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

Tatsuhiro Chiba and Tamiya Onodera

IBM Research - Tokyo

April. 17-19, 2016 | IEEE ISPASS 2016 @ Uppsala, Sweden

Overview

- Introduction
  - Motivation
  - Goal and Result
- Workload Characterization
  - How Spark and Spark SQL work
  - Environments
  - Application level analysis
  - System level analysis
  - GC analysis
  - PMU analysis
- Problem Assessments and Approach
- Result
- Summary and Future Work

Motivation

Apache Spark is an in-memory data processing framework, runs on JVM
Spark executes Hadoop similar workloads, but ***optimization points are not same***

- Complexity to find a fundamental Bottlenecks
  - I/O bottleneck → CPU, Memory, Network bottleneck
  - JVM handles many worker threads

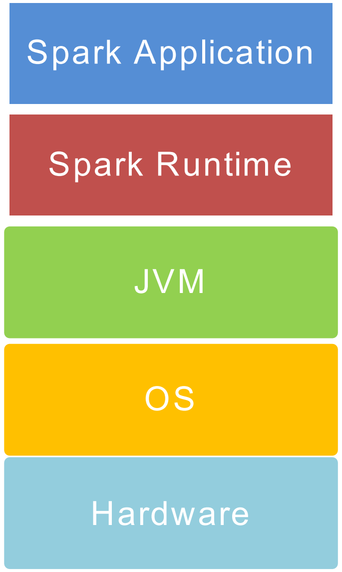
    → What is a best practice to achieve high performance?

- Managing large Java heap causes high GC overhead
  - Keeps as much data in memory as possible
  - Generates short-lived immutable Java objects

    → How we can reduce garbage collection overhead?

- From Scale-out to Scale-up
  - Utilizes many worker threads w/ SMT on multiple NUMA nodes
  - Need to know micro architectural efforts

    → How we can exploit underlying Hardware features?



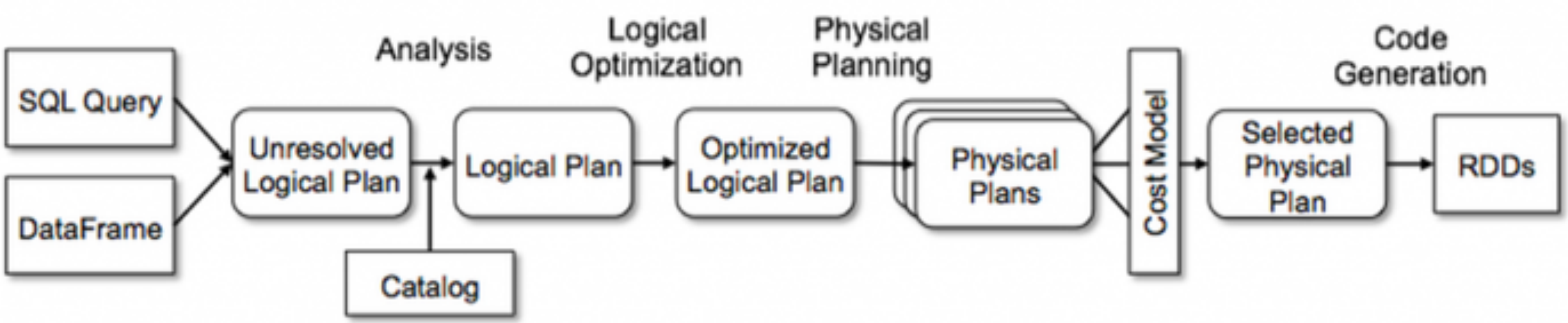
Goal and Result

- Goal
  - To characterize Spark Performance through SQL workloads
  - To find optimization best practice for Spark
  - To investigate potential problem in current Spark and JVM for scaling up

- Result
  - Achieved up to 30% improvement by reducing GC overhead
  - Achieved up to 10% improvement by utilizing more SMT
  - Achieved up to 3% improvement by NUMA awareness
  - Achieved **30 – 40% improvement** on average with applying all optimization

How Spark and Spark SQL work

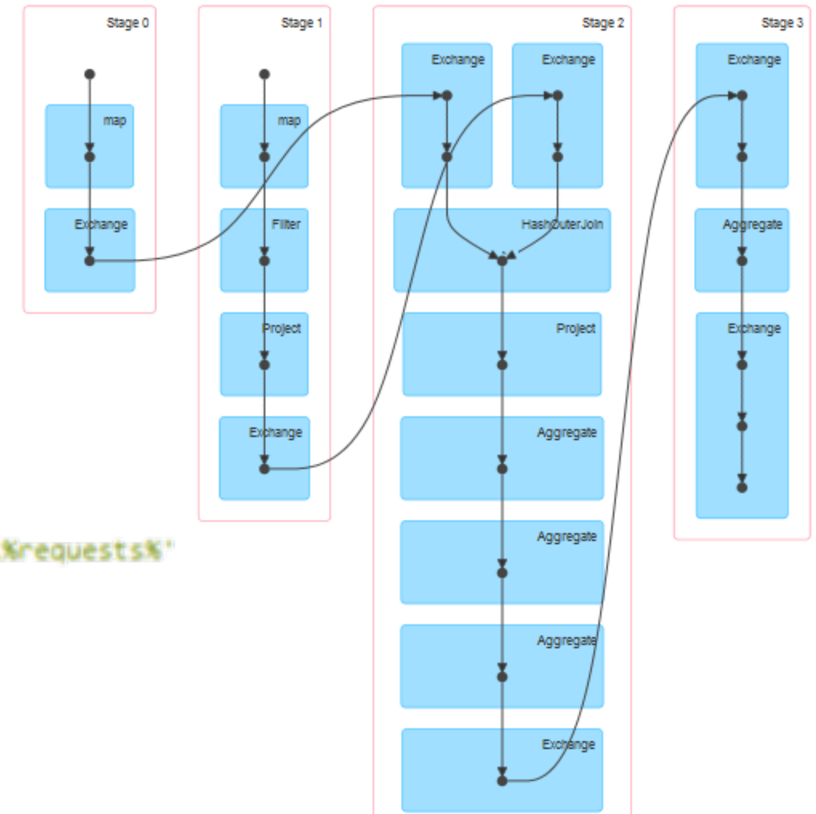
- Spark
  - Job is described as ***a data transformation chain*** and divided into multiple stages
  - Each stage includes multiple tasks
  - Tasks are concurrently proceeded by worker threads on JVMs
  - Data shuffling occurs between stages
- Spark SQL
  - Catalyst, a query optimization framework for Spark, generates an optimized code
  - It has a compatibility for HIVE query



Cited from Michael et al., **Spark SQL: Relational Data Processing in Spark** , *SIGMOD'15*

How SQL code translates into Spark - TPC-H Q13

- stage 0 : Load CUSTOMER table named as 'c'
- stage 1 : Load ORDERS table named as 'o'
- stage 2 : Join c and o where c.custkey = o.custkey
- stage 3 : Select c_count, count(1) groupby c_count




```sql
select c_count, count(1) as custdist
from (
   select c_custkey, count(o_orderkey) as c_count
     from
     customer c left outer join (
        select o_custkey, o_orderkey
        from orders where not o_comment like '%special%requests%'
     ) o on c.c_custkey = o.o_custkey
   group by c_custkey
) c_orders
group by c_count
order by custdist desc, c_count desc;
```

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	select c_count, count(1) as custdist from (select c_custkey, count(o_orderkey) as c_count from... Spark JDBC Server Query +details	2015/06/24 04:35:34	0.4 s	200/200			693.4 KB	
2	select c_count, count(1) as custdist from (select c_custkey, count(o_orderkey) as c_count from... Spark JDBC Server Query +details	2015/06/24 04:34:47	47 s	200/200			15.0 GB	693.5 KB
1	select c_count, count(1) as custdist from (select c_custkey, count(o_orderkey) as c_count from... Spark JDBC Server Query +details	2015/06/24 04:31:48	3.0 min	1348/1348	33.3 GB			14.2 GB
0	select c_count, count(1) as custdist from (select c_custkey, count(o_orderkey) as c_count from... Spark JDBC Server Query +details	2015/06/24 04:31:48	12 s	185/185	400.1 MB			817.6 MB

Machine & Software Spec and Spark Settings

Processor	# Core	SMT	Memory	OS
POWER8 3.30 GHz * 2	24 cores (2 sockets * 12 cores)	8 (total 192 hardware threads)	1TB	Ubuntu 14.10 (kernel 3.16.0-31)

software	version
Spark	1.4.1
Hadoop (HDFS)	2.6.0
Java	1.8.0 (IBM J9 VM SR1 FP10)
Scala	2.10.4

- Baseline Spark settings
  - # of Executor JVMs: 1
  - # of worker threads: 48
  - Executor heap size: 192GB (nursery = 48g, tenure = 144g)

- Other picked up Spark configurations (spark-defaults.conf)
  - spark.suffle.compress = true
  - spark.sql.parquet.compression.codec = Snappy
  - spark.sql.parquet.fileterPushdown = true

Workload Characterization – Spark job level

* Picked up several queries

Query	SQL Characteristics	Converted Spark Operation (# of stages)	Input (total, GB)	Shuffle (total, GB)	Stages / Tasks	Time (sec)
Q1	1 GroupBy Load 1 Table	1 Load 1 Aggregate	4.8	0.002	2 / 793	48.7
Q3	1 GroupBy, 2 Join Load 3 Table	3 Load 2 HashJoin, 1 Aggregate	7.3	5.0	6 / 1345	64.6
Q5	1 GroupBy, 5 Join Load 6 Table	3 Load, 3 HashJoin 1 BcastJoin, 1 Aggregate	8.8	14.1	8 / 1547	125
Q6	1 Select, 1 Where Load 1 Table	1 Load 1 Aggregate	4.8	0	2 / 594	15.1
Q9	1 GroupBy, 5 Join Load 6 Table	4 Load, 4 HashJoin 1 BcastJoin, 1 Aggregate	11.8	34.4	10 / 1838	370
Q18	3 Join, 1 UnionAll Load 3 Table	6 Load, 3 HashJoin 1 Union, 1 Limit	7.7	13.8	11 / 3725	202
Q19	3 Join, 1UnionAll Load 2 Table	6 Load, 3 HashJoin 1 Union, 1 Aggregate	19.8	0.4	8 / 2437	80.8

Shuffle-light queries

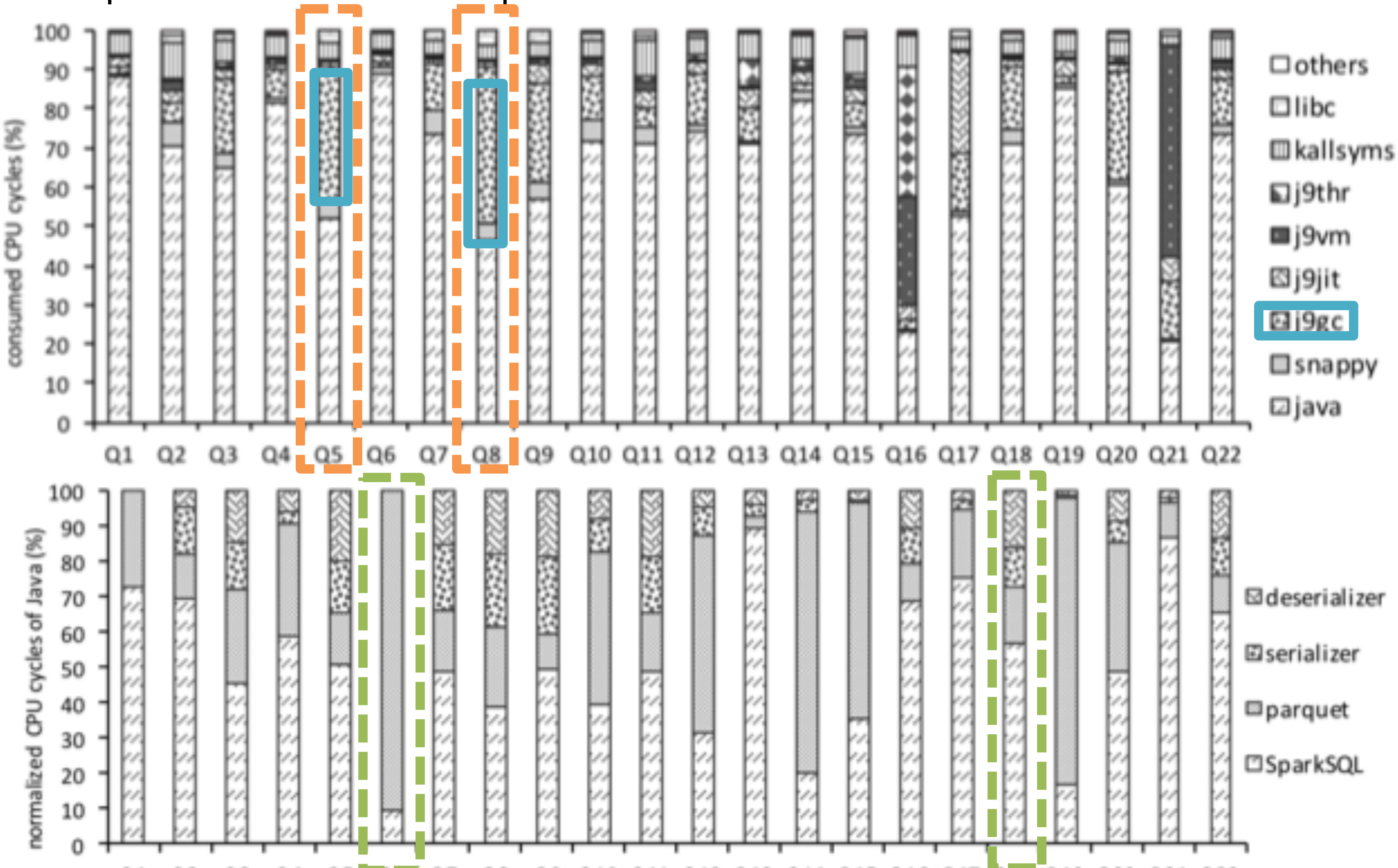
Q1, Q6, Q19

Shuffle-heavy queries

Q5, Q9, Q18

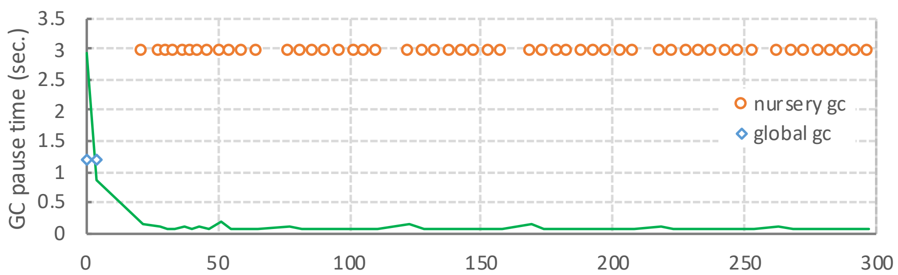
Workload Characterization – oprofile

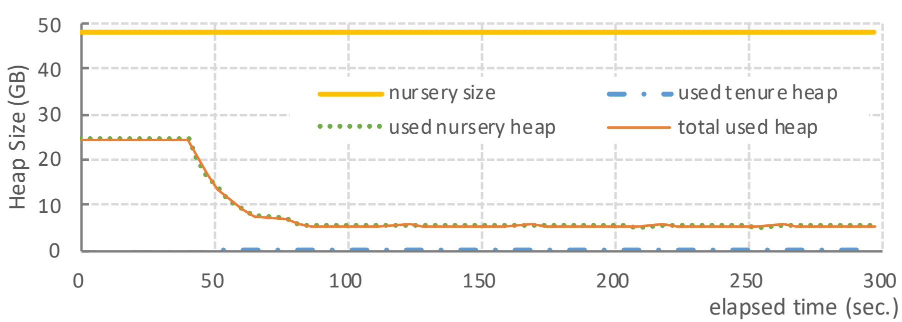
- Shuffle-heavy queries (e.g. Q5 and Q8) : over 30% cycles are spent in GC
- Shuffle-light queries (e.g. Q1 and Q6) : low SerDes cost
- Unexpected JIT and Thread Spin Lock overheads exists in Q16 and Q21



Workload Characterization – garbage collection

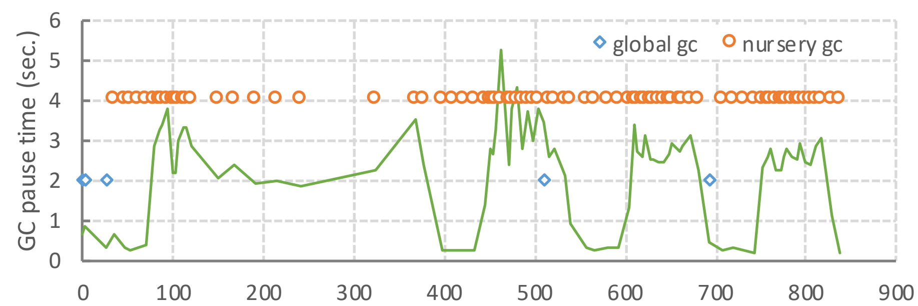
- GC
  - Many nursery GC ab
  - **Small Pause time (0.02 sec)**
  - Few global GC
- Java Heap
  - Low usage level (within nursery space)

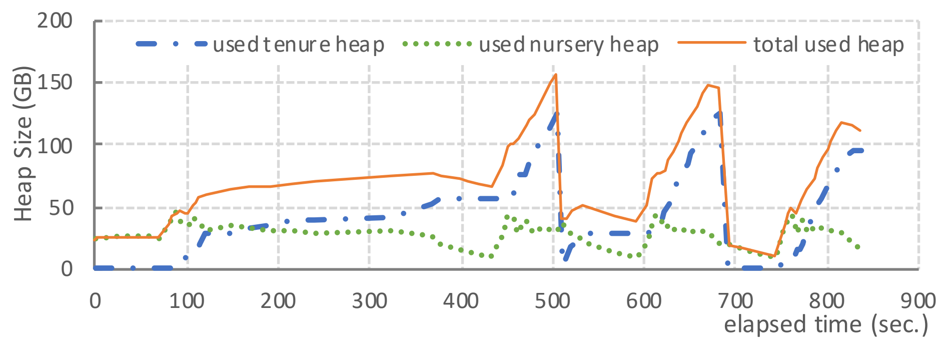




Shuffle-Light Query (Q1)

- GC
  - Many nursery GC
  - **Big pause time (3 – 5 sec)**
  - Global GC while execution
- Java Heap
  - Objects are flowed into tenure space





Shuffle-Heavy Query (Q5)

Workload Characterization – PMU profile

- Approach
  - Observed performance counters by perf
  - Categorized them based on the CPI breakdown model [*]
- Result
  - Backend stalls are quite big
  - Lots of L3 miss which comes from distant memory access
  - CPU Migration occurs frequently

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

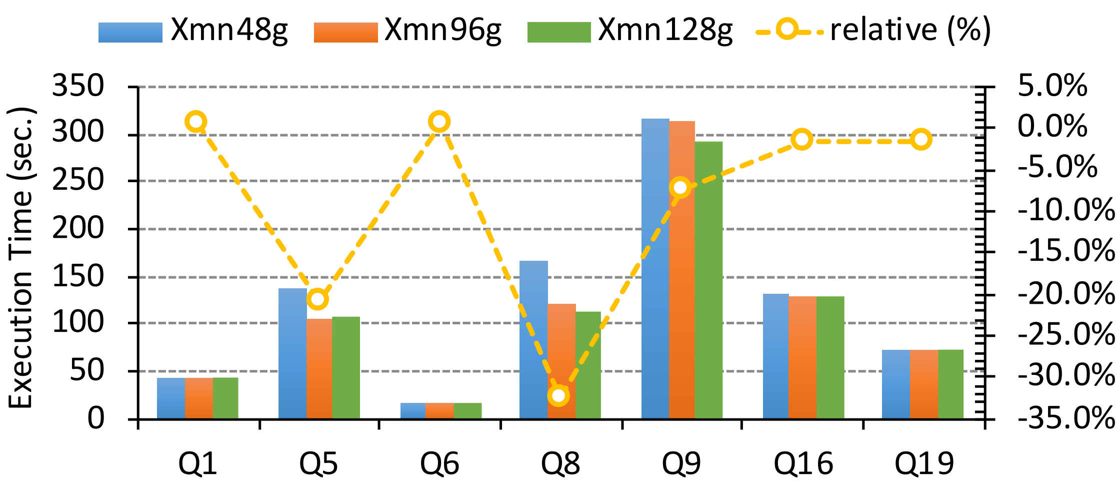
[*] https://www.ibm.com/support/knowledgecenter/linuxonibm/liaal/iplsdkcpieventspower8.htm

Problem Assessments and Optimization Strategies

- How we can reduce GC overhead?
  - 1). Heap sizing
  - 2). JVM Option tuning
  - 3). Changing # of Spark Executor JVMs
  - 4). GC algorithm tuning
- How we can reduce backend stall cycles?
  - 1). NUMA awareness
  - 2). Adding more hardware threads to Executor JVMs
  - 3). Changing SMT level (SMT2, SMT4, SMT8)

Efforts of Heap Space Sizing

- Heap sizing efforts
  - Bigger nursery space achieves up to **30% improvement**
- Small tenure space may be harmful
  - Run out of tenure space by caching RDDs in memory
  - Leaked objects from nursery space

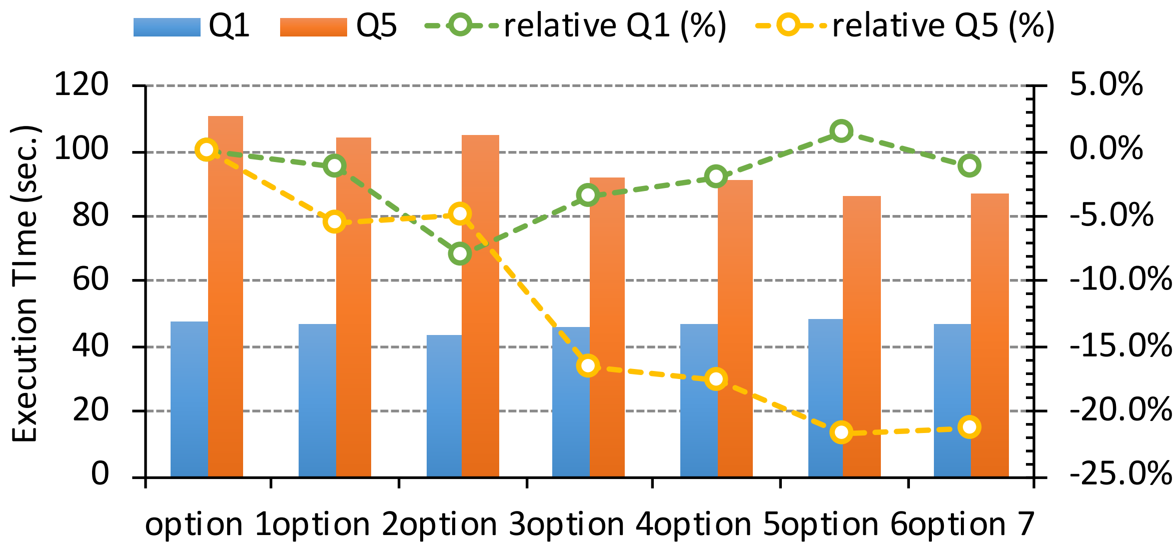


TPC-H Q9 Case

Nursery Space (-Xmn)	Execution Time (sec)	GC ratio (%)	Nursery GC Avg. pause time	Nursery GC	Global GC
48g (default)	316 s	20 %	2.1 s	39	1
96g	310 s	18 %	3.4 s	22	1
144g	292 s	14 %	3.6 s	14	0

Efforts of Other JVM Options

- JVM options
  - Monitor threads tuning
  - # of GC threads tuning
  - Java thread tuning
  - JIT tuning, etc.

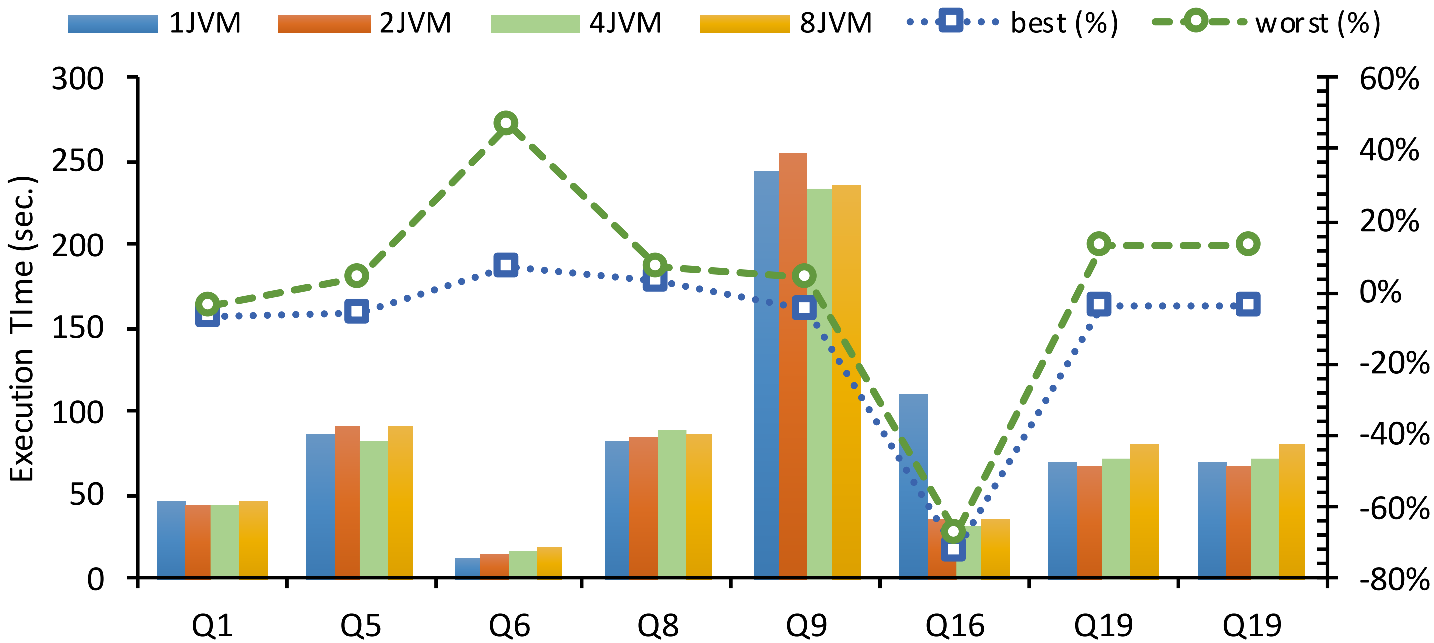


- Result
  - Improved **over 20%**

GC Threads →
Monitor Threads →
Disable Large Object Area →
Reduce thread lock cost →
Stop compaction →
Stop JIT feature →
Stop System.gc() →

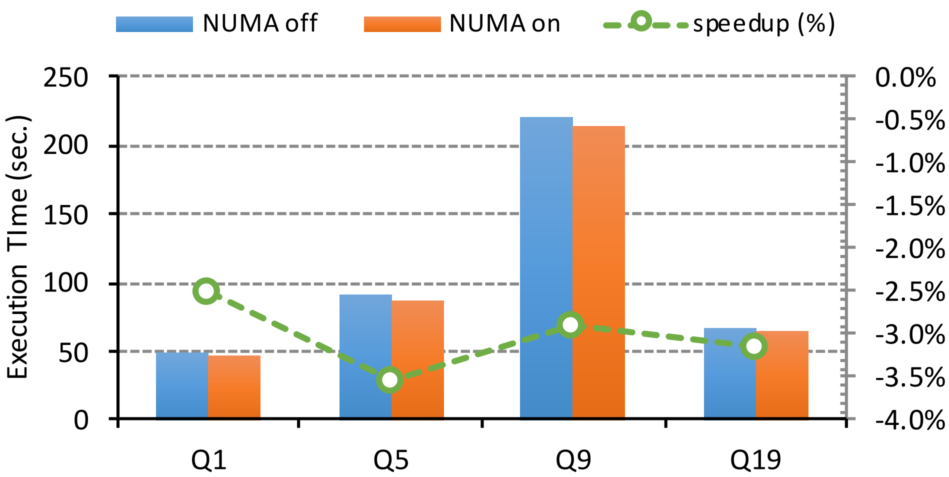
#	spark.executor.extraJavaOptions
1	**-Xgcthreads48** -Xmn96g -Xdump:system:none -Xdump:heap:none
2	**-Xtrace:none** -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
3	**-Xnoloa** -Xtrace:none -Xgcthreads48 -Xmn96g -Xdump:system:none -Xdump:heap:none
4	**-XlockReservation** -Xnoloa -Xtrace:none -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

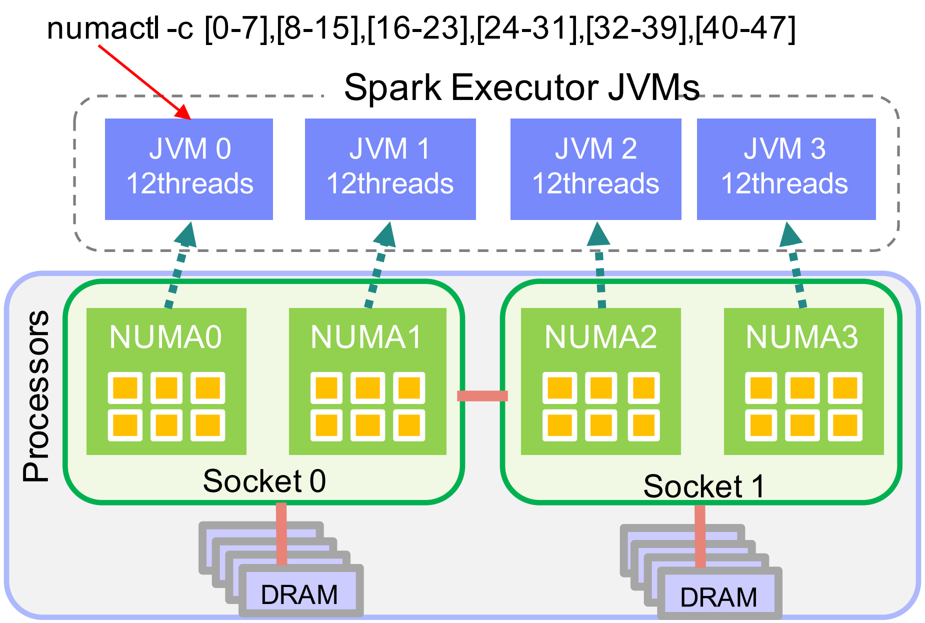
Efforts of changing JVM Counts



- Result
  - Up to 70% improvement in Q16 (by avoiding unexpected threads lock activity)
  - Reduced heavy GC overhead
  - Has a drawback a little for shuffle-light queries
  - Using 1 JVM frequently occurs task execution failure than 4 JVMs

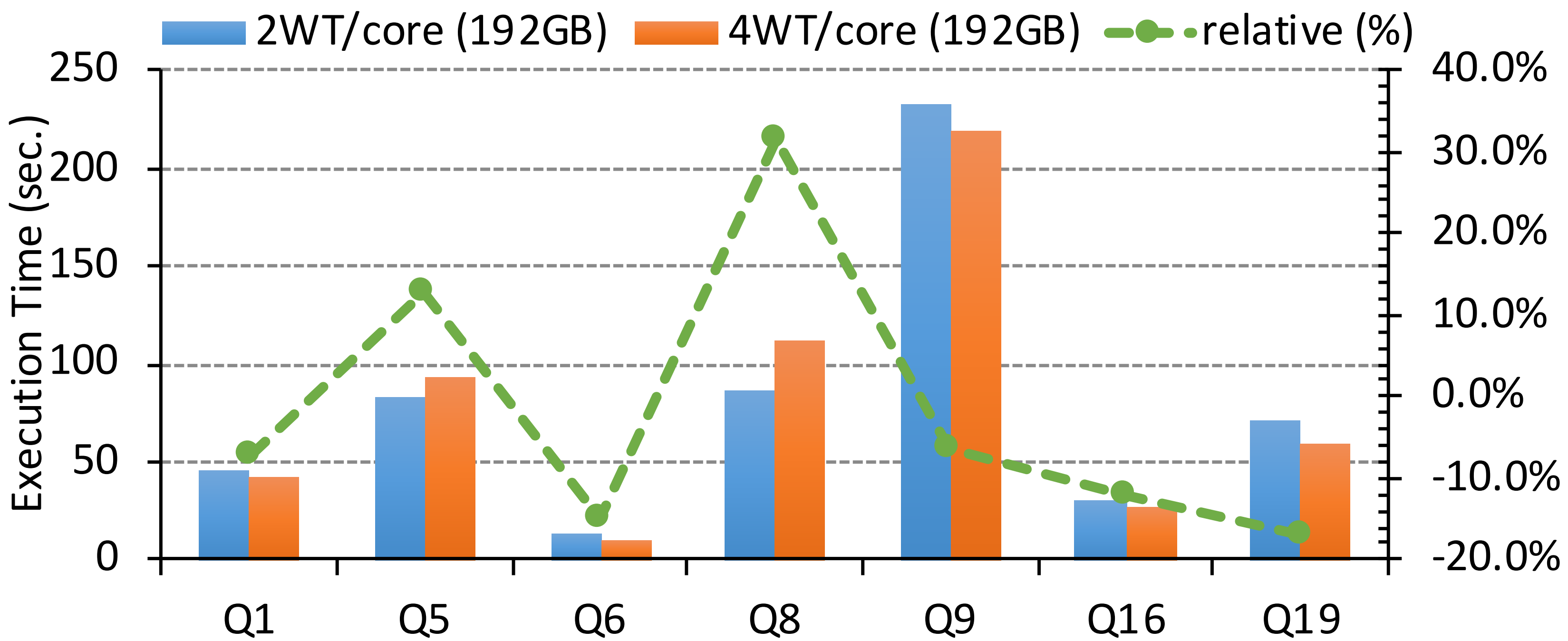
Efforts of NUMA aware process affinity





- Setting NUMA aware process affinity to each Executor JVM helps to speed-up
  – By reducing scheduling overhead
  – By reducing cache miss and stall cycles
- Result
  – Achieved **3 – 3.5 % improvement** in all benchmarks without any bad effects

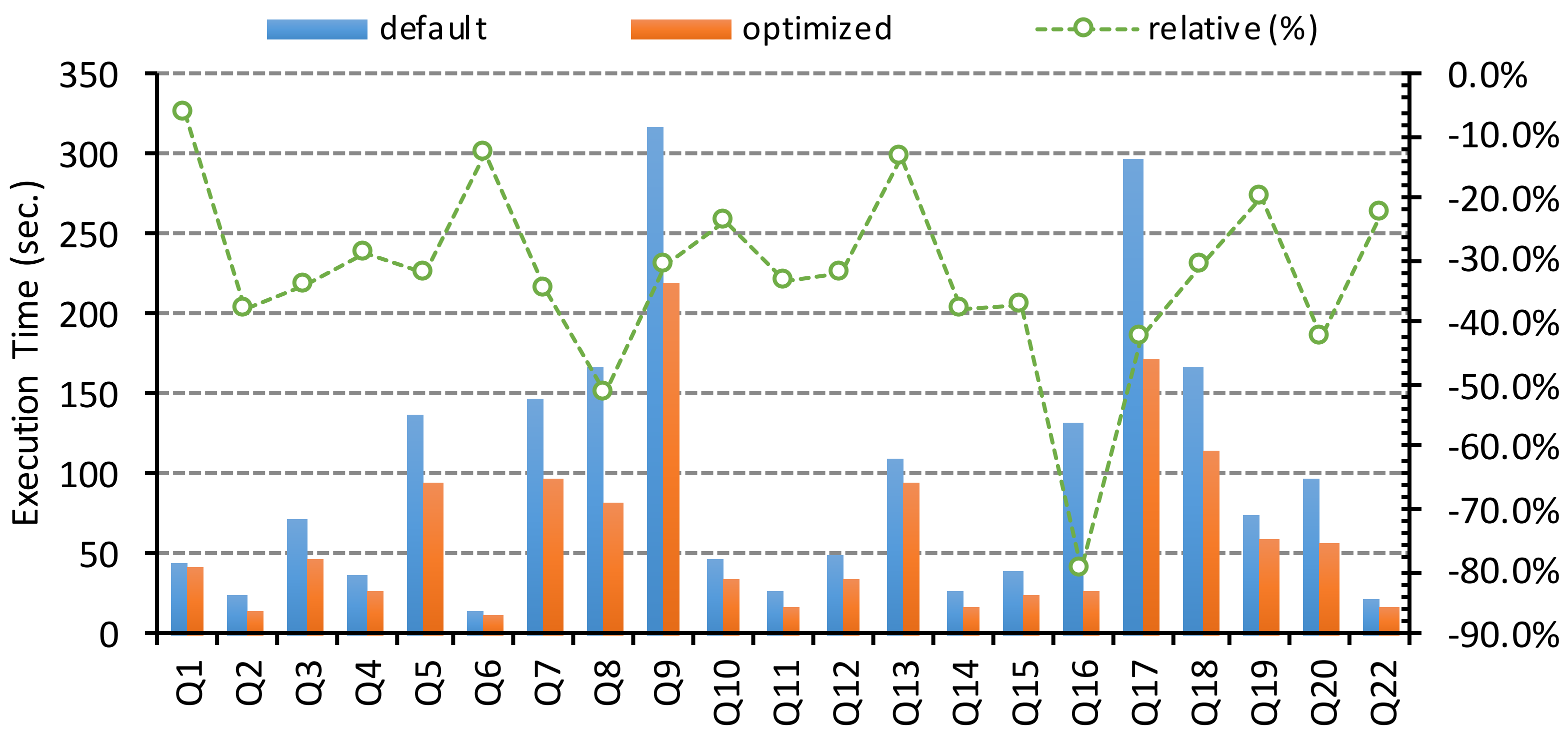
Efforts of Increasing worker threads



- Settings
  - 2WT/core : handles 12 worker threads on 6 cores (in total, 48 worker threads)
  - 4WT/core : handles 24 worker threads on 6 cores (in total, 96 worker threads)

- Result
  - Some queries gain over 10% improvement regardless of shuffle data size
  - Q5 and Q8 had a drawback

Summary of applying all optimizations



- Shuffle-light: 10 – 20% improvement
- Shuffle-heavy: 30 – 40% improvement
- Eliminated unexpected JVM behavior in Q16 and Q21
  - Q21 took 543 sec, which took over 3000 sec before tuning

Summary and Future Works

- Summary
  - Reduced GC overhead from 30% to 10% or less by heap sizing, JVM counts, and JVM options
  - Reduced distant memory access from 66.5% to 58.9% by NUMA awareness
  - In summary, achieved 30 – 40 % improvement on average
  - All experiment codes are available at https://github.com/tatsuhirochiba/tpch-on-spark
- Future works
  - Comparison between x86_64 and POWER8
  - Other Spark workloads