Vol.2017-OS-141 No.18
2017/7/27

SQL on Hadoop における実行エンジン及びJVMの適応的選択を用いたクエリの高速化に向けて

千葉 立寛[1,a)]　吉村 剛[1]　堀江 倫大[1]　堀井 洋[1]

概要：近年，Spark SQL や Hive/Tez，Impala，Presto など様々な SQL on Hadoop システムが登場しており，Hadoop エコシステム上に保存されているデータに対してクエリ処理を行いたい場合，ユーザは自由に実行エンジンを選ぶことが可能であるが，実行したいクエリの実行に最も適した実行エンジンが何であるかを判断することは容易ではない．さらに，これらの実行エンジンの多くは JVM 上で動作し，ユーザは OpenJDK や IBM J9 JVM などの JVM 実装を必要に応じて切り替えることも可能ではあるが，JIT コンパイラ や GC アルゴリズムを始めとするそれぞれの JVM が持つ特性によってシステム上で実際に実行されるコードの性質も変化して性能に影響を与えるため，JVM に関しても最適な選択を行うことは難しい．本論文では，TPC-DS をベンチマークとして用い，Spark SQL および Hive/Tez の各フレームワークにおいて，OpenJDK と IBM J9 VM の 2 つの異なる JVM 実装を用いた場合の性能特性と TPC-DS のクエリの特性とを比較・分析した結果を示す．

1. はじめに

近年，Hadoop エコシステムを中心とする大量に蓄積されたデータを用いたビックデータ処理にますます注目が集まっている．ETL，SQL，機械学習，グラフ処理，ディープラーニングなど分散処理フレームワーク上では多種多様なアプリケーションが実行されるようになるにつれて，分散処理フレームワーク自体も多様化している．ユーザーは必要に応じて自由に分散処理システムを組み合わせたり選択することが出来るようになった反面，実行したいアプリケーションにとってベストなフレームワークを選択することは難しくなってきている．とりわけ，Hadoop エコシステム上での SQL 実行に対するニーズは依然として高く，SQL on Hadoop システムは最も多くのフレームワークが存在するエリアの一つである [1]．Apache Hive をはじめとして，Spark SQL[2]，Impara[3], Presto[4], Drill[5] などといったシステムが利用可能であるが，仮に他のフレームワークを用いたほうが最適であったとしても，複数のバックエンドを用途に応じて使い分けることはシステム全体が複雑になるため，一つの分散処理フレームワークを使い続けることになるのが現状である．

また，Hadoop エコシステムを構成するシステムの多くは Java や Scala を用いて記述されており，Java 仮想マシン (以下，JVM) 上で動作する．現在，リファレンス実装である OpenJDK をはじめ，Oracle JDK，IBM J9 VM といった様々な JVM をランタイムとして選択可能である．ヒープのアロケーション，ガーベージコレクションのアルゴリズム，JIT コンパイラの性能および生成されるアセンブリコードの性能など，JVM ごとに異なった様々な最適化を経てアプリケーションは実行されるが，JVM の特性によっては実行されるアプリケーションの性能に差が生じるため，どちらか一方の JVM が常に最適であるということもなく，適応的に JVM を選択出来る環境が整えば，よりアプリケーション実行が最適化される可能性がある．

従来よりも早いソフトウェアサイクルで，かつ，オープンソースとして分散処理フレームワークおよび JVM ともに開発が推し進められる時代においてアプリケーションの性能を追求するためには，柔軟にそのアプリケーションの実行に最適なランタイムを選択していく枠組みが必要であり，その前提として，アプリケーションおよび複数の実行ランタイムの特性が性能にどのような影響を及ぼすのかを分析する必要がある．

本稿では, TPC-DS をベンチマークとして, 代表的な SQL on Hadoop システムである Apache Hive および SparkSQL の各フレームワークにおいて，2 つの JVM 実装である OpenJDK，IBM J9 VM を用いてクエリ処理を実行したときの実行時間性能を示す．さらに，それらの結果を元に，クエリの特徴や各実行エンジンで生成される DAG，JVM

[1] 日本アイ・ビー・エム（株）　東京基礎研究所
IBM Research - Tokyo
[a)] chiba@jp.ibm.com

の特性をプロファイルしてどのような場合にどの組み合わせが最適な結果となるのかを分析する．

2. 背景

2.1 SparkSQL/Spark

Apache Spark[6] は，インメモリ指向の分散処理フレームワークであり，大規模データ向け分散処理基盤として現在広く使われている．Spark 上での SQL を実行するための分散クエリエンジンとして，SparkSQL[2] は最も開発が盛んに行われているプロジェクトの一つである．Spark では，Catalyst 呼ばれるコンポーネントでデータの統計情報やルールベースに基づく実行プランの最適化を行う．さらに，共通に実行できる処理に関しては whole-stage code generation により，そのステージを単一の Java Method として実行できる Java コードの生成および Janino コンパイラによってクラスが生成される．その結果，無駄な仮想関数呼び出しや JIT コンパイラによる SIMD 化の恩恵を受けられるようになっている．また，SparkSQL はカラムナフォーマットとして Parquet に最適化されており，無駄なインプットファイルの読み込みを抑制するようになっている．

2.2 Hive/Tez

Apache Hive[7] は，最も広く使われている Hadoop 上で SQL ベースのクエリを実行するクエリエンジンである．元々，HiveQL で記述した ETL のバッチ処理を容易に MapReduce エンジン上で実行するために開発されたが，OLAP 用途でも利用されるようになり，よりレイテンシを重視した機能が取り込まれている．バックエンドとして MapReduce，Spark，Tez[8] が現在選択可能であるが，Tez での実行により最適化されて開発されている．Tez では，データ処理のフローを DAG で記述し，より汎用的な MapReduce 処理としてタスクを実行するたもの分散処理フレームワークであり，無駄な中間ファイルを生成しないなど，従来の MapReduce と比べて効率よく処理可能となっている．Hive/Tez においても，Parquet，RC，ORC カラムナフォーマットが利用可能であるが，実装上は ORC に最適化されている．

近年，Hive においてもインタラクティブに数秒のオーダーでクエリ実行が可能となる機能として，Low Latency Analytical Processing(LLAP) が開発されている．LLAP は，デーモンプロセスとして常駐し，インメモリ上にカラムナテーブルをプリフェッチおよびキャッシュをしたり，リソース効率を高めるためアプリケーションスレッドと I/O スレッドが非同期に動作したり，JIT での SIMD 化およびインライン最適化効きやすいオペレータパイプラインなどを提供することでインタラクティブクエリを実現する．

2.3 JVM

OpenJDK は，Java のリファレンス実装としてオープンソースで開発されており，多くのアーキテクチャで動作する JVM である．従来の I/O インテンシブから CPU インテンシブにシフトしてきた現在の SQL on Hadoop システムにおいては，JIT コンパイラで生成されるコードの性能が直接クエリ処理の性能に影響するため，JIT コンパイラにとって最適化しやすいコードを生成・定義することは重要である．OpenJDK での JIT(hotspot) では，階層的コンパイラをサポートする C1 および C2 コンパイラや，Array copy などの Native 実装のアセンブリコードを定義する Intrinsics によってより高速な実行コードを生成する．

一方，主にエンタープライズ向けの JVM 実装としては，IBM J9 VM（以下，J9) が広く使われている．J9 での JIT(TR JIT)[9] では，5 階層からなる階層的コンパイラや，システムプロファイル情報を用いた最適化，AOT コンパイラなどによってアーキテクチャに最適なコードの生成を行う．J9 自体は現在オープンソースではないが，オープンソースのマネージドランタイムを作成するツールキットである Eclipse OMR[*1] をベースにした OpenJ9 がリリースされる予定である．

3. 性能評価

3.1 実験環境

本稿では，TPC-DS の性能測定を 3.3 GHz の POWER8 プロセッサ (12 コア) を 2 ソケット搭載した 1 ノード上で実験を行った．コアそれぞれで，L1 Cache 64KB，L2 Cache 512KB，L3 Cache 8MB のキャッシュを備え，システム全体として 1TB のメモリを有し，また，HDFS や中間ファイルを書き出すための領域として SAN で繋いだ IBM Flash System を使用した．OS は Ubuntu 16.04 (kernel 4.4.0-31) を用い，SMT8 で動作しているため，システム合計で 192 ハードウェアスレッドが動作している．ソフトウェアはそれぞれ，Spark 2.1.0，Tez 0.9.0，Hive 2.2，Hadoop 2.7.2 を用い，JVM は IBM J9 VM (1.8.0, SR4FP2, jdk8u121-b13)，OpenJDK8 (build 1.8.0_121-8u121-b13) の 2 つを用いた．データセットに関しては，Hortonworks が提供する hive-testbench[*2] を用いて，Scale Factor 500 (500GB) のデータを作成した．今回比較対象とした SparkSQL および Hive では，Parquet，ORC の両カラムナフォーマットが共にサポートされている．しかしながら，それぞれの SQL on Hadoop システムでは，一方のカラムナフォーマットでの実行に最適化されており，どちらか片方のカラムナフォーマットを用いることで片方のシステムにとって有利な条件となってしまうため，SparkSQL の評価には Parquet，Hive の評価には ORC を用いて実験を行った．さらに，それぞ

[*1] http://www.eclipse.org/omr
[*2] https://github.com/hortonworks/hive-testbench

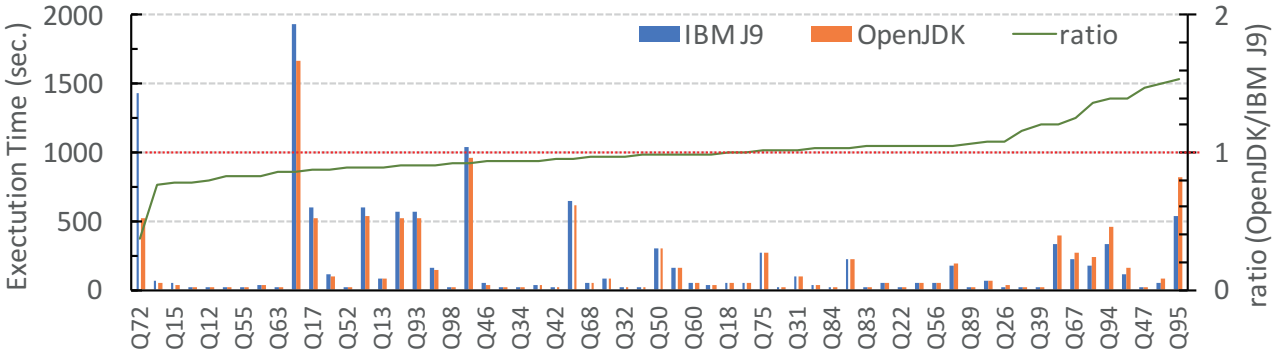


図 1 Spark における J9 と OpenJDK を用いたときの TPC-DS(SF=500) の性能比較

れのカラムナーデータを gzip(zlib) で圧縮して HDFS に保存した．クエリ関しては，同じく hive-testbench の中に用意されている TPC-DS の 68 個のクエリを用いた．

システムの設定では，実行ランタイムが Spark および Tez と異なるため，完全に同じまたは公平な設定にすることは不可能ではあるが，タスクを実行するワーカースレッドの数やメモリのサイズを揃えるようにした．SparkSQL では 12 ワーカースレッド，192GB のヒープを設定した．Hive/LLAP では 12 ワーカースレッド，12 I/O スレッド，96GB のヒープおよび 96GB の LLAP cache を設定した．今回の実験においては，Hive/LLAP では常にクエリ処理を行えるようデーモンプロセスとして立ち上がるため，同じ JVM が常に使い続けられる．Spark においても，Thrift Server を用いて Execturo JVM を起動し，同一の JVM を使い続けるようにした．その結果，両システムともに JVM 立ち上げ後のランプアップのオーバーヘッド (JIT，class loading 等）の影響が少ない状況になっている．

3.2 Spark

図 1 は，Spark 上での TPC-DS の各クエリの性能を J9 と OpenJDK のそれぞれで測定した結果である．全 68 クエリ中，6 個のクエリはエラー（クエリのフォーマット等）により実行できなかったため，62 個の実行時間を比較した．緑色のラインで示す J9 と OpenJDK の差 (ratio) が小さい順にソートされており，例えば，ratio が 1 より小さい場合，OpenJDK のほうが J9 よりも実行時間が短いことを示している．35 個のクエリにおいては OpenJDK のほうが高速に，また，27 個のクエリにおいては J9 のほうが高速にクエリ処理を実行出来た．36 個のクエリでは性能差が 10%以内に収まっているが，特徴的な差としては，Q72 においては OpenJDK が 2.7 倍ほど高速に実行できた一方で，Q95 では J9 が 1.5 倍ほど高速に実行できた．

3.3 Tez

同様に，Tez における TPC-DS の各クエリの性能を J9 と OpenJDK を用いて計測したものが図 2 である．全 68 クエリ中 3 個のクエリでエラーが発生したため，65 個の実行時間の比較を行った．35 個のクエリにおいては OpenJDK のほうが高速に，残りの 30 個のクエリにおいては J9 のほうが高速に処理が出来た．41 個のクエリにおいては性能差が 10%以内に収まっている．OpenJDK が最も高速に実行出来たのは Q87 で，その差は 1.56 倍であった．一方，J9 が最も高速に実行出来たのは Q84 で，その差は 1.74 倍であった．

3.4 Spark vs Tez

次に，OpenJDK を使った場合における Spark と Tez の両方の実行エンジンで共通にクエリが実行出来た全 60 クエリに対する性能の比較を行ったものが図 3 である．J9 に関しても細かい違いはあるが，ほぼ同様の結果のグラフとなっているので，本稿では省略している．41 個のクエリでは Tez が高速に実行できた一方で，19 個のクエリでは Spark が高速に実行出来た．同一の実行エンジンを使った場合と比べて性能差が大きくばらついており，性能差 20%以内では 8 個，20-40%では 16 個，40-60%では 12 個のクエリが該当した．また，いくつかのクエリにおいては非常に大きな実行性能差が生じた．Q66 では，Spark では 468 秒であったのに対し，Tez では 27 秒で完了しており，約 17 倍 Tez のほうが高速に実行完了できた．一方 Q22 は，Spark での実行は 3.66 秒なのに対し，Tez では 62 秒となっており，同様に約 17 倍 Spark のほうが高速に実行完了できたクエリの例である．

最後に，実行できた全 60 クエリそれぞれで，Spark，Tez，J9，OpenJDK のどの組み合わせが最も高速に実行できたかを集計したものが表 1 である．例えば，Q13 では Spark(J9) が 81.9 秒，Spark(OpenJDK) が 73.5 秒，Tez(J9) が 25.1 秒，Tez(OpenJDK) が 29.7 秒という結果であったので，Q13 では最も良い組み合わせとして Tez(J9) の組の数字に計上する．この結果，実行エンジンの観点では Tez が 31 個のクエリで，Spark が 19 個のクエリにおいて適しているという結果となった．JVM の観点では，J9 が 35 個のクエリで，OpenJDK が 15 個のクエリで高速に実行できてい

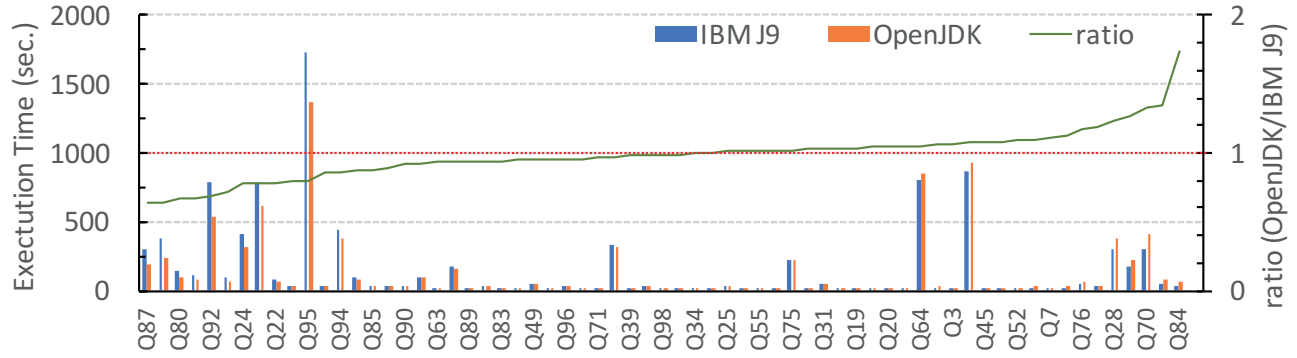


図 2 Tez における J9 と OpenJDK を用いたときの TPC-DS(SF=500) の性能比較

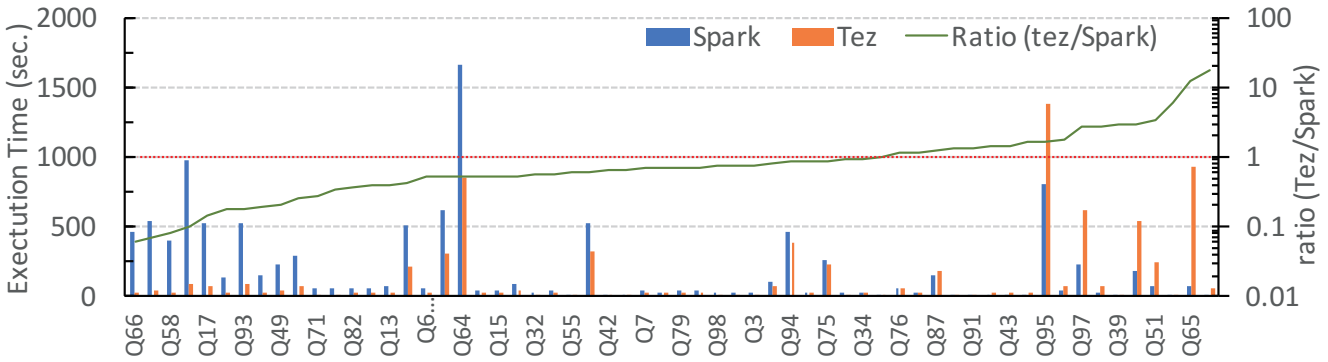


図 3 OpenJDK での Spark と Tez での TPC-DS(SF=500) の性能比較

表 1 実行エンジンと JVM での最適な組み合わせの数

	IBM J9	OpenJDK	total
Spark	13	6	19
Tez	22	19	41
total	35	25	60

る．これは，3.1 節と 3.2 節で示した結果からすると意外な結果となっている．つまり，単一の実行エンジンだけの結果を見ると OpenJDK のほうが全体の割合としては速く実行出来るクエリが多かったが，複数の実行エンジンで比較すると全体の割合として J9 のほうが速いことを示している．上に示す Q13 が良い例であるが，Spark だけで評価した場合，OpenJDK のほうが J9 よりも高速であったが，両実行エンジンを含めると Tez(J9) のほうが高速であり．OpenJDK での良さが消えている例となっている．

4. 性能解析

3 節で示した通り，実行するクエリによっては実行エンジン，JVM の違いにより大きく性能に差が出ることがあり，最適な組み合わせを一意に決定することは難しい．本節では，特徴的な性能差が出ているいくつかのクエリの性能解析を通じて，どのような場合に Spark，Tez それぞれの実行エンジンで性能が発揮できるのか，もしくはボトルネックはどこにあるのかの分析を行う．

4.1 解析手法および対象クエリ

今回，それぞれの実行エンジンが生成するクエリプラン，DAG の実行ログや各ステージごとでの統計情報を用いて，クエリの傾向 (入力データサイズ，出力サイズ，シャッフルサイズ) をベースに分類し，どの部分が最もホットスポットであるのかの分析をしていく．さらに，JVM レベルや Java メソッドレベルのどの部分で CPU が多く利用されているかを分析するため，Oprofile や Linux Perf を用いて CPU ボトルネックの部分を分析していく．また，dstat を用いたシステム全体のモニタリング結果や，jstack を用いたスレッドスタック，GC のログ，JIT のログなどを元にクエリの特徴を多角的に判断していく．

本稿では，特徴的な結果を示したいくつかのクエリに関して，実際に分析を行っていく．図 4 は，異なる実行エンジン（Spark，Tez）および異なる JVM 実装（OpenJDK，J9）のそれぞれの組み合わせで実行したときの，4 つのクエリ (Q50，Q51，Q58，Q82) の実行時間を比較したものである．Q51 は，Spark，Tez の両方のランタイムで J9 が OpenJDK よりも 30-40%ほど速いケースであると同時に，Spark が Tez に比べて 3.5 倍ほど高速に実行出来た例である．Q50 は，Tez において OpenJDK が J9 より 35%程度速い例であり，Q82 は Spark において OpenJDK が J9 より 30%ほど高速な例である．Q50，Q58，Q82 はいずれも Spark より Tez のほうが 2-10 倍高速に実行出来た例となっ

図 4　解析対象クエリの実行時間の比較

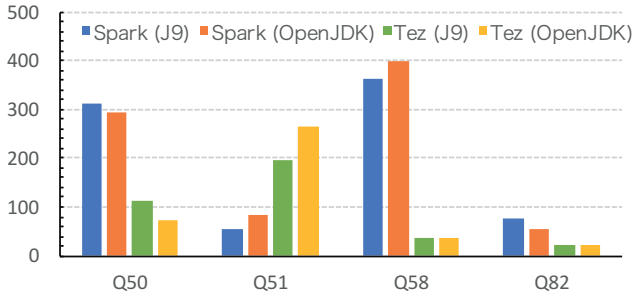


図 5　各クエリの各カテゴリにおける CPU サイクル比率

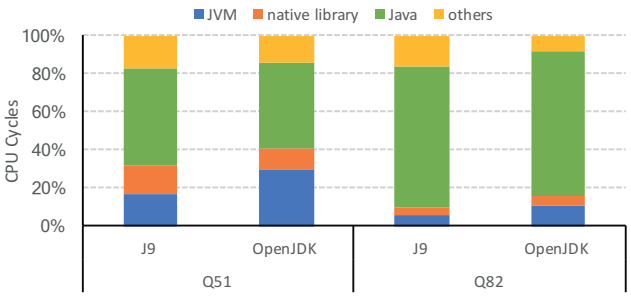


表 2　Q51 および Q82 の Spark の DAG の解析

	Q51	Q82
# of map stage	2	2
# of reduce stage	6	2
# of bcast hash join	2	2
# of sort-merge join	1	1
# of grpby/sort	5	1
input read size (parquet,zlib)	6.0GB	2.5GB
shuffle output size (lz4)	1.0GB	5.6GB

表 3　各クエリの Java メソッドにおける CPU サイクル比率

	Q51		Q82	
	J9	OpenJDK	J9	OpenJDK
catalyst	25.3%	13.2%	15.7%	13.0%
sql core	3.49%	3.65%	24.4%	13.5%
sql sort	0.41%	0.29%	10.3%	5.2%
crc32c	2.5%	2.0%	0.59%	0.9%
lz4	3.0%	3.6%	6.9%	14.5%
Unsafe	0.0001%	2.0%	0.0003%	2.2%

ている．

4.2　Spark

まず，Spark において J9 と OpenJDK で性能差が生じた Q51 および Q82 に着目して分析していく．Q51 は，2 つのファクトテーブル (web_sales，store_sales) と 1 つのディメンションテーブル (date_dim) を用い，ファクトテーブルとディメンションテーブルをそれぞれ Join したサブクエリの結果に対して，さらに Join を行っていくなど，非常にサブクエリが多用されており，最終的な Spark 上での Stage としては，2 つの map stage と 6 つの reduce stage で構成される．Q82 は，2 つのファクトテーブル (store_sales，inventory) と 2 つのディメンションテーブル (item, date_dim) を単純に内部結合するクエリであり，2 つの map stage と 2 つの reduce stage で構成される．表 2 に，Q51，Q82 の特徴をまとめた．Q51 は，入力データが大きく比較的 Shuffle のデータ量は少ないが，Shuffle の段数が多いクエリである．一方 Q82 は，入力データが少ないながら Shuffle のデータ量が多く，Shuffle の段数が少ないクエリである．

また，Oprofile を用いてクエリ実行中の Spark Executor JVM の CPU Cycle を調べたものが図 5 である．この図から，Q51 では JVM(MM，GC，JIT，Thread，Monitor，etc.) および Native Library (zlib，libzip，lz4) 起因するタスクが Q82 と比べて多くなっている．さらに，Java メソッドとして処理されている部分をブレイクダウンし，着目すべきポイントをまとめた結果を表 3 に示す．catalyst は Whole Stage Codegen として生成された Generated Code での処理，sql core はシリアライズ・デシリアライズ・および Parquet ファイルのロード処理，sort は sort-merge-join 処理，crc32c はインプットデータロード時の処理，lz4 はシャッフルデータの compress/decompress 処理，Unsafe は sun.misc.unsafe.copyMemory を用いたメモリコピー処理である．シャッフルデータ量が多い Q82 に関しては，ソート・データ圧縮・シリアライズに関連する処理が大きい．一方，インプットデータが多い Q51 に関しては，crc32c の計算コストが Q82 よりも高くなっている．

J9 と OpenJDK の違いの一つに，JNI 経由での Native 実装の呼び出しオーバーヘッドがある．とりわけ Spark で多用される Unsafe.copyMemory に関しては，OpenJDK 側に Unsafe.copyMemory の Intrinsic が用意されていないため，シャッフルのデータサイズが小さい状況でシャッフル処理の段数が多い Q51 に関しては特に JNI 呼び出しオーバーヘッドが顕著となる．また，zlib や lz4 のネイティブ実装を呼び出す際，複数スレッドが同時にインスタンスを取得出来ないようにするため，synchronized メソッドによってガードされており，結果としてアプリケーションスレッドがブロックしている状況が OpenJDK で見受けられた．

一方 Q82 に関しては，約 2.5GB のデータを読み込み 5.6GB のシャッフル用データを書き出す ShuffleMapTask の中で，シリアライゼーションと圧縮を行いつつ DataOutputStream を用いてファイルにデータを書き出す部分が J9 では OpenJDK に比べて重い処理となっており，とりわけシリアライゼーションのコストが高いことが分かった．

以上の結果から，stage 数が多いクエリに関しては J9 のほうが高速であり，stage 数が少ないシンプルなジョブかつ shuffle のデータを大量に書き出すクエリに関しては OpenJDK のほうが高速でった．この結果は，他の Q50 および Q58 にも当てはまる傾向であった．

表 4　Tez における Q50，Q51 の CPU サイクル比率

	Q50		Q51	
	J9	OpenJDK	J9	OpenJDK
kallsyms	18.4%	6.5%	4.1%	3.3%
jvm	9.1%	25.9%	11.2%	7.0%
java	68.6%	61.0%	81.5%	87.0%
tez	3.1%	2.6%	28.9%	12.0%
orc	11.9%	17.0%	0.8%	0.3%
java.io	1.0%	4.0%	6.0%	18.3%
hive.ql	38.0%	30.7%	16.0%	9.3%
serDe	2.3%	1.3%	12.4%	29.0%

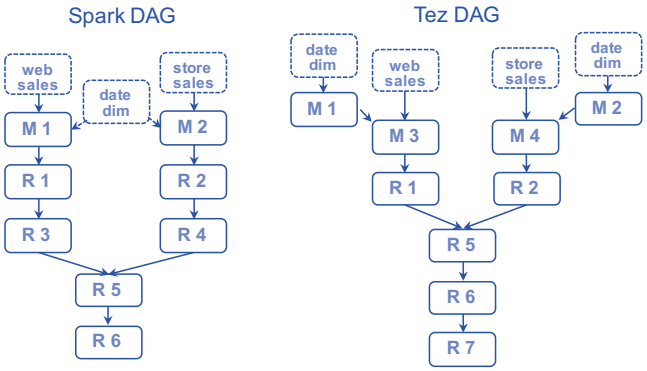
4.3　Tez

次に，Tez における J9 と OpenJDK の性能差が生じたケースとして，Q50 と Q51 を検証する．Q51 は Spark での結果と同様，J9 が OpenJDK より高速であったケースであり，Q50 は OpenJDK が J9 より高速であったケースである．表 4 は，Q50 および Q51 での Oprofile を用いたプロファイル結果の一部をまとめたものである．上段の java レイヤに該当する部分をブレイクダウンしてまとめたものが下段となり，tez は shuffle の中間ファイルのリード/ライトや PipelineSort が主な処理で，orc はインメモリの ORC フォーマットのデータのリード処理，java.io は主に DataOutputStream や DataInputStream を使った処理，hive.ql はベクタライズされた Aggregate や Join の処理，serDe はシリアライズ・デシリアライズ処理である．

両方のクエリにおいて，LLAP 上でのインメモリ ORC フォーマットデータを各 MapTask が読み込んでいるが，全ての場合において J9 のほうが OpenJDK よりも高速であった．例えば，Q51 において web_sales を読み込む MapTask において各タスクの平均実行時間が J9 は 7 秒，OpenJDK では 21 秒と 3 倍の差が生じていた．このタスク内でインメモリからデータをコピーした後，集約処理などを経て中間ファイルの書き出し・シリアライズまで行うが，OpenJDK ではシリアライズのコスト (serDe，java.io) が非常に大きく，この部分の差が最終的な差となっている．

一方 Q50 では，MapTask は上記の通り J9 のほうが高速であるが，ReduceTask では OpenJDK のほうが高速に実行出来た．一番重い ReduceTask において，J9 はトータルで 106 秒，OpenJDK は 60 秒で実行している．表 4 で示したプロファイル結果において，J9 における kallsyms の比率の高さが際立っており，実際，dstat で確認したときの system cpu の利用率が 3 倍ほど多く，user cpu（アプリケーションスレッド）は OpenJDK に比べて半分ほどしか使えていない状況であった．Tez では，Spark と比べて非常に多くのスレッドを一つの JVM 内で消費しており，Tez ランタイムのシャッフル時のマージおよびソート・中間ファイル書き出し・中間ファイルの Fetcher など様々なスレッドが各 Vertex 単位で存在しており，これはシステムの CPU コア数に応じて生成されている．さらに，コンテキストスイッチが 7 倍ほど多く J9 では発生しており，システム側に起因する性能低下であったと考えられる．

図 6　Spark および Tez で実行された Q51 の DAG



4.4　Spark vs Tez

最後に，Spark と Tez それぞれで性能の優劣がついたクエリに対して解析していく．図 4 で示したように，Q51 は Spark が Tez よりも速かった例であり，その他の Q50，Q58，Q82 に関しては Tez のほうが速かった例である．図 6 は，Q51 における Spark と Tez での生成された DAG を模式的に表したものである．M は MapTask，R は ReduceTask を表し，どのテーブルがどの MapTask でロードされたかが示されている．date_dim テーブルは小さいため，Spark，Tez ともに Broadcast Hash Join が使われているが，Spark ではドライバプログラムが直接 M1，M2 に送信する一方，Tez では，Executor 側でロードした後，それぞれの MapTask 内で処理される．Spark における R3，R4，R5 と Tez における R1，R2，R5 はソートマージジョインが実行されており，両方の DAG ともほぼ同様の形となっている．両方のクエリ実行プランを比較したところ，date_dim に対するフィルタの条件が，Spark のほうがより最適化されていた．最終的には Spark では 366 行分，Tez では 8116 行分のデータとなってブロードキャストされており，その後に続くシャッフルの中間ファイルの書き出しが Spark が 687MB であったのに対して，Tez では 3.6GB であった．そのため，その後に続く ReduceTask もデータサイズに比例して増加しており，結果的には全体の実行時間として 3 倍程度の差となっていた．

一方，Q50，Q58，Q82 の 3 つのクエリに関しては，Hive/LLAP の機能であるブルームフィルタによるフィルタが効果的に機能しており，ジョイン時の行数の削減に大きく貢献したことが Spark に比べて Tez での実行が高速になっている要因であった．

4.5 性能解析結果の検証

それぞれでの性能解析の結果をまとめると以下の通りになる．Spark においては，JOIN が大量に発生して STAGE 数が多くなる複雑なクエリにおいては J9 のほうが性能が良く，JOIN がなく Stage 数が少ないシンプルなクエリにおいては OpenJDK のほうが性能が良い結果であった．図 1 において，J9 で性能が出ているクエリの多く (Q87，Q94，Q95，etc.) は，OUTER JOIN や多段のサブクエリを含んでおり，傾向と一致した．同様に，OpenJDK で性能が出ているクエリ (Q15，Q43，Q73，etc.) の多くは，Groupby や Orderby だけで実行時間も 10-30 秒程度であり，同じく傾向と一致した．

一方 Tez においては，Spark での結果とは異なり，インメモリカラムナのリード性能の差により，単純なクエリまたはインプットファイルを大量にリードするクエリでは J9 のほうが性能が良く，段数が多くなるクエリに関しては OpenJDK のほうが性能が良い結果であった．図 2 において，Q70，Q84 といった J9 で性能が出ているクエリはインプットが支配的で傾向と一致した．同様に Q80，Q95 といった OpenJDK で性能が出ているクエリは 20-40 段の DAG が構成されており，上記傾向と一致していた．

Spark と Tez での性能比較では，CBO による最適化や各 Vertex 内で実行されるブルームフィルタがどれだけ適用されたかによって，インプットおよびシャッフルのデータ量が削減されており，この差が直接的な性能差の多くを占めていた．

5. 関連研究

SQL on Hadoop システムそれ自体のランタイムの性能解析および JVM のボトルネック解析が様々行われている．文献 [1] では，TPC-H および TPC-DS を実行したときの Hive/Tez および Impala での性能やデータフォーマット (ORC，Parquet) の違いについて詳細な解析が行われているが，現在の Hive/Tez では，CPU 効率をより意識したベクタライゼーションやパイプラインソート，および LLAP の機能が追加されているため，最新の実装での解析結果とは大きく異なる．また，Spark に関しての評価は含まれていない．文献 [10] では，TPC-H の性能を Spark 上で評価しているが，J9 のみでの評価であり OpenJDK に関する評価はされていない．

CPU キャッシュの局所性や SIMD 命令を用いて DBMS や SQL-on-Hadoop システムにおけるクエリ処理の効率化に関する様々な研究がなされており [11][12]，クエリオプティマイザの中で最適な実行コードを生成するアプローチも多く存在する．例えば，Impala[3] では LLVM を用いた JIT で実行アセンブリを CPU に最適化している．JVM 上で動作する Spark や Hive/Tez においても Java の JIT コンパイラによる実行コードの SIMD 化などが進められている [2]．複数の異なる JVM 実装において，JIT コンパイラの性能や特性も異なるため，JVM の観点から最適なコード生成をするための手法を検討するこは重要である．

アプリケーション特性によって選択した実行ランタイムでの性能が最適ではないケース柔軟に対処するため，実行時に中間表現（IR や DAG) 上で最適化する研究もいくつかなされている [13][14]．例えば，文献 [14] では実行したいアプリケーション (Frontend) と実行するランタイム (Backend) の中間に位置して IR や DAG を動的に最適化後，Backend のコードを生成するというアプローチが取られている．本稿で解決したい問題と共通する部分は多いが，メタなアプローチで最適化するという点で異なる．

6. おわりに

本稿では，Hive/Tez および SparkSQL/Spark の各実行エンジンにおいて，2 つの JVM 実装である OpenJDK，J9 を用いて TPC-DS ベンチマークを実行したときのクエリのレスポンスタイムの比較および性能が異なるクエリでの Spark および Tez の各ランタイムの比較，J9 および OpenJDK の各 JVM の性能特性の分析を行った．TPC-DS の 68 種類のクエリの全体を通してそれぞれの組み合わせで実行時間を計測した結果，クエリの種類によって Spark，Tez，OpenJDK，J9 ともに高性能となる組み合わせが存在することが分かった．Spark においては，複雑なクエリでは J9 が，単純なクエリでは OpenJDK が優位となる傾向があった．Tez においては，複雑なクエリでは OpenJDK が，単純なクエリでは J9 が優位となる傾向があった．Spark および Tez での性能差は，実装上の問題および CBO による実行プランの最適化の性能に起因することが多いことが分かった．

今回の解析結果や傾向は，一部のクエリから導かれたランタイムや JVM 特性の一側面であり全てを表すものではない．実行したいクエリにとって最も適切な組み合わせを選択するメタスケジューラを作るためには，さらに深い分析が必要となる．今後の課題としては，残りのクエリやシステムの設定を変えたときの性能特性を分析をしたり，性能モデルの学習をしたりすることを検討しており，それらを YARN または Thrift Server 上に実装してシステム全体のスループットが向上することを確認したいと考えている．さらに，文献 [15] のような ramp up 済みの JVM をプールするようなメカニズムと組み合わせることで，適応的に JVM を選択できるようにする仕組みを検討する．また，それぞれのランタイム・JVM で特徴的なホットスポットとなっていた部分に関しては，別途改善していくことを考えている．

参考文献

[1] Floratou, A., Minhas, U. F. and Özcan, F.: SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures, *Proc. VLDB Endow.*, Vol. 7, No. 12, pp. 1295–1306 (online), DOI: 10.14778/2732977.2733002 (2014).

[2] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A. and Zaharia, M.: Spark SQL: Relational Data Processing in Spark, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, ACM, pp. 1383–1394 (online), DOI: 10.1145/2723372.2742797 (2015).

[3] Kornacker, M., Behm, A. and Bittorf, V. e. a.: Impala: A Modern, Open-Source SQL Engine for Hadoop, *Proceedings of 7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*, Asilomar, California, USA (2015).

[4] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A. and Schreiber, R. S.: Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices, *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, ACM, pp. 197–210 (2013).

[5] Hausenblas, M. and Nadeau, J.: Apache drill: Interactive Ad-hoc Analysis at Scale, *Big Data*, Vol. 1, No. 2, pp. 100–104 (online), DOI: 10.1089/big.2013.0011 (2013).

[6] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*, USENIX Association, pp. 10–10 (2010).

[7] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H. and Murthy, R.: Hive - a petabyte scale data warehouse using Hadoop, *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pp. 996–1005 (online), DOI: 10.1109/ICDE.2010.5447738 (2010).

[8] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A. and Curino, C.: Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, ACM, pp. 1357–1369 (online), DOI: 10.1145/2723372.2742790 (2015).

[9] Sundaresan, V., Maier, D., Ramarao, P. and Stoodley, M.: Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler, *International Symposium on Code Generation and Optimization (CGO'06)* (2006).

[10] Chiba, T. and Onodera, T.: Workload characterization and optimization of TPC-H queries on Apache Spark, *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 112–121 (2016).

[11] Zukowski, M., van de Wiel, M. and Boncz, P.: Vectorwise: A Vectorized Analytical DBMS, *2012 IEEE 28th International Conference on Data Engineering*, pp. 1349–1350 (online), DOI: 10.1109/ICDE.2012.148 (2012).

[12] Costea, A., Ionescu, A., Raducanu, B., Switakowski, M., Barca, C., Sompolski, J., Luszczak, A., Szafranski, M., de Nijs, G. and Boncz, P.: VectorH: Taking SQL-on-Hadoop to the Next Level, *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, ACM, pp. 1105–1117 (online), DOI: 10.1145/2882903.2903742 (2016).

[13] Shoumik Palkar, S., Thoms, J. J., Shanbhag, A. S., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S. and Zaharia, M.: Weld: A common runtime for high performance data analytics, *Proceedings of 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*, Chaminade, CA, USA (2017).

[14] Gog, I., Schwarzkopf, M., Crooks, N., Grosvenor, M. P., Clement, A. and Hand, S.: Musketeer: All for One, One for All in Data Processing Systems, *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, EuroSys '15, New York, NY, USA, ACM, pp. 2:1–2:16 (online), DOI: 10.1145/2741948.2741968 (2015).

[15] Lion, D., Chiu, A., Sun, H., Zhuang, X., Grcevski, N. and Yuan, D.: Don'T Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-parallel Systems, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, USENIX Association, pp. 383–400 (online), available from ⟨http://dl.acm.org/citation.cfm?id=3026877.3026907⟩ (2016).