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

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

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

***Abstract*—Besides being an in-memory-oriented computing framework, Spark runs on top of Java Virtual Machines (JVMs), 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 a long garbage collection pause time, which accounts for over 10–20% of application execution time. Moreover, recent computing nodes have many cores with simultaneous multi-threading technology and the processors on the node are connected via NUMA, so it is difficult to exploit best performance without taking into account of these hardware features. Thus, optimization in a full stack is also important. Not only JVM parameters but also OS parameters, Spark configuration, and application code based on CPU characteristics need to be optimized to take full advantage of underlying computing resources. In this paper, we used the TPC-H benchmark as our optimization case study and gathered many perspective logs such as application, JVM (e.g. GC and JIT), system utilization, and hardware events from a performance monitoring unit. We discuss current problems and introduce several JVM and OS parameter optimization approaches for accelerating Spark performance. As a result, our optimization exhibits 30–40% increase in speed on average and is 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 at scale and with fault tolerance. Since this simple programming model enables us to implement distributed data-intensive applications more easily, various types of analytics applications such as relational data processing, machine learning, and graph algorithms have been built on Hadoop and its related ecosystem. However, they do not always work efficiently on the current Hadoop system because the current framework is not suitable for iterative and interactive analytics applications. As a result, multiple alternatives to Hadoop systems [1][2][3] have been developed to overcome this inefficiency.

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 memory as possible; therefore, it can drastically reduce disk I/O compared to Hadoop. Spark provides many operators and useful libraries for machine learning (MLLib), relational data processing (SparkSQL [6]), and graph computation (GraphX). It also provides APIs for parallel data collections in Scala, Java, Python, and R. Spark retains scalability and resiliency as well, so it has attracted much attention recently.

Although the new features are frequently applied to Spark, characterizing Spark internal and gaining deep insight into the tuning of Spark applications from the system side are important, because the knowledge is helpful to Spark users and system researchers who try to apply their own optimization to Spark. 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 layers (OS, Java Virtual Machine (JVM), runtime, etc.) have been used to achieve higher performance in accordance with their own optimization policies, so high performance is difficult to achieve unless these components cooperate.

To 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 directly affects its data processing performance. In contrast, maximization of I/O throughput is still important for Spark, but its performance bottlenecks are moved to the CPU, memory, and network communication layer because of Spark's in-memory data processing policies. Second, Spark runs on top of JVMs 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 garbage collection (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, we investigated the characteristics of Spark performance with multiple metrics through running Spark applications. We used TPC-H queries on Spark

as reference applications and captured JVM logs such as GC, just-in-time (JIT) compiled methods, hardware counter events, and system resource monitor logs. From these logs, we defined several problems on current Spark runtime, and then described JVM and OS parameter optimization approaches to solve them. Finally, we evaluated 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 the JVM side to reduce GC overhead without increasing heap memory and also to increase the instructions per cycle (IPC) by using non-uniform memory access (NUMA) and simultaneous multi-threading (SMT). (3) We also discuss potential problems we found through our experiments that would be useful for designing or developing JVM and Spark core runtime.

The rest of the paper is organized as follows. Section 2 describes the background of Apache Spark. Section 3 summarizes the TPC-H workload and breakdown measurement results from many metrics. Section 4 considers what is the current problems are for further accelerating Spark performance and how we can optimize each layer. Then, Section 5 describes the tuning results. Section 6 mentions related work. Finally, Section 7 concludes 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. It keeps as much intermediate data in memory as possible to order reduce data loading latency; therefore, it performs much better than Hadoop in iterative and interactive workloads such as machine learning and data mining. It also runs other workloads such as batch jobs and relational queries efficiently. Spark has been used as a general-purpose distributed computing engine [7].

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 a directed acyclic graph (DAG). The transformation tasks are processed for each partitioned data and the data shuffling tasks are launched if the transformation task requires the data to be shuffled across these partitioned data. Resilient distributed datasets are evaluated lazily, so computation tasks are not launched before applying action types of operations such as *count*. To process the divided tasks, Spark manages many worker threads within an Executor JVM.

B. Spark Benchmarks and Applications

Spark benchmark suites [8][9][10] have been recently developed that provide a comprehensive set of Spark workloads, such as machine learning, graph processing, and query processing, together with synthetic data generators for each workload. Among these workloads, query processing is the most popular in the Spark community [11]. However, these benchmark suites only support simple queries.

TPC-H [12] is a decision support benchmark consisting of a suite of business-oriented ad-hoc queries and concurrent data modifications, defined as 22 queries for 8 different sized tables. The TPC-H benchmark was originally used for evaluating database systems but has recently been used for Hadoop-based query engines [13][14]. The TPC-H benchmark includes a wide variety of queries (e.g. simple data selection, aggregation and multiple types of join operations). Accordingly, when TPC-H is run on Spark, the generated code has a wide variety of characteristics. TPC-H is thus one of the ideal benchmarks to study the performance of query processing on Spark.

C. Performance Measurement Tools

Spark provides statistical reports about multiple metrics of executed jobs. For example, the report shows how many stages are executed for a job and how many tasks are executed for a task. It also shows the duration and the sizes of input, output, shuffle read, and shuffle write for each task. These metrics provide us with helpful information about the execution of an application at the Spark level, guiding us in tuning the application. However, to fully optimize it, we need to understand the whole picture about the execution, getting profiling information from the other layers. Thus, we use various performance monitoring tools including IBM Monitoring and Diagnostic Tools for Java [15] for tracking Java-level performance, a statistical profiler for Linux, OProfile [16], for profiling all the running code, the Linux perf command [17] for counting hardware events, and the Linux sysstat utilities for tracking the system-level performance.

III. SPARK WORKLOAD ANALYSIS

A. Experimental Platform and Setup of TPC-H Workloads

We conducted all experiments on a single node of POWER System S824L, which is equipped with two 3.3 GHz POWER8 processors. Each processor has 12 cores, while each core has a private 64 KB L1 cache, private 512 KB L2 cache, and shared 96 MB L3 cache. This system has 1 TB of RAM and 1 TB RAID5 disk. The software stack of this system consists of Ubuntu 14.10 (kernel: 3.16.0-31-generic), Hadoop 2.6.0, and Spark 1.4.1, and the IBM J9 JVM (1.8.0 SR1 FP10).

Regarding a TPC-H dataset, we generated TPC-H table data files using an official data generator with a 100 GB scale factor then loaded them into Hive tables on Hadoop Distributed File System (HDFS). The chunk size of HDFS is

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

	Key Characteristics of Queries	Converted Spark Stage	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	3145
Q22	1groupby, 1outer join, 3table	3load, 1OuterJoin+CartesianProduct, 1Aggregate	0.6	1	5/571	27.6

128 MB. The original data sizes of all tables are as follows: lineitem is 75 GB, orders 17 GB, partsupp 12 GB, customer 2.3 GB, part 2.3 GB, supplier 137 MB, nation 2.2 KB, and region 389 B. All tables are stored in the Parquet columnar format [18] and compressed with Snappy [19].

We used TPC-H Hive queries published at github [20] as a basis because SparkSQL has compatibility to directly execute Hive queries on Spark runtime. However, since some queries did not finish or take too long, we revised them as follows. First, we eliminated temporary table creation since it forces writing to HDFS and reduce opportunities for Spark to generate a better execution plan. Second, we changed the order for some joins, since the version of the Spark runtime we used generated an inefficient execution plan from the original order.

B. Queries Characterization at SQL and Spark Levels

Table I shows the query response time on a single Spark Executor JVM with 48 worker threads and 192 GB heap. It also lists key characteristic of queries, generated spark operators, total input data size loaded from HDFS, total shuffled data size between stages, and total number of stages and tasks. We summarized only the key operations of each query, together with the number of tables accessed, since they will affect the Spark query execution plan. For example, Q5 performs a groupby operation and five inner join operations with six tables. These operations are converted into an 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. During execution, Q5 loaded a total of 8.8 GB of data at three data loading stages, and shuffled 14.1 GB of data between join stages. As a result, Q5 took 125 second until 48 worker threads completed 8 stages with 1547 tasks.

From these results, we can find trends and characteristics for these TPC-H queries. First, the queries 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. Their performances depend more than others on the data loading from the disk. Second, queries Q5, Q7, Q8, Q9, and Q18 have huge shuffling data and also take over 100 sec. These have more shuffle data than input data, so their performances depend on the data shuffling between stages. Based on these observations, we could categorize the queries into two, shuffle light and shuffle heavy.

C. Queries Characterization at JVM and OS Levels

Next we characterized more details about TPC-H queries based on the results of OProfile. Figure 1 shows the break-down of the profiling results of sampled methods into key components. The java component represents application code and Spark runtime code, while snappy does the native library call for compression. The components of j9gc, j9jit, j9thr and j9vm include JVM level methods related to GC, JIT, multithreading, and others internal to JVM, respectively. The kallsyms represents the overhead of using OProfile. Figure 2 focuses on Java-related methods, categorizing them into four components. SparkSQL represents actual computation, parquet related to I/O, such as data loading, and serialization

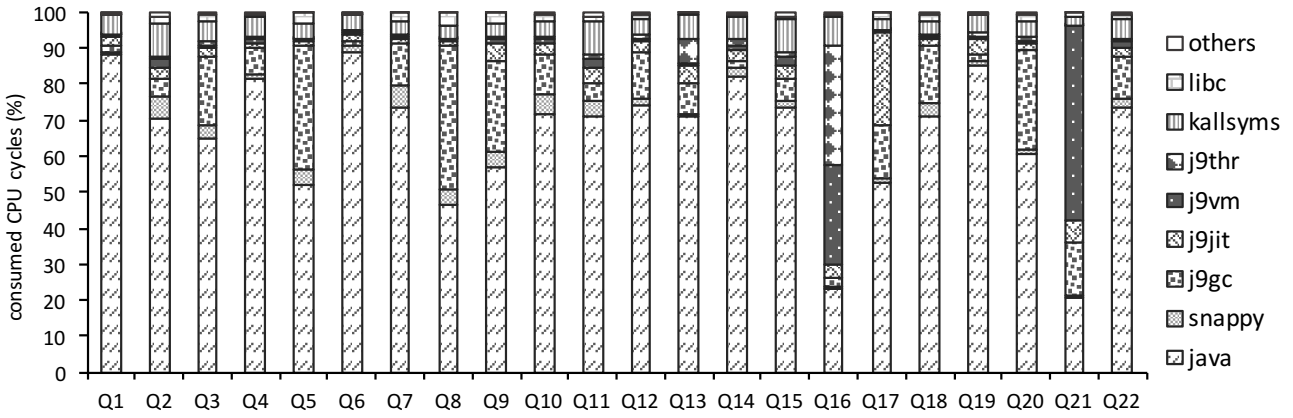

Fig. 1. CPU cycles of nine categorized components from Oprofile sampling results for all of queries

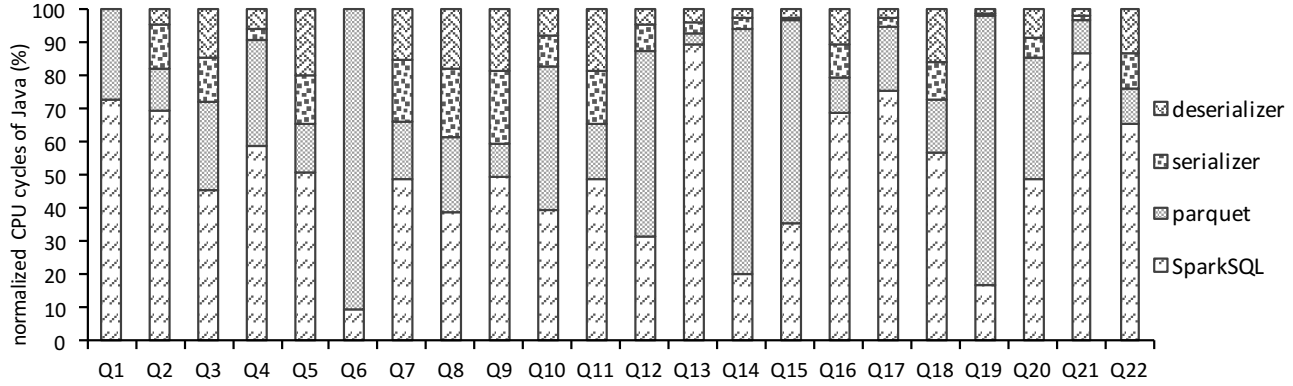

Fig. 2. Normalized CPU cycles of details regarding Java-related processes

and deserialization concerns data shuffling. These results were also collected on a single Spark Executor JVM.

We can observe distinctive characteristics between shuffle-heavy and shuffle-light in these figures. The queries that have large shuffle data such as Q5 and Q8 pay relatively higher GC cost and (de)serialization than others. On the other hand, the queries that have few shuffle data such as Q1 and Q19 spend a lot of time consuming in computation or data load. We also find unexpected j9thr and j9vm overhead in Q16, and j9vm overhead in Q21. The further analysis showed that Q16 spent much time on spin locks (thus high j9thr overhead), which Q16 and Q21 executed many methods with the bytecode interpreter.

In the following subsections, we describe the metrics at the JVM and system levels in details, focusing on Q1 and Q5 as representatives of shuffle-light and shuffle-heavy, respectively.

D. GC and JIT Behaviors

First, we analyzed the behaviors of garbage collection. Figures 3 and 4 show heap usage and GC pause time while executing Q1 and Q5, respectively. By default, the 192GB heap was divided into the 48GB nursery space and the 144GB tenure space. For Q1, almost all the objects created in the nursery space were collected after each nursery GC. As a result, the pause times were quite small, less than 0.2 second. For Q5, on the other hand, GC behaviors were completely different. Running out of nursery space, objects were gradually promoted into the tenure space then collected when the tenure space became full. We observe that the total used heap in Q5 decreased when global GCs were happened around 500 and 700 seconds. Most surprisingly, the pause times by the nursery GCs were longer than the global GCs. In summary, Q1 spent about 2% of the execution time for GC and Q5 spent more than 34%, based on the profiling results in Figure 1 and 2.

We then checked how many methods were compiled at each optimization level by analyzing the JIT log. The total number of compiled methods reached over 8,000. Almost all the methods were compiled at the warm level, while a few methods reached scorching, which is the highest optimization level. We also confirmed that the compilation levels of the

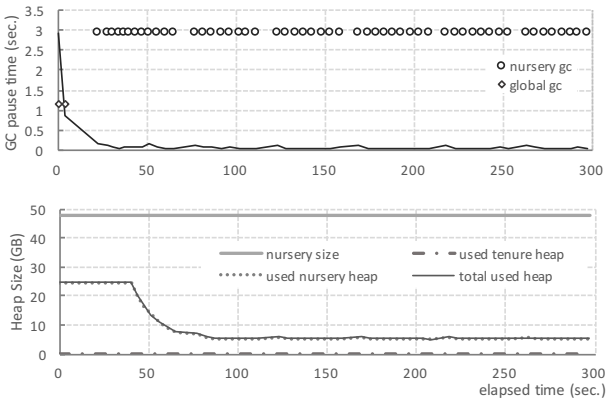

Fig. 3. GC and heap statistics of default Q1: pause time (upper), heap usage (lower)

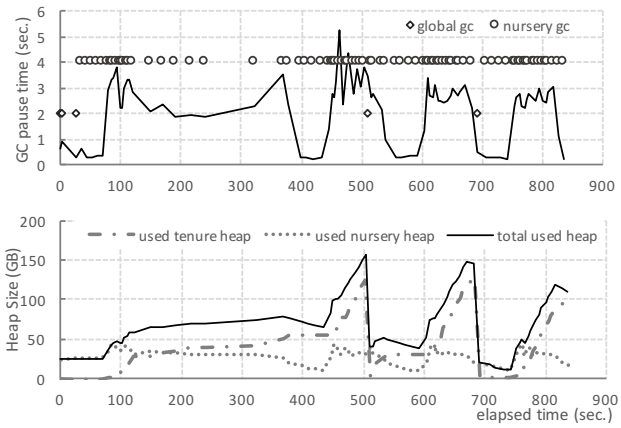

Fig. 4. GC and heap statistics of default Q5: pause time (upper), heap usage (lower)

top ten methods reached warm, hot, and scorching.

Next, we evaluated the actual query response time of Q1 and Q5 in each iteration. Figure 5 shows the results of Q1 and Q5 through six iterations. The first iteration was about 1.5x–2x slower than other iterations in both queries because the JIT compilation for hot methods had not yet finished. To be more precise, the first round of each stage took much longer. For example, the Q1 execution plan consisted of two stages. Data loading in the first and second stages were divided into 593 and 200 tasks, These tasks are assigned to 48 worker threads in the Executor JVM. If evenly assigned, each worker thread will process 12 or 13 tasks for the first stage, which means the stage consists of 13 rounds. The threads in the second round of a stage used more optimized code, so the execution time was 1.5–2x faster than in the first round. A similar observation can be made for all eight stages in Q5.

Finally, we often observed failures in both queries, such as the fourth iteration of Q5 in Figure 5. The failure occurred on the single JVM configuration, which launches many worker threads. Java stack trace indicated the failure happened when calling the native snappy compression library. Tracing the cause of this is beyond the scope of our paper, but using one large JVM brings performance fluctuation.

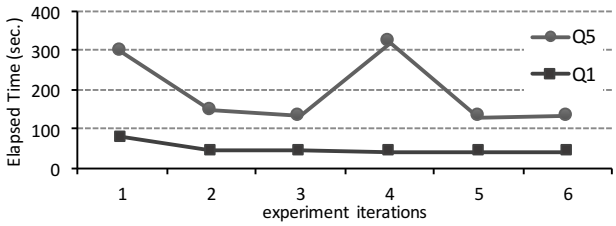

Fig. 5. Transition of Q1 and Q5 response time through six time iterations

TABLE II
PMU PROFILING RESULT FOR Q1 AND Q5

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-backend	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

E. System Utilization

We evaluated system utilization including CPU, memory, and I/O context switches. Figures 6 and 7 show CPU utilization and memory usage for Q1, and Figures 8 and 9 show them for Q5. Both queries ran six times continuously on the same Executor JVM. We can see that 25% of CPU resources were used for user time in both queries, and then I/O wait time and system time are very few. Due to the use of only 48 worker threads for 192 logical cores, we confirm that all worker threads fully consume CPU resources. In the Q5 CPU usage graph, we can see some spikes in the later part of iteration. This spikes are caused by heavy GC activity in the shuffle phase. From the perspective of memory usage, used memory and page cache increase after iterations start. The total used memory in the system when running Q1 does not exceed around 70 GB, while that in Q5 reaches 220 GB.

F. PMU Profiling

Finally, we captured performance counter events from a performance monitoring unit (PMU). Table II lists the ”perf stat” command results. Both queries accounted for 50–60% of wasteful CPU stall cycles on the backend pipeline. Due to this large backend stall, the IPC rate remained low. To determine from where this backend stall originated, we gathered hardware counter events and analyzed them to the CPI breakdown model described in [21]. As a result, we found that the wasteful stall originated from L3 cache miss and this cache miss caused by mainly distant memory access.

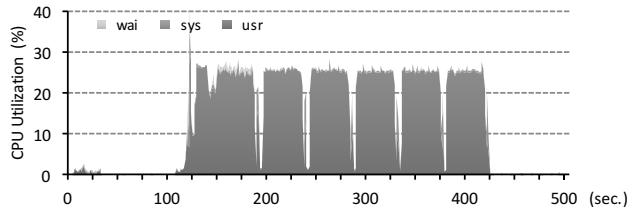

Fig. 6. CPU utilization while running Q1 with default

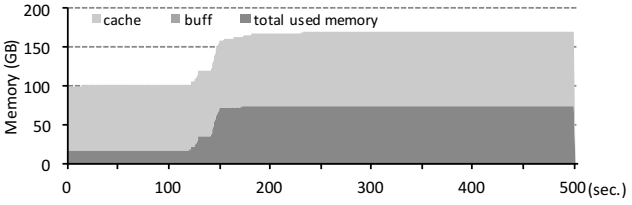

Fig. 7. Memory utilization while running Q1 with default

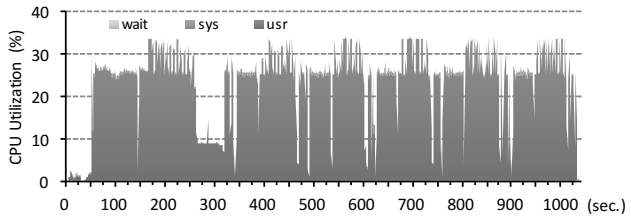

Fig. 8. CPU utilization while running Q5 with default

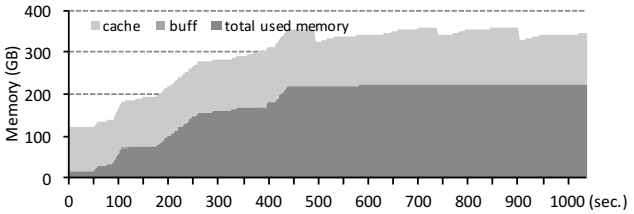

Fig. 9. Memory utilization while running Q5 with default

Table II also shows that CPU migrations occur frequently, especially in Q5.

IV. Problem Assessment and Optimization Strategy

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

A. How to Reduce GC Overhead

First, GC overhead affects application performance, as shown in Figure 4. Choosing an 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 the 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 used generational GC in this paper. In terms of heap sizing, a large heap likely causes a long pause time when global GC occurs. The small nursery space also could make pause time longer due to heavy copying, as we observed in Q5 (Figure 4). Consequently, we should search for the optimal combination of the total heap and 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. This is another approach to change heap size. Keeping each JVM heap small helps to directly reduce GC cost, so we should evaluate the effect of changing the number of JVMs. 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 virtually available from the 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 affect not only GC performance but also application performance.

Hence, we try to find the 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 the number of Executor JVMs while maintaining the total heap and working thread count.

B. How to Accelerate IPC and Reduce Stall Cycles

As shown in Table II, the IPC for TPC-H queries on Spark was not very high. One approach to accelerate IPC is to use more SMT features [22]. POWER8 supports up to the 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 mode. Of course, we can assign 192 worker threads in the Executor JVM but cannot expect large 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 performance may be further improved by increasing worker threads to 96 in total. In this case, we ideally expect four worker threads to run on the same physical core.

The other problem that retards the increase in IPC is huge stall in the backend pipeline. In our investigation into the 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

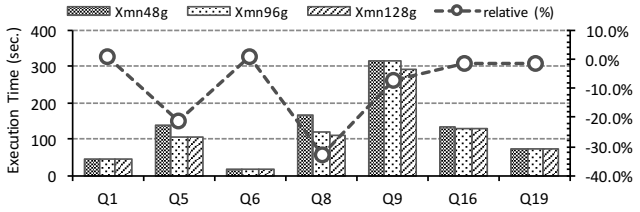

Fig. 10. Performance Comparison while changing nursery heap size

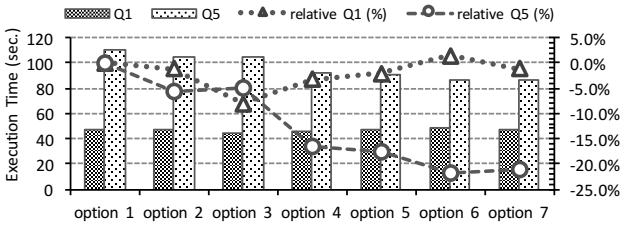

Fig. 11. Comparison of JVM options

precisely, each POWER8 processor has two NUMA nodes, so we could use four NUMA nodes. Therefore, we may improve application performance by pinning NUMA-aware affinity for Spark Executor JVMs.

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

V. Performance Evaluations

We applied several of the optimization ideas described above then evaluated how application performance is improved by better controlling JVM and OS behaviors. We selected typical queries for evaluation: Q1, Q6 and Q19 are representative of shuffle-light queries and Q5, Q8 and Q9 are shuffle-heavy ones. We also picked up Q16 since its characteristics is unusual as described in Section III.

A. Heap Sizing

First, we changed the nursery-heap size from 48 to 96 or 128 GB, which are halves or three-fourths of the total heap. The J9 JVM reserves one-fourth of the total heap as nursery space by default, which is 48 GB in our setting. Figure 10 shows query execution time and its change ratio compared to the default of 48 GB nursery heap. For instance, the execution time of Q5 decreased by 20% or changed by -20%.

As seen in the figure, Shuffle-light queries are insensitive because heap usage was basically small in these queries and GC did not frequently occur. On the other hand, most shuffle-heavy queries improved by 20–30%. By increasing the nursery heap, we can lower the frequency of nursery GC. In addition, this results in fewer objects being promoted from the nursery to the tenure space, helping reduce the number of global GC. Actually, there was no global GC for Q5 and Q8 while query execution ran six times. The performance of Q9 improved by only 10%, since the global GC still occurred periodically.

B. JVM Option Sweep

Next, we changed several JVM options as listed in Table III. While there are many selectable JVM options, we focused on those which control the behaviors of the key components such as GC and locks. To evaluate which JVM option contributes to performance improvement, we appended one JVM option in each configuration starting from option set 1. The experiments were run six times per configuration for Q1 and Q5, picking up one from each of shuffle-light and shuffle-heavy. Figure 11 shows the query execution time as well as its change ratio compared with option set 1. As shown in the figure, while Q1 is relatively insensitive, most JVM options helped improve query execution time for Q5. The improvements were especially drastic when lock reservation with "-XlockReservation" (option set 4).

TABLE III
Tested JVM option set: bold denotes 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

Lock reservation [23] enables a lock algorithm to be optimized when a thread frequently acquires and releases a lock. 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 [24]. In Spark, since many worker threads are running for processing their respective RDD partitions, performance will be improved if synchronized methods are heavily called for RDD processing.

To further investigate why the lock reservation is so effective, we analyzed method-call stacks via oprofile. Figure 12 shows the results with and without the lock reservation option, breaking down the execution time by major components in Spark execution. The left graph presents the breakdown of the whole cycles of the Executor JVM cycles. The ratio of the java component changed from 66.8 to 73.6%. The right graph

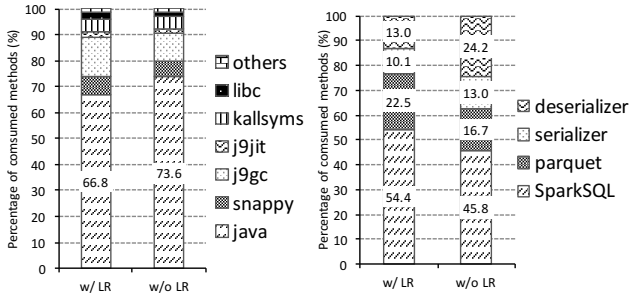

Fig. 12. Profiling method stack with and without lock reservation option. Categorized all of related processes (left) and drill-down Java-related tasks (right)

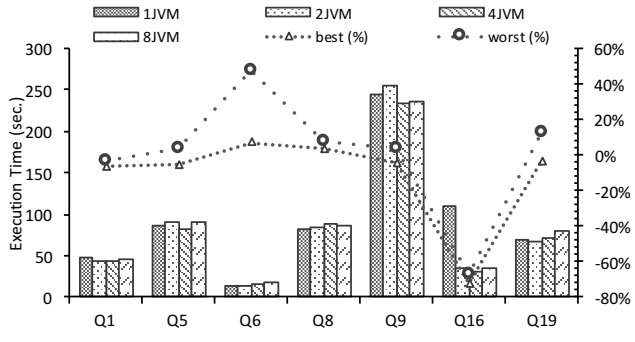

Fig. 13. Varying the number of JVMs between 1, 2, 4 and 8

shows the further breakdown for the java component. Without the lock reservation option, the consumption by deserializer increased from 13.0 to 24.2%, while that by serializer part increased from 10.1 to 13.0%. As a result, we found that the improvement in the lock reservation option is derived from the serializer and deserializer. In Spark, Kryo is used for this purpose as the default implementation. Objects are serialized or deserialized when intermediate data are shuffled between other Executor JVMs. The improvement ratio is thus in proportion to the size of shuffle data.

C. Number of JVMs

We then evaluated multiple Executor JVMs on a single node, as shown in Figure 13. We varied the number of JVMs between 1, 2, 4, and 8. The total number of worker threads and total aggregate heap size remained the same as those for a single JVM; total worker threads equaled 48 and total heap equaled 192 GB. By increasing the number of JVMs, the assigned worker threads changed between 48, 24, 12, and 6. The total heap size also varied in the same manner.

We can make three observations from These results. First, a single JVM is not always the best choice. Second, using eight or more JVMs tends to degrade performance. This is due to the communication overhead between JVMs. Finally, using two or four JVMs mostly exhibit better performance than using a single JVM.

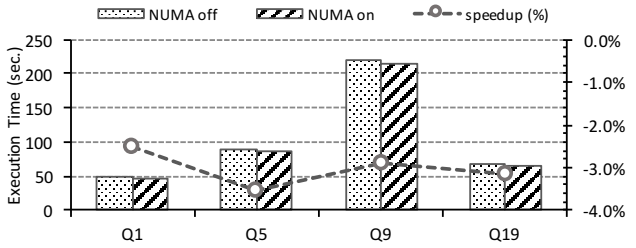

Fig. 14. Average execution time with NUMA aware CPU bindings

Figure 13 also illustrates the best and the worst change ratio against the single JVM configuration. We observe that a smaller number of JVMs is suitable for a shuffle-light query such as Q6. For shuffle-heavy ones, 2 or 4 JVMs achieves better than other numbers of JVMs. The results for Q16 is striking. Using multiple JVMs is 3x faster than the single JVM. Our analysis showed that over 30% of time is spent in spin lock for the single JVM. Because of many more threads per JVM, the single thread is more vulnerable to this issue of contention.

In addition, we often observed that, while executing queries on the single JVM, tasks fail when calling the native snappy compression library. Since Spark resubmit failed tasks, the tasks often become 2x slower than in the no-failure case. We have not yet found the fundamental reason why this failure occurs while the native snappy compression library run with many worker threads, but we suspect that there are some non-thread safe codes in the library.

Finally, when we run a Spark application with multiple JVMs on a node, the NUMA-aware CPU binding is critical, which we describe in the next section.

D. NUMA Aware Affinity

We evaluated the efficiency of applying NUMA-aware scheduling to Executor JVMs to reduce access to remote NUMA nodes. Figure 14 shows the average query execution time of six iterations. In this experiment, we used four JVMs, each having six worker threads. We specified CPU affinity for each JVM by the *numactl* command. As a result, they were assigned to the corresponding NUMA node individually. All queries improved by about 2–3% by considering CPU affinity, but performance did not improve as much as we had expected.

We also evaluated the scheduled CPU for worker threads and memory access events of a PMU to estimate NUMA efficiency. We periodically captured where worker threads were running every 5 seconds and plotted them to the physical CPU cores. By setting CPU affinity, worker threads were scheduled only on the corresponding NUMA node, which means all worker threads benefited from memory locality. On the other hand, in the results without NUMA affinity shown in Figure 15, the worker threads were first scheduled over NUMA then the threads seemed to gather into the same

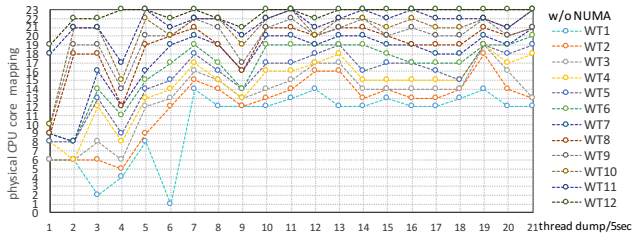

Fig. 15. Transition of worker threads on where they are mapped to actual CPU cores

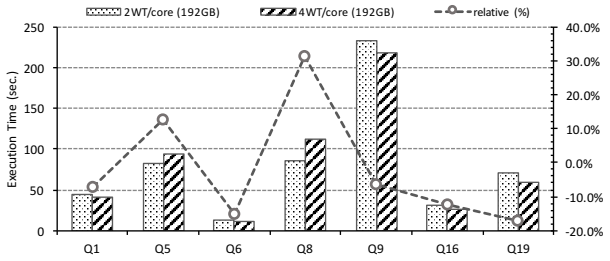

Fig. 16. Scalability comparison of 48 and 96 worker threads over 4 Executor JVMs

NUMA node. However, several 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 attempts to schedule tasks while taking NUMA nodes into account. However, it does not always bind worker threads to the same NUMA node. As a result, worker threads must access remote NUMA nodes at that time. We also confirm that distant memory access events of a PMU decreases from 66.5 to 58.9%, but efficiency is very limited in this case.

E. Increasing Worker Threads and Summary Results

Finally, we increased the assigned worker threads from 48 to 96. From this change, the estimated running worker threads per physical core also increased from 2 to 4. Figure 16 illustrates that many queries benefited from an increase in hardware threads regardless of shuffle-data size. Although Q5 and Q8 were the only two queries that had a drawback, all other queries achieved 10–20% improvement.

Figure 17 shows a comparison summary of applying all optimizations. Shuffle-light queries achieved 10–20% improvement. For shuffle-heavy queries, we achieved basically 30–40% improvement. Unusual characteristic queries such as Q16 and Q21 were drastically improved by changing the number of JVMs and choosing optimal JVM option set to reduce GC. Although the Q21 result which took over 3,000 seconds with the default is not shown in Figure 17, it takes about 543 seconds after tuning.

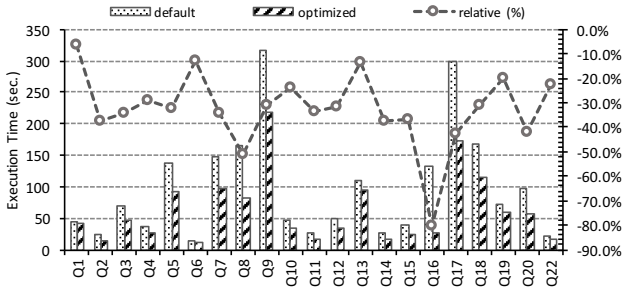

Fig. 17. Performance comparison between default score and all optimization applied score

VI. Related Work

Several tuning guides and hints have been published on Spark's official site and developer 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 [25] and used a G1GC-based algorithm instead of generational GC. Since Spark was developed for running on computing clusters, data shuffling over network is an important optimization topic. Davidson et al. revealed that shuffle connections between nodes increase due to the product of a mapper and reducer and proposed an approach to consolidate shuffle data into one per destination [26]. Lu et al. proposed a RDMA-based data shuffling architecture for Spark [27] and showed that RDMA-based shuffle outperformed current a NIO- or Netty-based communication layer. Shi et al. compared Hadoop with Spark in terms of performance and execution model and evaluated several machine learning algorithms [28]. Their work is similar to ours, but the target workload, profiling approach, and knowledge about tuning strategies are different.

VII. Conclusion and Future Work

We characterized TPC-H queries on Spark from many aspects such as application log, GC log, system utilization, method profiling, and performance counters to establish a common optimization insight into and existing problems with Spark. Our JVM and OS parameter optimization strategies performed up to 5x faster than the default, and 30–40% improvement in many queries on average. The GC cost is still high because of Spark ' s in-memory feature and generation of a massive amount of immutable objects; however, we reduced the GC overhead from 30 to 10% or less not by increasing heap memory unnecessarily but by optimizing the number of JVMs, options, and heap sizing even with a limited heap. Then, NUMA-aware affinity is slightly advantageous in preventing remote memory access, and SMT can potentially increase IPC as long as Spark runtime can keep data in a heap. Our analysis will help to improve Spark core runtime,

apply various system-side optimization approaches, and provide the opportunity to develop more advanced algorithms regarding a JVM including GC, thread scheduler, cluster scheduler, etc. For future work, we plan to evaluate how our tuning is effective on other Spark workloads and focus more on the JVM and OS.

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, 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.

[7] 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.

[8] 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.

[9] 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.

[10] "Spark performance tests." https://github.com/databricks/spark-perf/.

[11] M. Zaharia, "How spark usage is evolving in 2015," *Spark Summit Europe 2015*, 2015.

[12] "TPC-H benchmark." http://www.tpc.org/tpch/.

[13] 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.

[14] 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.

[15] "IBM Monitoring and Diagnostic Tools for Java." https://www.ibm.com/developerworks/java/jdk/tools/.

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

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

[18] "Apache Parquet." http://parquet.apache.org/.

[19] "Snappy." http://google.github.io/snappy/.

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

[21] "CPI events and metrics for POWER8." https://www-01.ibm.com/support/knowledgecenter/linuxonibm/liaal/iplsdkcpieventspower8.htm.

[22] 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.

[23] 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.

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

[25] E. Kaczmarek and L. Yi, "Taming gc pauses for humongous java heaps in spark graph computing," *Spark Summit 2015*, 2015.

[26] 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.

[27] 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.

[28] 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.