October 16, 2015

RT0968
Computer Science 12 pages

Research Report

Workload Characterization and Optimization of TPC-H Queries on Apache Spark

Tatsuhiro Chiba and Tamiya Onodera

IBM Research - Tokyo
IBM Japan, Ltd.
19-21, Hakozaki, Chuo-ku
Tokyo 103-8510, Japan

Workload Characterization and Optimization of TPC-H Queries on Apache Spark

Tatsuhiro Chiba
IBM Research - Tokyo
19-21, Hakozaki, Chuo-ku
Tokyo, 103-8510, Japan
chiba@jp.ibm.com

Tamiya Onodera
IBM Research - Tokyo
19-21, Hakozaki, Chuo-ku
Tokyo, 103-8510, Japan
tonodera@jp.ibm.com

Abstract—**Besides being an in-memory oriented computing framework, Spark runs on top of a Java Virtual Machine (JVM), so JVM parameters must be tuned to improve Spark application performance. Misconfigured parameters and settings degrade performance. For example, using Java heaps that are too large often causes long garbage collection pause time, which accounts for over 10-20% of application execution time. Moreover, recent modern computing nodes have many cores and support running multiple threads simultaneously with SMT technology. Thus, optimization in full stack is also important. Not only JVM parameters but also OS parameters, Spark configuration, and application code itself based on CPU characteristics need to be optimized to take full advantage of underlying computing resource. In this paper, we use TPC-H benchmark as our optimization case study and gather many perspective logs such as application log, JVM log such as GC and JIT, system utilization, and hardware events from PMU. We investigate the existing problems and then introduce several optimization approaches for accelerating Spark performance. As a result, our optimization achieves 30 - 40% speed up on average, and up to 5x faster than the naive configuration.**

I. Introduction

As data volumes increase, distributed data parallel processing on large clusters is useful to accelerate computing speed for data analytics. Hadoop MapReduce is one of the most popular and widely used distributed data processing frameworks with scale out and fault tolerance. While this simple programming model enables us to implement distributed data-intensive applications more easily, nowadays, various types of analytics applications such as relational data processing, machine learning, and graph algorithms have been applied to Hadoop and its related ecosystem. However, they do not always work efficiently on the existing Hadoop system because the current framework is not suitable for low latency or iterative and interactive analytics applications. As a result, multiple alternatives to Hadoop systems [1], [2], [3] have been developed to overcome the deficient area. Apache Spark [4], [5] is an in-memory oriented data processing framework that supports various Hadoop compatible data sources. Spark keeps as much data in a reusable format in memory as possible, so that Spark can reduce disk I/O drastically more than Hadoop. Spark provides many operators and APIs for parallel data collections in Scala, Java, Python, and R, and also useful libraries and components for machine learning (MLLib), relational query (SparkSQL and DataFrame), and graph processing (GraphX). Moreover, Spark retains scalability and resiliency as well, so it has attracted much attention recently.

Although Spark has been developed by the open source community and new features and various Spark runtime optimization techniques are applied frequently, characterizing habits of Spark and gaining deep insight into the tuning of Spark applications from the system-side and application-side aspects are important for not only Spark users but also Spark runtime developers and system researchers who try to apply their own optimization algorithms to Spark itself and other similar frameworks. However, Spark's core concept and design are different from those of Hadoop, and less is known about Spark's optimal performance, so how Spark applications perform on recent hardware has not been investigated thoroughly. Furthermore, various components (OS, JVM, runtime, etc.) try to achieve higher performance in accordance with their own optimization policies, so stable performance is difficult to achieve unless these components cooperate.

To do achieve this cooperation, there are several challenges. First, Spark application performance bottlenecks are more complex to find than those of Hadoop. Hadoop MapReduce is a disk I/O intensive framework, and I/O throughput performance influences its data processing performance directly. In contrast, maximization of I/O throughput is still important for Spark, but its performance bottlenecks are moved to CPU, memory, and the network communication layer because of Spark's in-memory data processing policies. Second, Spark runs on top of Java Virtual Machine (JVM) with many task executor threads and a large Java heap, so JVM tuning is important for improving performance. Spark creates many immutable and short-lived objects in heaps, so GC pause time, which is often a dominant part of application execution time with a large heap, may be insufferably long if we do not provide suitable GC algorithms or JVM parameters for Spark.

To address these challenges, in this paper, we investigate the characteristics of Spark performance with multiple metrics through running Spark applications. We use TPC-H queries on Spark as reference applications and capture JVM logs such as GC and JIT, hardware counter events, and system resource monitor logs. From these logs, we define several problems and optimization approaches for them. Finally, we evaluate how these optimizations help to improve TPC-H query performance.

We make the following contributions in this paper: (1) We characterize TPC-H queries on Spark and analyze many performance metrics to help our comprehension. (2) We provide several Spark optimization methodologies from JVM side to reduce GC overhead without increasing heap memory and also to increase IPC by utilizing NUMA and SMT. (3) We also provide potential problems we found through our experiments and it would be useful for design or developing JVM and Spark core runtime.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 summarizes TPC-H workload and breakdown measurement result from many metrics. Section 4 considers what is the existing problem to accelerate Spark performance and how we can optimize each layer, Then, Section 5 describes the results with tuning. Section 6 mentions related works. Finally, Section 7 concludes the summary of this paper.

II. BACKGROUND

A. Apache Spark Overview

Apache Spark is an open source in-memory oriented cluster computing framework with APIs in Scala, Java, Python, and R. Spark can cache working set data or intermediate data in memory to reduce data loading latency as much as possible, so that Spark performs much better than Hadoop in iterative types of workloads such as machine learning algorithms and interactive data mining. Not only iterative workloads but also other general workloads such as batch jobs, ETL, and relational queries are also run on Spark by developing various libraries for graph processing, stream computing, and query processing, so Spark has recently been used as a general purpose distributed computing engine and also outperforms Hadoop [6].

Spark provides a functional programming API for the abstraction of immutable distributed collections called Resilient Distributed Datasets (RDDs) [4]. Each RDD contains a collection of Java objects that is partitioned over multiple nodes. RDDs are transformed into other RDDs by applying operations (e.g. map, filter, reducebykey, count, etc.), and this RDD transformation flow is represented as task DAGs, so the transformation tasks for partitioned data are launched on each node. RDDs are evaluated lazily, so computation tasks are not launched before applying certain types of operations like count. To process the divided tasks, Spark manages many task executor threads within a JVM.

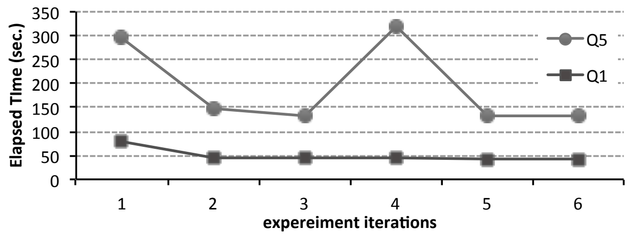

Fig. 1. The transition of Q1 and Q5 response time through six time iterations

B. Spark Benchmarks and Applications

1) Spark workload benchmarks: Some benchmark frameworks [7][8][9] for Spark have recently been developed that are mainly used for evaluating performance and searching for suitable workload configurations. They provide a comprehensive set of Spark workloads for machine learning, graph processing, and streaming and also synthetic data generators for each workload. In this paper, we select three machine learning algorithms (Kmeans, Logistic Regression, and Support Vector Machine) and two graph processing algorithms (PageRank, SVD++) as iterative types of workloads.

2) TPC-H benchmark: TPC-H[1] is a decision support benchmark, consisting of a suite of business oriented ad-hoc queries and concurrent data modifications, defined as 22 queries for eight different sized tables. TPC-H was originally used for evaluating MPP DBMS systems but has recently been used for Hadoop based query engine systems [10][11]. Spark SQL [12] is a major component in Spark, and it bridges relational processing and procedural processing with Spark APIs. With compatibility for Hive, Spark SQL can directly execute Hive Query on Spark runtime and load data from existing Hive tables. TPC-H queries include the core part of SQL (selection, aggregation, and join), and join operation is the most costly because data shuffling occurs. Consequently, TPC-H is also useful to evaluate Spark runtime performance as well. Moreover, TPC-H is not an iterative workload unlike the above machine learning workloads, so it is compared with machine learning workloads.

C. Performance Measurement Tools

Spark provides statistical reports about multiple metrics of executed jobs. For example, the report shows how many tasks and stages are executed, which part of a stage is the most expensive, and which phases such as data loading, computing, and data shuffling are costly. These metrics give us helpful information about the overall performance and hints for performance improvement, but it is difficult to tune application performance by using the given metrics. Perf [13] and Oprofile [14] are commonly used system profiling tools on Linux.

[1]http://www.tpc.org/tpch/

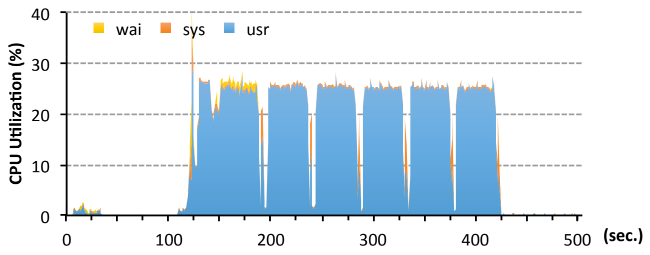

Fig. 2. CPU utilization while running Q1 with default

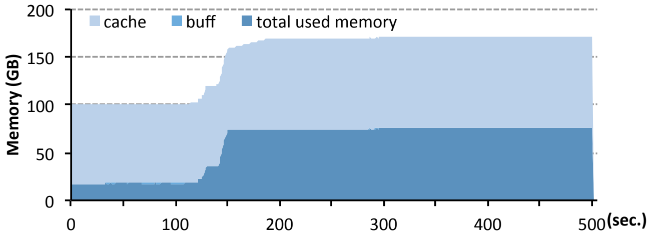

Fig. 3. Memory utilization while running Q1 with default

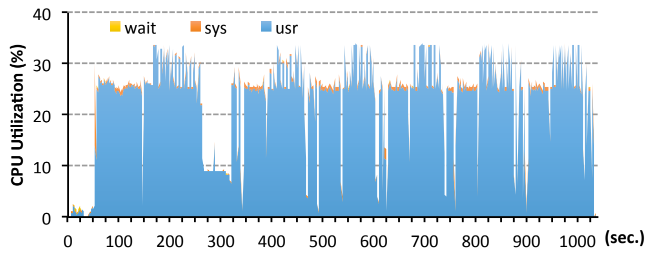

Fig. 4. CPU utilization while running Q5 with default

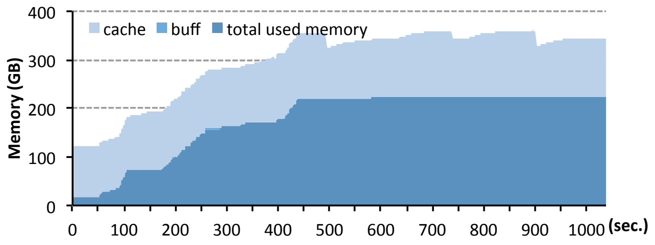

Fig. 5. Memory utilization while running Q5 with default

III. Spark Workload Analysis

A. *Experimental Platform and Setup TPC-H Workloads*

First, we introduce our experiment environment. We compared all experiments on a single node of POWER System S824L, which is equipped with two 3.3GHz POWER8 processors. Both POWER8 processors have 12 cores each, and each core has a private 64KB L1 cache, private 512KB L2 cache, and a shared 96MB L3 cache. This system has 1TB RAM and 1TB RAID5 disk. The software stack of this system is Ubuntu 14.10 (kernel: 3.16.0-31-generic), Hadoop 2.6.0, and Spark 1.5.0, and we used IBM J9 JVM(1.8.0 SR1 FP10).

Next, we introduce a TPC-H dataset and queries. We generated TPC-H table data files using an official data generator with a 100GB scale factor and then loaded them into Hive tables on HDFS. The chunk size of HDFS is 128MB. Original data sizes of all tables are as follows: lineitem is 75GB, orders 17GB, partsupp 12GB, customer 2.3GB, part 2.3GB, supplier 137MB, nation 2.2KB, and region 389B. All tables are stored with Snappy compressed and Parquet columnar formats.

We used TPC-H Hive queries published at github[2] as a basis. However, several queries did not finish or took too long because they included wasteful temporary table creation, so Spark could not generate a Spark-friendly query execution plan. Therefore, we modified them by eliminating several temporary tables and changing query order in HiveQL directly. As a result, the queries can finish in an acceptable response time. We also evaluated each query six times and then chose the best one.

[2]https://github.com/rxin/TPC-H-Hive

B. *TPC-H query results with default settings*

Table I shows the query response time on a single Spark Executor JVM with 48 worker threads and 192GB heap. It also lists each query characteristic, generated spark operators, total input data size loaded from HDFS, total shuffled data size between stages, and the total number of stages and tasks. We summarized only the key operations of each query since they affect Spark query execution plan. For example, Q5 performs a groupby and fix inner joins using six tables. These operations are converted into RDD based execution plan, which has three data loading stages, three hash based shuffle join stages, one broadcast hash join stage, and one aggregation stage, through the query optimizer in SparkSQL. The total stage equals the sum of Spark operations listed in Converted Spark Operation in Table I. Q5 loads 8.8GB table data in total in three data loading stages for later shuffle hash join. Then it transfers 14.1 GB shuffle data in total. The amount of shuffle equals the total read-and-write shuffle size in a query. As a result, Q5 takes 137 seconds until eight stages with 1547 tasks are completely calculated by 48 worker threads.

From these results, we can find some trends and characteristics in queries. First, the queries such as Q1, Q6, and Q19, which have little shuffling data, can finish early even if the input size is large and there are multiple shuffle join stages. These queries performances depend more than others on the data loading from the disk. Next, the queries such as Q5, Q8 and Q9, which have huge shuffling data, take over 100 sec.

	Key Characteristics of HiveQL	Converted Spark Operation	input (GB)	shuffle (GB)	Stages/ Tasks	time (sec)
Q1	1groupby, 1table	1load, 1Aggregate	4.8	0.002	2 / 793	48.7
Q2	1groupby, 4join, 5table	2load, 2HashJoin, 1BcastJoin	0.92	2.3	5 / 512	23.7
Q3	1groupby, 2join, 3table	3load, 2HashJoin, 1Aggregate	7.3	5.0	6/1345	64.6
Q4	1groupby, 1join, 2table	2load, 1HashJoin, 1Aggregate	4.2	1.0	4/1126	56.2
Q5	1groupby, 5join, 6table	3load, 3HashJoin, 1BcastJoin, 1Aggregate	8.8	14.1	8/1547	125
Q6	1select, 1where, 1table	1load, 1Aggregate	4.8	0	2/594	15.1
Q7	1groupby, 1unionall, 6join, 5table	4load, 1Unionall, 4HashJoin, 1Aggregate	9.3	16.5	10/1755	132
Q8	1groupby, 7join, 7table	4load, 4HashJoin, 1BcastJoin, 1Aggregate	11.7	14	10/1766	159
Q9	1groupby, 5join, 6table	4load, 4HashJoin, 1BcastJoin, 1Aggregate	11.8	34.4	10/1838	370
Q10	1groupby, 3join, 4table	3load, 2HashJoin, 1Aggregate	7.7	3.8	6/1345	49.1
Q11	1groupby, 2join, 3table, 1write	1load, 1HashJoin, 1BcastJoin, 1Aggregate	0.89	1.7	4/493	23.0
Q12	1groupby, 1join, 2table	2load, 1HashJoin, 1Aggregate	5.0	1.5	4/1126	44.5
Q13	1groupby, 1outer join, 2table	2load, 1HashOuterJoin, 1Aggregate	3.9	1.8	4/552	100
Q14	1join, 2table	2load, 1HashJoin, 1Aggregate	6.6	0.3	4/813	20.9
Q15	1groupby, 1table, 1write	1load, 1Aggregate, 1write	6.6	0.4	2/793	29.4
Q16	1groupby, 2join, 3table	2load, 1HashJoin, 1BcastJoin, 1Aggregate	0.65	0.8	4/510	132
Q17	1join, 1unionall, 2table	4load, 1HashJoin, 1BcastJoin, 1Union, 1Aggregate	16.7	7.1	8/3966	297
Q18	3join, 1unionall, 3table	6load, 3HashJoin, 1Union, 1Limit	7.7	13.8	11/3725	202
Q19	3join, 1unionall, 2table	6load, 1Union+3HashJoin, 1Aggregate	19.8	0.4	8/2437	80.8
Q20	1groupby, 4join, 5table	3load, 3HashJoin, 1BcastJoin	6.7	2.2	7/1305	88.7
Q21	1groupby, 4join, 1outer join, 4table	4load, 2HashJoin, 1BcastJoin, 1OuterJoin, 1Aggregate	15.5	13.9	9/2714	-
Q22	1groupby, 1outer join, 3table	3load, 1OuterJoin+CartesianProduct, 1Aggregate	0.6	1	5/571	27.6

TABLE I
CHARACTERIZATION OF ALL OF TPC-H QUERIES IN TERMS OF SQL AND SPARK METRICS

Also, these have more shuffle data than input data, so these queries performances depend on the data shuffling between stages. Q21 takes over 10,000 sec because of many execution failures with default configuration, so we do not describe the time in the Table I. In consequence, we categorized the queries into two classes: shuffle light and shuffle heavy. In the next subsection, we describe several metrics in more detail: system utilization, Spark log, JVM, and system side.

C. System Utilization and Spark Metrics

We picked up Q1 and Q5 as representatives of the shuffle light and shuffle heavy categories, respectively. First, we evaluated actual query response time in each iteration. Figure 2 shows the results of Q1 and Q5 through six iterations. The first iteration is about 1.5x - 2x slower than other iterations in both queries, because Just-In-Time (JIT) compilation for hot methods is not finished. To be more precise, the first round of each stage takes too long. For example, the Q1 execution plan consists of two stages, and then the data loading tasks in the first and second stages are divided into 593 and 200 tasks, respectively. Executor JVM has 48 worker threads, so these threads process assigned tasks individually and simultaneously. If these threads are evenly assigned tasks, each worker thread will process 12 - 13 tasks in total, which means the first stage consists of 13 rounds. The threads in the second round of a stage can use optimized JITed code, so processing time is 1.5 - 2x faster than in the first round. A similar drawback exists in all eight stages in Q5. Moreover, we often observed a failure in both queries such as the fourth iteration of Q5. The failure occurs when using a single JVM with many worker threads and it mentions the failure causes by calling native snappy compression library. To trace the cause of it beyonds the scope of our paper, but using one large JVM provides us unstable.

Then we checked system utilization including CPU, memory, I/O context switches, etc. Figures 2 and 3 show CPU utilization and memory usage for Q1, while Figures 4 and 5 show Q5 while each query is run six times continuously on the same Executor JVM. The six hills correspond to the iterations. In both queries, we can see that 25% of CPU resources are used for user time and all worker threads can utilize available slot without I/O wait and system time. This because we only assigned 48 worker threads, which are one-fourth the threads of 192 logical CPU, for Executor JVM. In the Q5 CPU usage graph, we can observe some spikes in the later part of iteration. This burst is caused by heavy GC activity in shuffle phase. From the perspective of memory usage, used memory and page cache grow after iterations start. The total used memory size in Q1 does not exceed around 70 GB, while that in Q5 reaches 220 GB. From this graph, Q5 run out of all available JVM heap.

D. GC, JIT and Hot Method Profiling

Next, we evaluated how many heaps Executor JVM used and how often GC is called. Figures 6 and 8 show heap usage and pause time transition while executing Q1 and Q5 six times, so we can see six peaks, and each peak corresponds to the execution order. The 192 GB heap is divided into two generational spaces: nursery and tenure. The 48 GB heap is used as nursery since one-third of the total heap is assigned to it as default. In the upper graph, the blue, yellow, and red lines represent total used heap, total used tenure heap, and

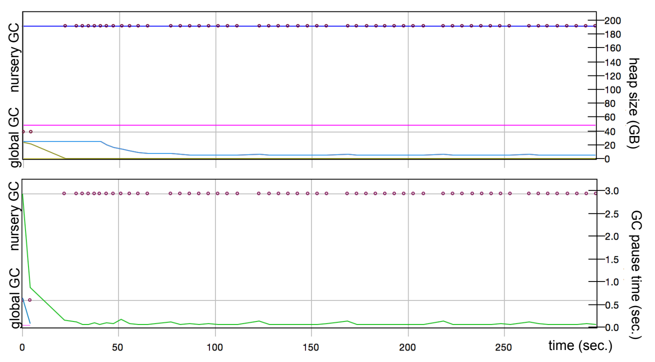

Fig. 6. Q1 with default configuration: GC heap usage (upper), GC pause time (lower)

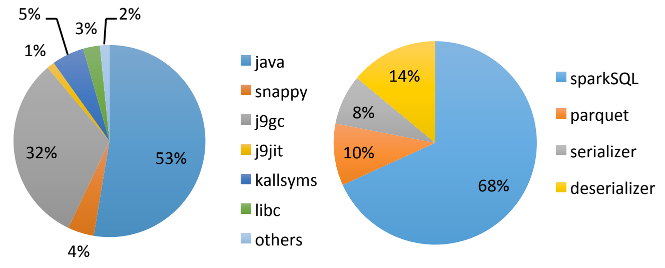

Fig. 7. Oprofile method profiling result of Q5 with default configuration

total used nursery heap after GC. The pink line represents the capacity of nursery, and circles mean the time when GC occurs and also what GC types are chosen: nursery or global GC. In the lower graph, the green line represents actual GC pause time when GC is called, and the light blue line is time for global GC. In Q1, objects in nursery space are almost all cleaned up after each nursery GC and also do not flow into tenure space. As a result, pause time is quite small and accounts for only 2% of the whole execution time. In Q5, on the other hand, GC performance is completely different from that in Q1. Due to running out of nursery space, objects flow in tenure space gradually and then are collected when tenure space becomes full. Moreover, pause time in nursery GC is bigger than in global GC.

To estimate actual GC cost in Q5 execution, we profiled Q5 with oprofile. By using oprofile, we can also determine how much time is spent in GC and also what kind of hot methods there are as well. Figure 7 shows the ratio of hot methods to total execution time for Q5. We divide parts into five categories to understand what part should be improved. As seen in Figure 7, GC cost is obviously higher than other parts, which accounts for over 30% in whole execution time. In terms of application related breakdown, over 20% of time is spent for object serialization and deserialization, and 10% is for parquet columnar data loading.

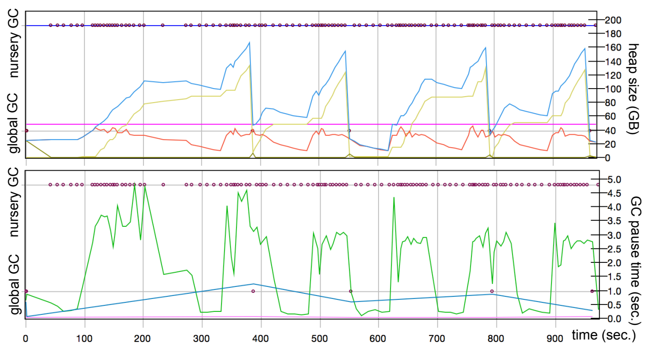

Fig. 8. Q5 with default configuration: GC heap usage (upper), GC pause time (lower)

Finally, Figure 9 shows how many methods are compiled in each optimization level. Almost all methods are compiled in the warm optimization level. On the other hand, a few methods reach scorching, which is the highest optimization level. While the first query execution is running, an application cannot use higher optimization level code, so the performance is slower than later iterations as explained in the previous subsection. We also confirmed that the compilation levels of the top ten methods reach warm, hot, and scorching.

E. *PMU Profiling*

We captured performance counter events from a performance monitoring unit (PMU) for all queries. Table II lists *"perf stat"* command results. As you can see, both queries account for 50 - 60% of wasteful CPU stall cycles on the backend pipeline. Due to this big backend stall, the rate of instructions per cycle (IPC) remains low. To know from where this backend stall comes, we calculated other hardware counter events. The CPI breakdown model is provided elsewhere [15], and we used it as a reference. Following this model, we found that 60% of PM_RUN_CYC spends on PM_CMPLU_STALL which stalls mostly in Load Store Unit (LSU). With a further breakdown, 40% of LSU completion stall comes from DCACHE_MISS that comes from L3MISS. Then, L3MISS is caused by mainly DMISS_DISTANT, which means distant memory access. Moreover, the CPU migrations occur frequently, especially in Q5. Consequently, distant memory access must be reduced while running and its migration rate kept as low as possible.

IV. Problem Assessment and Optimization Strategy

In the previous section, we describe several performance metrics. In this section, we enumerate the existing problems in each layer on the basis of preliminary experiments and then decide optimization and performance improvement approaches.

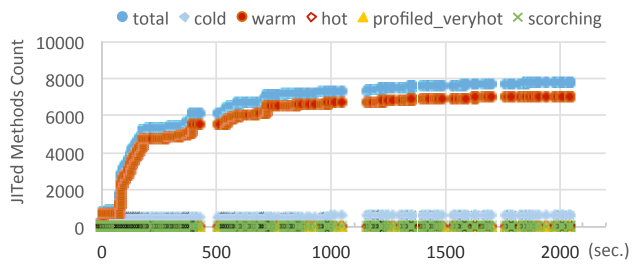

Fig. 9. JITed methods counts of Q5 in each compilation level

counters	Q1	Q5
CPU cycles	6.8×10^{12}	2.2×10^{13}
stalled-cycles-frontend	2.1×10^{11} (3.20%)	6.2×10^{11} (2.76%)
stalled-cycles-frontend	3.3×10^{12} (49.0%)	1.3×10^{13} (59.1%)
instructions	7.0×10^{12}	1.5×10^{13}
IPC	1.03	0.67
context-switches	407K	440K
cpu-migrations	11K	26K
page-faults	308K	1045K

TABLE II
PMU PROFILING RESULT FOR Q1 AND Q5

A. How We Can Reduce GC Overhead

First of all, GC performance has a negative influence as shown in Figure 8. Choosing optimal GC policy and suitable heap size for application is the most important factor to reduce GC overhead. In terms of GC strategies, generational GC is generally used in J9 JVM though we can use other GC strategies. Generational GC is also the default GC policy in OpenJDK, because it is suitable for almost all kinds of applications, so we use generational GC in this paper. In terms of heap sizing, the large heap causes a long pause time when global GC happens, and the small nursery space causing heavy coping GC occurs as we observed in Q5. Consequently, we should search for the optimal combination of total heap and the ratio of nursery space.

Although we used only a single Executor JVM, there is no limit to using multiple Executor JVMs on the same machine. That is another approach to change heap size. Keeping each JVM heap small helps to reduce GC cost directly, so we should evaluate the effect of changing JVM counts. Moreover, we can apply many JVM options that are also helpful to improve performance. For example, default JVM manages up to 64 GC threads because there are 192 cores available virtually from OS. However, using too many GC threads degrades performance due to contention, many context switches, etc., so we may improve application performance by reducing GC threads. These options are for not only GC performance but also application performance.

Hence, we try to find optimal settings for JVM performance, especially for GC, to accelerate Spark application in the following three evaluation metrics: (1) to change nursery space size, (2) to sweep JVM options, and (3) to change Executor JVM counts while maintaining the total heap and working thread count.

B. How We Can Accelerate IPC and Reduce Stall Cycles

As shown in Table II, IPC for TPC-H query on Spark was not very high. One approach to accelerate IPC is to utilize more Simultaneous Multi Threading (SMT) features [16]. POWER8 supports up to SMT8 mode, and our POWER8 has 24 cores, so there are 24, 48, 96 and 192 available hardware threads. In our preliminary experiment, we only used 48 worker threads in SMT8. Of course, we can assign 192 worker threads in Executor JVM but cannot expect big improvements by increasing worker threads up to 192 due to resource contention between threads and other processes. It would be excessive to have 192 worker threads, but the performance may be improved more by increasing worker threads to 96 in total. In this case, we expect that four worker threads are running on the same physical core ideally.

The other problem that retards the increase of IPC is huge stall in the backend pipeline. From our investigation into PMU counter, CPU migration and distant memory access frequently occurred while queries were running on Spark. To reduce such wasteful stall, setting NUMA aware affinity for tasks is one well known approach. Our machine has two POWER8 processors that are connected via NUMA. More precisely, each POWER8 processor has two NUMA nodes, so we can use four NUMA nodes. Therefore, we may improve application performance by pinning NUMA aware affinity for Spark Executor JVMs.

Accordingly, we evaluate the following approaches to increase IPC for TPC-H on Spark: (1) changing SMT mode and the number of worker threads, and (2) applying NUMA affinity for Executor JVM.

V. PERFORMANCE EVALUATIONS

In this section, we apply several of the optimization ideas described above and then evaluate how application performance is improved in terms of execution time, garbage collection performance, CPU cycles, etc.

A. Method Level Workload Characterization on Spark

Before starting evaluations for each tuning, we characterized more details about each TPC-H workloads from the perspective of method sampling. Figure 10 shows the percentage of sampling methods in all queries and then summarized them into several categories. Java represents application code and Spark runtime code, snappy does the native library call for compression, j9gc, j9jit, and j9vm include JVM level methods related to GC, JIT, and others respectively, and kallsyms is derived from Kernel which attaches debug symbols to kernel code. Figure 11 describes the drill down of java and the ratio is normalized. SparkSQL represents actual computation, others are related to I/O such

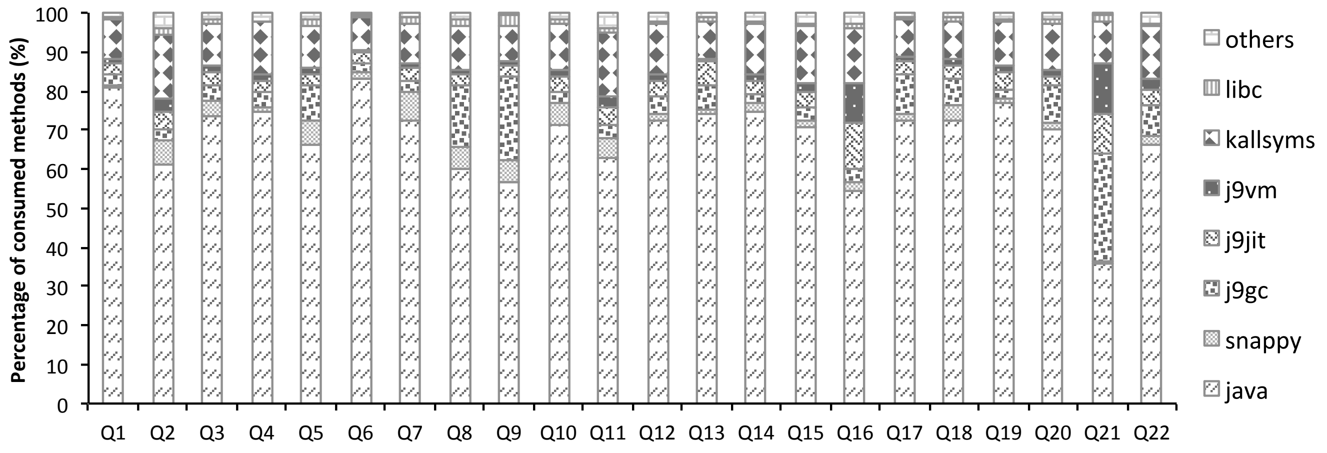

Fig. 10. The ratio of categorized eight components of Oprofile sampling result for all of queries

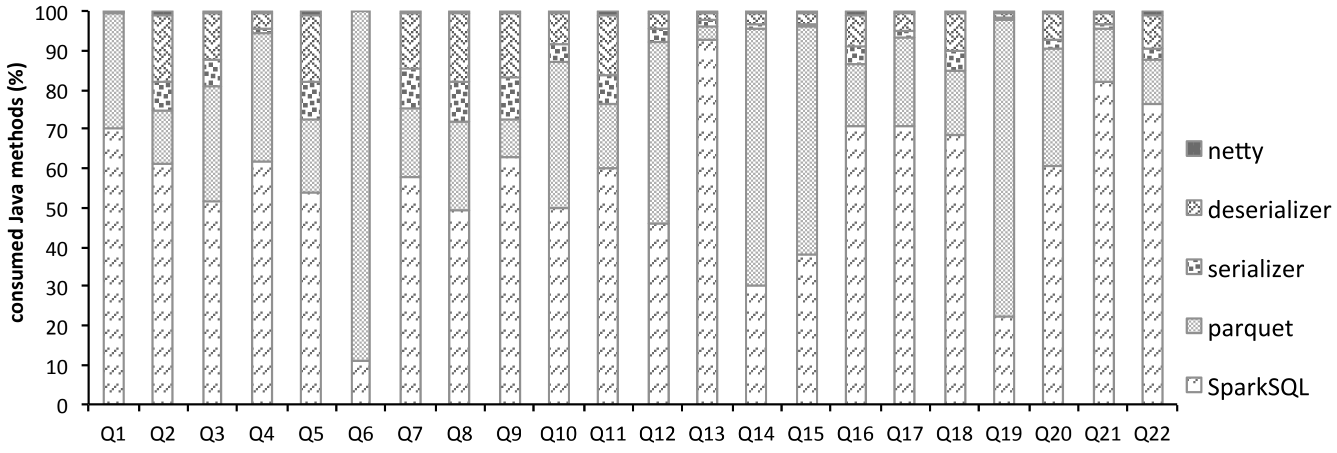

Fig. 11. The ratio of details about Java related processes

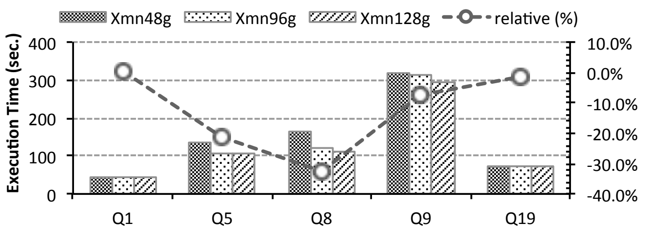

Fig. 12. Performance Comparison while changing nursery heap size

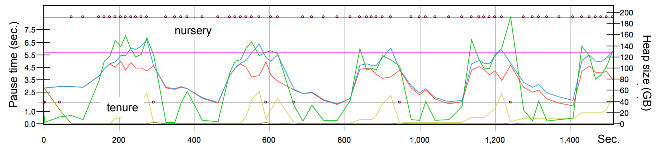

Fig. 13. heap statistics and GC pause time while running Q9 on a single JVM with 128GB nursery heap and 48 worker threads

as loading data (parquet), and data shuffling (serialization and netty). These results collected on the four JVMs instead of one JVM, because one JVM is a little fragile for query execution. As mentioned at Section III, shuffle data size oriented categorization is reasonable for workload characterization. The queries that has large shuffle data pay relatively higher GC, snappy and (de)serialization than others. On the other hand, both of computation and data loading ratio in shuffle less queries's is relatively higher as well. In our experiment, we pick up typical queries such as Q1, Q3, Q5, Q6, Q8, Q9, Q13, Q16, and Q19 for evaluation.

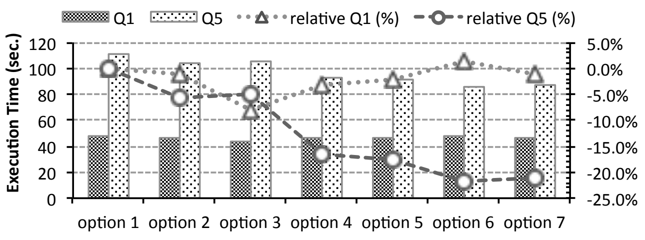

Fig. 14. Comparison of JVM Options

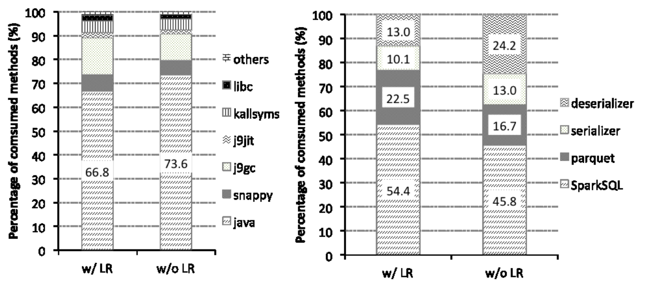

Fig. 15. Profiling method stack with and without lock reservation option. Categorized all of related processes (left) and drill-down Java related tasks (right)

B. Heap Sizing

First, we changed nursery heap size from 48GB to 96GB or 128GB, which are halves or three-fourths of total heap. J9 JVM reserves one-fourth of total heap as nursery space, so 48GB is a default value in our setting. Figure 12 shows query execution time and its relative improvement percentage on the second y-axis compared with using 48GB nursery heap. Shuffle data light queries such as Q1 and Q19 are not improved because heap usage is basically small in these queries, so GC does not happen frequently even if the nursery size is increased.

On the other hand, the shuffle data heavy queries improved by 30 - 40%. By increasing nursery heap, we can prevent objects in nursery heap from flowing into tenure heap. Therefore, the frequency of copying GC in nursery heap stays low. Moreover, there is no global GC while query execution is run six times because almost all objects are collected within nursery heap. However, the performance in Q19 is improved by only 10%. Although the frequency of copying GC within nursery heap is reduced, 128GB nursery heap is not enough for keeping all objects for Q19. As a result, many objects still flow into tenure heap, so global GC occurs periodically as shown in Figure 13.

C. JVM Options Sweep

Next, we tested several JVM options listed in Table III. There are many selectable JVM options, but we chose the following options that are expected to improve GC and application code execution. The experiments were run six times per configuration on a single Executor JVM with 48 worker threads. To evaluate which JVM option supports improvement, we used option 1 as a baseline and then appended one JVM option in each experiment. Figure 14 shows the query execution time as well as relative improvement percentage in each option compared with option 1. As shown in Figure 14, all JVM options help to improve query execution performance. The improvements were especially drastic when lock reservation with -"XlockReservation" (option 4) and disabling runtime instrumentation with "-XX:-RuntimeInstrumentation" (option 6) were enabled.

Lock reservation [17] enables a lock algorithm to be optimized when a thread acquires and releases a lock frequently. In lock reservation, when a lock is acquired by a thread for the first time, it is reserved for the thread. In the reserved mode, the thread acquires and releases the lock without any atomic instruction. If a second thread attempts to acquire the lock, the system cancels the reservation, falling back to the bimodal locking algorithm [18]. In Spark, since many worker threads are running and all threads process their respective RDD partitions, performance can be improved if a synchronized method is heavily called. However, there are no decisive points in Spark runtime code itself, so that we checked method call stacks via oprofile.

Figure 15 shows the stacked method profiling results with and without the lock reservation option. Each stack corresponds to major components in Spark execution. The left one describes the whole stack of Executor JVM cycles, and the Java related ratio changes from 66.8% to 73.6%. The right graph shows what Java related methods are frequently called. Without the lock reservation option, the deserializer part increases from 13.0% to 24.2% and serializer increases from 10.1% to 13.0%. As a result, we find that the lock reservation option improvement is derived from the serializer and deserializer. In Spark, the Kryo serializer is used as the default serialization implementation and the objects are serialized or deserialized when intermediate data is shuffled between other Executor JVMs, so the improvement ratio is in proportion to the size of shuffle data.

Runtime instrumentation enables compilation heuristics to be gathered for a JIT compiler. By profiling methods in several running threads from PMU, the JIT compiler decides which methods should be compiled and which compilation level is best. However, our results show that runtime instrumentation misdirects the JIT compiler for Q5.

D. JVM Counts

Then we evaluated multiple Executor JVMs on a single node shown in Figure 16. We varied the JVM counts between 1, 2, 4, and 8. The total number of worker threads and total aggregate heap size remain the same as those in a single JVM case; that is, total worker threads equal 48 and total

TABLE III
TESTED EXECUTOR JVM OPTIONS: RED MEANS APPENDED OPTION

#	spark.executor.extraJavaOptions
1	-Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
2	-Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
3	-Xnoloa -Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
4	-XlockReservation -Xnoloa -Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
5	-Xnocompactgc -XlockReservation -Xnoloa -Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
6	-XX:-RuntimeInstrumentation -Xnocompactgc -XlockReservation -Xnoloa -Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
7	-Xdisableexplicitgc -XX:-RuntimeInstrumentation -Xnocompactgc -XlockReservation -Xnoloa -Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none

heap equals 192GB. By increasing JVM counts, the assigned worker threads are changed between 48, 24, 12, and 6. Total heap size is also varied in the same manner.

This result shows that a single JVM is not always the best choice. In our experiment, utilizing 8 or more JVMs degrades performance due to overhead between JVMs, but using 2 or 4 JVMs can potentially achieve better performance than using a single JVM. Since NUMA aware CPU binding that we describe later should be taken into consideration for tuning JVM counts at the same time. Figure 16 also represents the best performance increase or the worst decrease ratio against a JVM as well. From this result, we observe that the smaller JVM counts is suitable for shuffle less query such as Q6. For shuffle heavy ones, 2 or 4 JVMs achieves better than other counts. Moreover, We observed the another typical tendency in Q16. Over 30% time is spent in spin lock within JVM, and this waste spin lock occurs on one JVM only. As a result, utilizing multiple JVM achieves 3x faster than one JVM.

In addition, multiple JVMs help to remove execution failure as a secondary effect. While queries are run on a single JVM with 48 worker threads, the execution often fails when calling the native snappy compression library during task execution. Due to this sudden failure, Spark tries to resubmit failed tasks. As a result, task execution time is often 2x slower than that in the no failure case. We have not yet found the fundamental reason this failure occurs while native snappy compression library is processed with many worker threads, but we believe that there is some non-thread safe code in the library.

E. NUMA Aware Affinity

Next, we evaluated the efficiency of applying NUMA aware affinity to Executor JVM to reduce remote NUMA memory access. Figure 17 shows the average query execution time of six iterations. In this experiment, we used four JVMs. We specified CPU affinity for each JVM by *numactl* command. As a result, they are assigned to corresponding NUMA node one by one if they enable NUMA. Also, each JVM

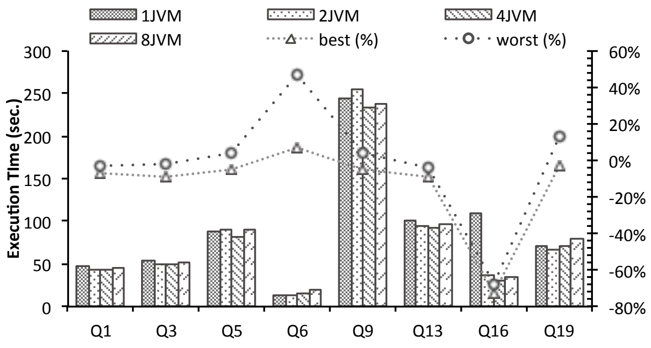

Fig. 16. Varying the number of JVMs between 1, 2, 4 and 8

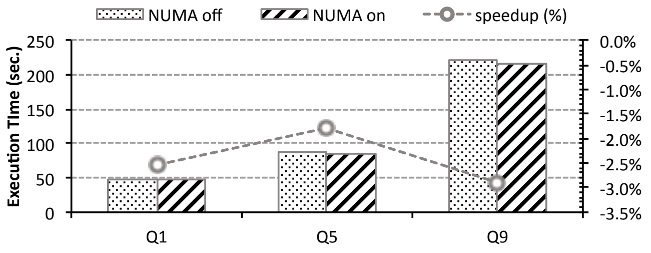

Fig. 17. Average execution time with NUMA aware CPU bindings

utilizes six worker threads. All queries were improved about 2 - 3% by considering NUMA locality, but the performance did not improve as much as we had expected.

In addition, we evaluated the scheduled CPU for worker threads and memory access events of PMU to estimate NUMA efficiency. We periodically capture where worker threads are running every five seconds and plot them to the physical CPU cores. By setting NUMA affinity, worker threads are scheduled only on the corresponding NUMA node, which means all worker threads get the benefit of memory locality. On the other hand, in results without NUMA affinity shown in Figure 18, the worker threads are scheduled over NUMA at first, and then the threads seem to be gathered into the same NUMA node. However, several

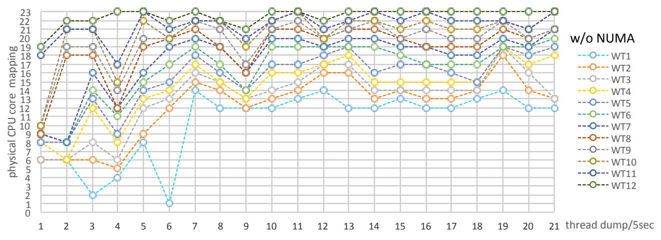

Fig. 18. Transition of worker threads on where they are mapped to actual CPU cores

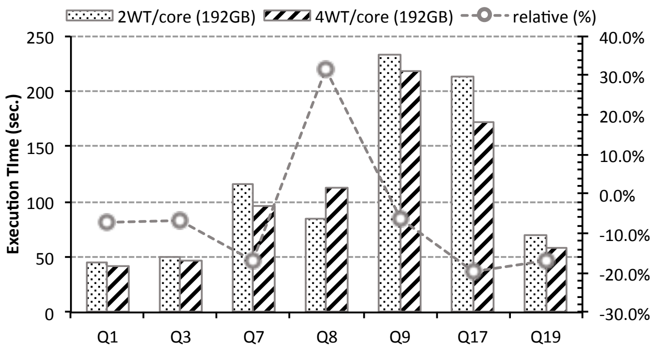

Fig. 19. Scalability Comparison of 48 and 96 worker threads over four Executor JVMs

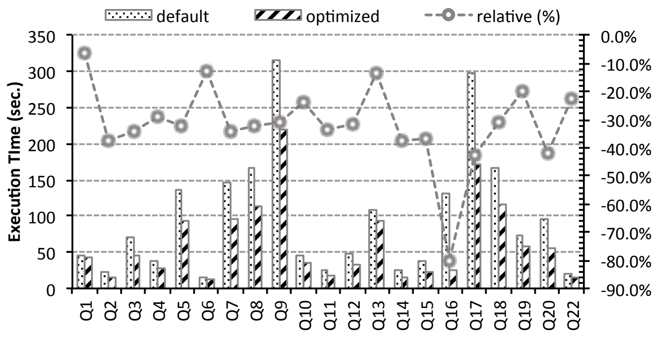

Fig. 20. Performance Comparison between default score and all optimization applied score

threads are often scheduled to another domain NUMA node due to the OS scheduler's manner. The Linux Completely Fair Scheduler (CFS) manages load balancing across CPUs and tries to schedule tasks while taking NUMA nodes into consideration. However, it does not always bind worker threads to the same NUMA node. As a result, worker threads need to access remote NUMA nodes at that time. We also confirmed that distant memory access events of PMU decreases from 66.5% to 58.9%, but the efficiency is very limited in this case.

F. Increasing Worker Threads and Summary Result

Finally, we increased the assigned worker threads from 48 to 96. From this change, the estimated running worker threads per physical core also increases from 2 to 4. Figure 19 represents that many queries receive the benefit of increase of hardware threads regardless of shuffle data size. Although Q8 and Q5 are the only two queries which has drawback, all of any other queries achieved 10 - 20% improvement. Especially, Q7 and Q17 that has unionall operation are drastically improved because unionall combines RDDs stored in memory simultaneously. Figure 20 concludes the comparison summary with applying all of optimizations. Shuffle less queries achieved 10 - 20% improvement. For shuffle heavy queries, we achieved basically 20% and up to 80% improvement. Q21 that took too much time with default can get result in 600 sec after tuning.

VI. Related Work

Several tuning guides and hints have been published on Spark's official site and developers blogs, but few research papers have discussed Spark performance, and no paper has done so from the perspective of JVM optimization and system optimization as far as we know. Kaczmarek et al. discussed GC tuning on OpenJDK [19] and used a G1GC based algorithm instead of generational GC. Since Spark is developed for running on computing clusters, data shuffling over network is an important optimization topic. Davidson and Or revealed that shuffle connections between nodes are increased by the product of a mapper and reducer and proposed an approach to consolidate shuffle data into one per destination [20]. Lu et al. proposed RDMA based data shuffling architecture for Spark [21] and showed that RDMA based shuffle outperformed existing NIO or Netty based communication layer. Shi et al. compared Hadoop with Spark in terms of performance and execution model and then evaluated several machine learning algorithms [22]. Their work is a little similar to us, but the target workload, profiling approach and the knowledge about tuning strategies are different.

VII. Conclusion and Future Work

In this paper, we characterized TPC-H Queries on Spark from many aspects such as application log, GC log, system utilization, method profiling and performance counters in order to establish a common optimization insights and existing problems for Spark. Our optimization strategies outperform up to 5x faster than default, and 10 - 40% improvement in many queries on average. GC cost is still high because of Spark in-memory feature and generating immutable objects massively, however we can reduce the GC overhead from 30% to 10% or less not by increasing heap memory unnecessarily but by optimizing JVM counts, options and heap sizing even if limited heap. Then, NUMA aware affinity takes a little advantages to prevent remote memory access, and SMT can increase IPC potentially as long as Spark runtime can keep data in heap and Spark runtime itself remove wasteful stall cycle more and more. Our analysis helps to improve Spark core runtime itself, apply various system side optimization approaches, and then brings a chance to develop more advanced algorithms about JVM including GC, thread scheduler, cluster scheduler, etc. for us. In our future work, we will plan to evaluate how our tuning is effective on other Spark workloads, and then move our focus into more deeper side of JVM and operating system.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[3] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 810–818, ACM, 2010.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, 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 Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 2–2, USENIX Association, 2012.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pp. 10–10, USENIX Association, 2010.

[6] 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," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1840–1843, 2015.

[7] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, p. 53, ACM, 2015.

[8] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 488–499, IEEE, 2014.

[9] spark perf, "https://github.com/databricks/spark-perf/."

[10] 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 2009 ACM SIGMOD International Conference on Management of data*, pp. 165–178, ACM, 2009.

[11] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proc. VLDB Endow.*, vol. 7, pp. 1295–1306, Aug. 2014.

[12] 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 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1383–1394, ACM, 2015.

[13] A. C. de Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, 2010.

[14] J. Levon and P. Elie, "Oprofile: A system profiler for linux," 2004.

[15] https://www-01.ibm.com/support/knowledgecenter/linuxonibm/, "CPI events and metrics for POWER8."

[16] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 392–403, ACM, 1995.

[17] K. Kawachiya, A. Koseki, and T. Onodera, "Lock reservation: Java locks can mostly do without atomic operations," in *ACM SIGPLAN Notices*, vol. 37, pp. 130–141, ACM, 2002.

[18] T. Onodera and K. Kawachiya, "A study of locking objects with bimodal fields," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 223–237, 1999.

[19] https://spark-summit.org/2015/events/taming-gc-pauses-for-humongous-java-heaps-in-spark-graph computing/.

[20] A. Davidson and A. Or, "Optimizing shuffle performance in spark," tech. rep., University of California, Berkeley - Department of Electrical Engineer- ing and Computer Sciences, Tec Rep., 2013.

[21] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with rdma for big data processing: Early experiences," in *High-Performance Interconnects (HOTI), 2014 IEEE 22nd Annual Symposium on*, pp. 9–16, IEEE, 2014.

[22] 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," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.