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

Tatsuhiro Chiba and Tamiya Onodera
IBM Research - Tokyo
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
    → Who 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

```
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)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Output</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>select c_count, count(1) as custdist from ( select c_custkey, count(o_orderkey) as c_count from ... Spark JDBC Server Query</td>
<td>2015/06/24 04:35:34</td>
<td>0.4 s</td>
<td>200/200</td>
<td></td>
<td></td>
<td>693.4 KB</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>select c_count, count(1) as custdist from ( select c_custkey, count(o_orderkey) as c_count from ... Spark JDBC Server Query</td>
<td>2015/06/24 04:34:47</td>
<td>47 s</td>
<td>200/200</td>
<td></td>
<td></td>
<td>15.0 GB</td>
<td>693.5 KB</td>
</tr>
<tr>
<td>1</td>
<td>select c_count, count(1) as custdist from ( select c_custkey, count(o_orderkey) as c_count from ... Spark JDBC Server Query</td>
<td>2015/06/24 04:31:48</td>
<td>3.0 min</td>
<td>1348/1348</td>
<td>33.3 GB</td>
<td></td>
<td>14.2 GB</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>select c_count, count(1) as custdist from ( select c_custkey, count(o_orderkey) as c_count from ... Spark JDBC Server Query</td>
<td>2015/06/24 04:31:48</td>
<td>12 s</td>
<td>185/185</td>
<td>400.1 MB</td>
<td></td>
<td>817.6 MB</td>
<td></td>
</tr>
</tbody>
</table>
### Machine & Software Spec and Spark Settings

<table>
<thead>
<tr>
<th>Processor</th>
<th># Core</th>
<th>SMT</th>
<th>Memory</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER8 3.30 GHz * 2</td>
<td>24 cores (2 sockets * 12 cores)</td>
<td>8</td>
<td>1TB</td>
<td>Ubuntu 14.10 (kernel 3.16.0-31)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>software</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark</td>
<td>1.4.1</td>
</tr>
<tr>
<td>Hadoop (HDFS)</td>
<td>2.6.0</td>
</tr>
<tr>
<td>Java</td>
<td>1.8.0 (IBM J9 VM SR1 FP10)</td>
</tr>
<tr>
<td>Scala</td>
<td>2.10.4</td>
</tr>
</tbody>
</table>

- **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

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

* Picked up several queries

- **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)

- GC
  - Many nursery GC
  - Big pause time (3 – 5 sec)
  - Global GC while execution

- Java Heap
  - Objects are flowed into tenure space

Shuffle-Light Query (Q1)

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

---

<table>
<thead>
<tr>
<th>counters</th>
<th>Q1</th>
<th>Q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cycles</td>
<td>$6.8 \times 10^{12}$</td>
<td>$2.2 \times 10^{13}$</td>
</tr>
<tr>
<td>stalled-cycles-frontend</td>
<td>$2.1 \times 10^{11}$ (3.20%)</td>
<td>$6.2 \times 10^{11}$ (2.76%)</td>
</tr>
<tr>
<td>stalled-cycles-backend</td>
<td>$3.3 \times 10^{12}$ (49.0%)</td>
<td>$1.3 \times 10^{13}$ (59.1%)</td>
</tr>
<tr>
<td>instructions</td>
<td>$7.0 \times 10^{12}$</td>
<td>$1.5 \times 10^{13}$</td>
</tr>
<tr>
<td>IPC</td>
<td>1.03</td>
<td>0.67</td>
</tr>
<tr>
<td>context-switches</td>
<td>407K</td>
<td>440K</td>
</tr>
<tr>
<td>cpu-migrations</td>
<td>11K</td>
<td>26K</td>
</tr>
<tr>
<td>page-faults</td>
<td>308K</td>
<td>1045K</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Nursery Space (-Xmn)</th>
<th>Execution Time (sec)</th>
<th>GC ratio (%)</th>
<th>Nursery GC Avg. pause time</th>
<th>Nursery GC</th>
<th>Global GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>48g (default)</td>
<td>316 s</td>
<td>20 %</td>
<td>2.1 s</td>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>96g</td>
<td>310 s</td>
<td>18 %</td>
<td>3.4 s</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>144g</td>
<td>292 s</td>
<td>14 %</td>
<td>3.6 s</td>
<td>14</td>
<td>0</td>
</tr>
</tbody>
</table>
Efforts of Other JVM Options

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

- Result
  - Improved over 20%
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