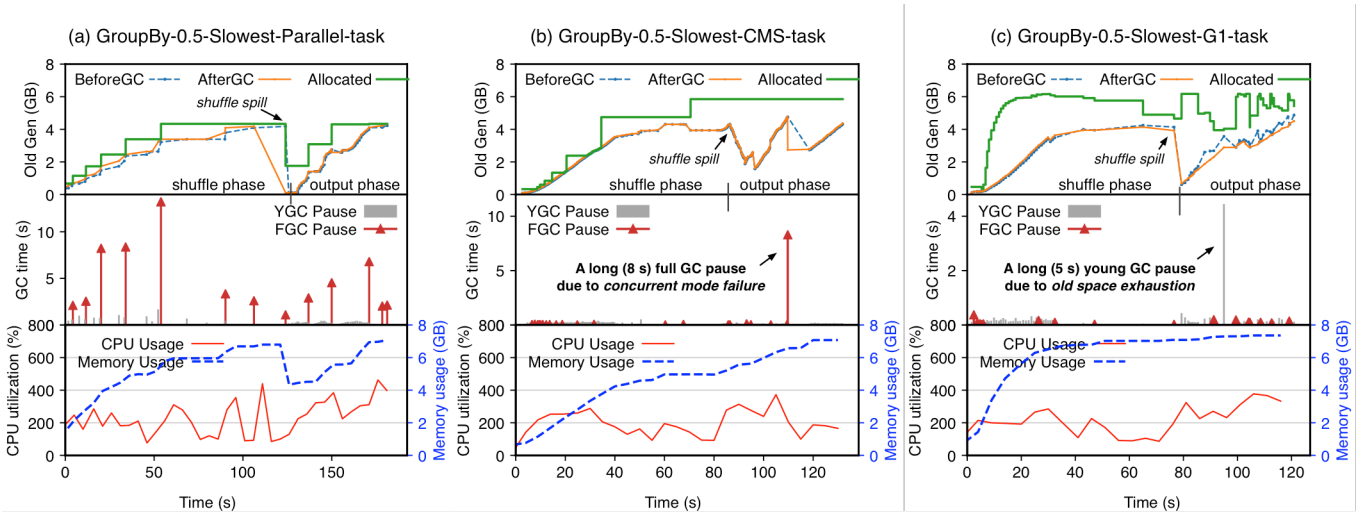

Figure 6: The memory usage and GC pause time comparison among the GroupBy-0.5 slowest tasks with different garbage collectors. *BeforeGC* denotes the size of live objects in old generation before each young or full GC. *AfterGC* denotes the size of live objects in old generation after each young or full GC. *Allocated* denotes the allocated space of old generation.

60% as the spill threshold, the spill threshold comparison is $Parallel_{3.29G} < CMS_{3.69G} < G1_{3.72G}$. The missing heap space (0.72G in Parallel GC) is used as an empty Survivor space for swapping the survival objects during young GC. Since this Survivor space is not used for storing new objects and occupies a contiguous space in Parallel collector, it is not included in the available heap size. CMS collector has the same problem but only 0.06G missing space due to its smaller Survivor space. In contrast, G1 adopts region-based heap layout, whose Survivor space consists of a logical set of regions. These regions can further be used as Eden or Old space for storing new objects, so G1 regards the Survivor space as available heap space. Since Spark does not change the spill threshold at runtime, ParallelGC tasks achieve the shortest spill time with only 3.29GB spilled data. However, ParallelGC tasks may suffer from more shuffle spills and higher disk I/O than CMS and G1 tasks, when the *accumulated records* are multiple times of the spill threshold.

Implication: We need dynamic spill threshold in accordance with the heap size to balance the spill time and spill frequency.

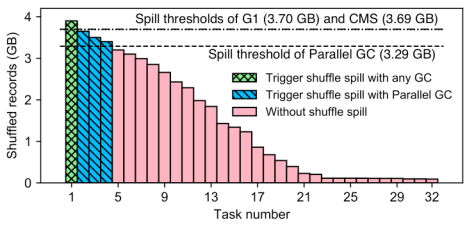

Figure 5: The distribution of shuffled records in 32 GroupBy-0.5 reduce tasks. Due to data skew, 28 tasks did not trigger shuffle spill since their accumulated records did not exceed any spill threshold.

Finding 6: The *long-lived accumulated records* require large old space to accommodate, so different young/old generation sizing polices lead to different young/full GC frequencies. By allocating large old space without shrinkage, CMS tasks achieve ~30% less full GC pauses than ParallelGC and G1 tasks. As described in Finding 5, the *groupByKey*() operator constantly accumulates shuffled records (about 3.9GB) into memory. We regard these records as *long-lived accumulated records*, because their lifetime spans 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. Since these records are *long-lived* objects, they are constantly transferred from young generation to old generation. Full GC will occur when the *long-lived accumulated records* are going to fill up the old generation. 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* [5]). However, we found the three collectors demonstrated different generation sizing patterns that lead to different 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 6a, 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 (~6GB) in Figure 6b/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 (~1.6GB). Small old space leads to frequent full GC pauses as shown in Figure 6a. (2) CMS prefers to expand the old space without shrinkage. Since CMS allocated 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 10 full GC cycles that are ~30% less than ParallelGC task. However, CMS tasks suffers from 4x 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 6c, G1 makes a right decision to allocate 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 is a wrong decision that leads to a long (5 s) young GC pause due to the *to-space exhaustion* (runs out of space of the survivor space or old space). The reason is that the read-back shuffled records still require large old space. Fortunately, G1 enlarges the old space again after this heavy GC.

Implication: Current young/old generation resizing polices lead to frequent GC pauses while accommodating the *long-lived accumulated records*. We need to design more intelligent heap sizing polices in awareness of the memory usage in each data processing phase.

Finding 7: Compared to CMS and G1 collectors, Parallel collector's inappropriate generation resizing timing mechanism leads to 69% more full GC pauses. As described in Finding 6, all the three collectors can resize the old space to accommodate the *long-lived accumulated records* in shuffle phase. However, the three collectors have different generation resizing timing mechanisms (i.e., *when* to resize the old generation) that lead to different GC frequencies. Parallel GC only resizes the old generation at full GC pauses. As a result, 69% of the full GC pauses in Parallel collector are caused by this resizing timing requirement. In contrast, CMS and G1 collectors can resize the old generation during light young GC pauses, which reduces the frequency of full GC pauses.

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

Finding 8: For reclaiming the *long-lived accumulated records*, Parallel collector's stop-the-world marking algorithm is 10x slower than CMS/G1 collectors' concurrent marking algorithms. As shown in Figure 6, ParallelGC tasks have 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 the application thread to *mark* the live objects and *sweep* the unreferenced objects. Since the *long-lived accumulated records* are numerous (~6 millions), this stop-the-world marking is time-consuming that leads to up to 12s full GC pause. In contrast, CMS and G1 collectors use concurrent marking algorithms, which perform most of the object marking work concurrently with the application thread. As a result, their average full GC pauses drop down to ~1s. However, these concurrent algorithms may suffer from long full GC pauses when the object reclamation cannot catch up with the object allocation. Figure 6b shows that the CMS task suffers from a long (6.7s) full GC pause caused by *concurrent mode failure*. The root cause is that the concurrent marking/sweeping phases have not finished reclaiming the unused objects before the old space becomes full (when the spilled records are read back into memory). In this occasion, CMS GC falls back to launch a stop-the-world full GC pause as that of Parallel full GC.

Implication: Concurrent object marking algorithm can reduce the GC pause time while reclaiming the *long-lived accumulated records*. However, they may suffer from unexpected *concurrent mode failure* when the object reclamation cannot catch up with the object allocation.

6.4 The OOM root causes in GroupBy-1.0

The OOM error in GroupBy-1.0 is caused by the accumulated shuffled records, the characteristics of our dataset, and the improper batch size in merge output phase. Firstly, GroupBy accumulated all the shuffled records into memory, which leads to high memory consumption. A part of the records (~3.6GB) are spilled onto disk in shuffle phase. Secondly, these accumulated records demonstrate the feature of *few keys with large values*. The spilled accumulated shuffled records only have 60 distant keys, but each value is about 60MB. Thirdly, in output phase, Spark reads the spilled records into memory to merge with the unspilled records. By default, the spilled records are read into memory with a batch of 1,000 records. Since 60 < 1,000, all the spilled records are read into memory at a time. Thus, the memory usage suddenly increases 3.6GB that leads to the OOM error. We have submitted this issue to Spark community [6]. In our new experiments on larger dataset, we added an aggregation function SUM, so that this OOM error is not triggered.

7. Join on small dataset (16G)

7.1 Application description

Join is a SQL application simplified from the join query in the benchmark [1, 2]. Figure 7 shows the dataflow of Join.

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

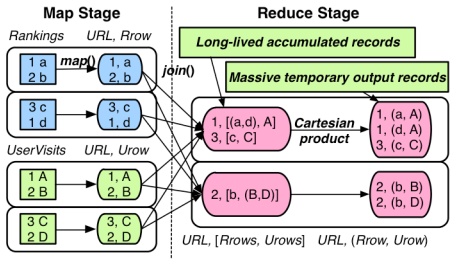

Figure 7: The dataflow of Join.

The map tasks perform *map*() that transforms each row of table *Rankings* to <*URL*, *Rrow*> and each row of table *UserVisits* to <*URL*, *Urow*>. The space complexity of *map*() is *O*(1). In shuffle phase, each reduce task performs a *join*() operator that groups the two tables' rows with the same key as <*URL*, *list*(*Rrows*, *Urows*)>. These grouped shuffle records are also *long-lived accumulated records*, as explained in GroupBy. In output phase, the *join*() operator calculates the Cartesian product of the two sets (*Rrows* and *Urows*), and output <*URL*, (*Rrow*, *Urow*)> records one by one. Since these records are directly outputted onto HDFS after generated, they are *massive temporary output records*. The space complexity of *join*() is *O*(*m*+*n*), where *m* and *n* denotes the length of *Rrows* and *Urows* respectively. This application has heavy shuffle, since the shuffled records is the sum of the number of rows from *Rankings* and *UserVisits* tables.

7.2 Input data

The dataset is generated by HiBench [3].

Application	Input dataset
Join-1.0	16GB Uservisits (90M), 8GB Rankings (120M)
Join-0.5	8GB Uservisits (90M), 4GB Rankings (60M)

7.3 Experimental results

Table 5: The applications' execution time on different GCs.

Application	The application execution time		
	Parallel	CMS	G1
Join-0.5	$4.8_{(1.3)}$	$3.7_{(0.4)}$	$4.4_{(0.2)}$
Join-1.0	$84.2_{(76)}$	$10.9_{(0.8)}$	$11.6_{(0.5)}$

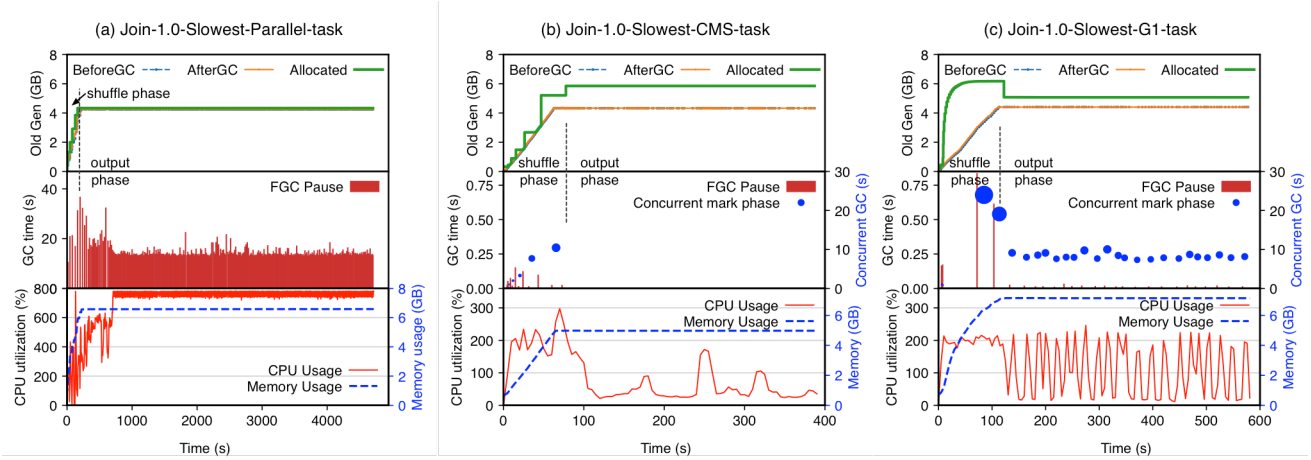

Figure 9: The memory usage and GC pause time comparison among the Join-1.0 slowest tasks. *FGC Pause* only illustrates the time of stop-the-world phases in each full GC cycles, including *initial-mark*, *remark*, and *cleanup* phases. The *concurrent-mark* phase is illustrated by the blue circle, and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

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

7.3.1 Performance comparison results

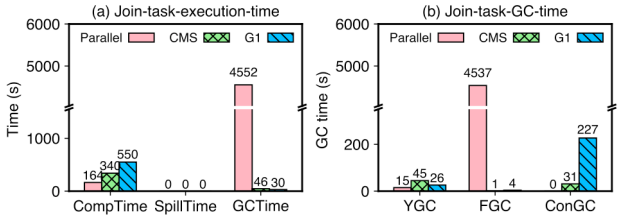

Figure 8: The execution time comparison among Join-1.0 reduce tasks is CMS_{386s} < $G1_{580s}$ < $Parallel_{4568s}$. ParallelGC task is 7.8~11.8x slower than CMS and G1 tasks, due to its extremely long full GC time. The data computation of CMS and G1 tasks are 2~3.4x slower than ParallelGC task. G1 task is 1.5x slower than CMS tasks due to its longer data computation time.

Join-1.0 application has two map stages (128/64 map tasks) and a reduce stage (32 reduce tasks). The performance differences only happened in reduce stage, where the memory usage consists of *long-lived accumulated records* and *massive temporary output records*. The *massive temporary output records* refer to the records generated by the Cartesian product operation in *join*(). Figure 8 compares the execution time and GC time among the slowest reduce tasks. The three slowest tasks have the same shuffled data, but did not trigger shuffle spill. From Figure 8a and 8b, we obtain two observations. (1) ParallelGC tasks suffer from 1000x full GC time than CMS and G1 tasks. This is mainly caused by the different full GC triggering conditions that will be interpreted in Finding 9. (2) CMS and G1 tasks suffer from 2~3.4x longer data computation time than ParallelGC task. This is caused by the CMS and G1 collectors' CPU-intensive object marking algorithms that will be interpreted in Finding 10.

7.3.2 Findings and their implications

Finding 9: Current threshold-based full GC triggering conditions lead to frequent, but unnecessary full GC pauses towards the *long-lived accumulated records*. With different full GC triggering thresholds, ParallelGC task triggers 11x more full GC pauses than G1 task, and G1 task triggers 4x more full GC pauses than CMS task. 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 ~300 full GC pauses that lead to ~1hr GC time. In contrast, G1 task triggers only ~30 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 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* 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 output phase ends. In contrast, CMS and G1 use *aggressive* triggering conditions that start the GC cycle before the old space is exhausted. For G1 GC, it starts a concurrent collection cycle when the heap usage reaches 45% of the heap space. For CMS GC, it starts a concurrent collection cycle at a higher threshold (default 92% of the old space) and based on 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.

Finding 10: Existing concurrent object marking algorithms used in CMS and G1 collectors are inefficient for handling *long-lived accumulated records* due to CPU resource contentions with CPU-intensive data operators like *join*(). As shown in Figure 9, CMS and G1 tasks have ~90% shorter GC time but 2~3x longer data computation time than ParallelGC task. The root causes include (1) *The concurrent marking algorithms in CMS and G1 collectors have CPU contention with the data processing thread.* The 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 algorithm that performs object marking in parallel with the data processing thread. As a result, concurrent object marking incurs CPU contention with the data processing thread. (2) *While reclaiming long-lived accumulated records, the concurrent marking algorithms are CPU-intensive that degrades 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 (~10 million) and living in both shuffle and output phase. In output phase, the *join*() operator needs to compute the Cartesian product of the rows with the same key from two tables. This data computation is CPU-intensive, since Cartesian product has $O(n^2)$ time complexity and processes large number of (~19 millions) temporary output records. Due to CPU contention, the concurrent marking algorithm slows down this CPU-intensive data computation of CMS and G1 tasks. Moreover, as interpreted in Finding 9 and shown in blue circles in Figure 9c, G1 task suffer from more full GC cycles (i.e., concurrent mark steps) than CMS task in output phase. As a result, the CPU usage of G1 task is much higher than CMS task in output phase as shown in Figure 11. Thus, G1 task has 1.6x longer data computation time than CMS task. Given that many Spark applications are CPU-intensive [7], such CPU contention between GC activities and Spark applications will persist.

Implication: Today's concurrent marking algorithm reduces 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 marking algorithm that balances the trade-offs between the GC pause time and CPU usage of the object marking.

References

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

[2] 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.

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

[4] 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

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

[6] [SPARK-22286] OutOfMemoryError caused by memory leak and large serializer batch size in ExternalAppendOnlyMap. https://issues.apache.org/jira/browse/SPARK-22286.

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