Spark上でのTPC-H Benchmarkのマルチレイヤ最適化の評価

千葉 立寛　堀井 洋　小野寺 民也

Apache Spark は Scala で実装されたインメモリ指向の分散処理フレームワークであり，Hadoop/MapReduce よりも高速に大規模データを処理可能なシステムとして注目されている．Spark は JVM 上で動作するため，JVM の設定により，GC やロックの挙動を制御することが可能である．また，Spark では多数のスレッドが並列してデータ処理を行うため，OS の設定によりスレッドがメニーコアや SMT を効率的に利用することが可能である．さらに，Spark ではアプリケーションの書き方によりメモリやディスクの利用法が変化するため，解析するデータや Spark の特性を考慮してアプリケーションを修正することにより，メモリを有効に利用可能になる．このように，アプリケーションをより高速に実行するためには，様々なレイヤを総合的にチューニングすることが重要である．本稿では，Hive クエリ，Spark，JVM，OS レイヤごとのチューニングの効果を TPC-H Benchmark を題材として定量的に示す.

Besides being an in-memory oriented computing framework, Spark runs on top of Java Virtual Machine (JVM), so JVM parameter tuning is important for improving Spark application performance. Misconfigured parameters and settings result in performance degradation, for example, using too large Java heaps often causes long garbage collection pause time, which accounts for over 10-20% of application execution time. Moreover recent modern computing node has many cores and it supports to run multiple threads simultaneously with SMT technology. Thus it is also important to optimize in full stack. Not only optimizing JVM parameter, but also OS parameters, Spark configuration and application code itself based on CPU characteristics are needed to take full advantage of underlying computing resource. In this presentation, we will use TPC-H benchmark as our optimization case study and we will introduce our optimization results in full stack. As a result, we will provide how they contribute to improve Spark application performance.

1 はじめに

Hadoop エコシステムを中心とした大規模データ向け分散処理基盤の普及とともに，ビジネス・サイエンスの分野を問わず，様々な分野でのビックデータ処理に対する期待が高まっている．とりわけ，文献[2] で様々な機械学習アルゴリズムが MapReduce で記述可能であることが示されたことにより，機械学習やグラフ処理などの繰り返し処理を多く含むようなワークロードを効率よく処理することが近年のビックデータ処理基盤に求められる重要な要素の一つである．このような背景の中で，Apache Spark[†1] は Hadoop/MapReduce よりも高速に処理可能なシステムとして注目されているフレームワークの一つである．Spark は，できるだけ再利用可能な形式でインメモリにデータを配置し，ディスク I/O を極力発生させないようにすることによって低レイテンシな分析処理を目指したシステムであるが，Spark ランタイム上で動作する様々なコンポーネントがコミュニティにより開発されている．例えば，機械学習向けライブラリ (MLlib) や，リレーショナルクエリ処理を実現する Spark SQL，ストリーム処理向け Spark Streaming，R や Python などデータ分析で広く使われる言語への API の提供など，インタラクティブな

* This is an unrefereed paper. Copyrights belong to the Author(s).

Tatsuhiro Chiba, Hiroshi Horii, Tamiya Onodera, 日本アイ・ビー・エム株式会社 東京基礎研究所, IBM Research - Tokyo.

†1 http://spark.apache.org/

処理からストリーム処理まで，様々な大規模データを処理するための共通基盤としての利用が急速に広まってきている．

Hadoop/MapReduce では，Map・Reduce の各処理ごとにデータをディスクに書き込むため，ディスクがボトルネックになるのに対し，Spark においても入力の一次データソースとしてディスクアクセスが発生するものの，Spark では多くの場合インメモリで処理されることが多いため，CPU やメモリ，さらにはネットワークがボトルネックになることが予想される．これらのボトルネックをできるだけ回避しノードでの実行性能を最大化するためには，Spark ランタイム自体のチューニングだけではなく，アプリケーションを記述するユーザーの側もこれまで以上に CPU やメモリを意識したプログラムの記述や設定を行う必要がある．アプリケーションの実行性能を最大化するための要素として，Spark アプリケーション自体の書き方，Spark の設定の変更，JVM のチューニング，ハードウェアを意識した OS のチューニングなどが挙げられるが，どのような設定を行うことで性能が向上するかについての知見は少なく，それらのチューニングは相互に影響するため，複数のレイヤでのチューニングを総合的に判断して最適化していく必要がある．

本稿では，TPC-H Benchmark を題材として，Hive クエリ，Spark，JVM，OS レイヤごとのチューニングによってアプリケーション全体の実行性能がどの程度改善できるかを定量的に示す．まず，何もチューニングを行わない設定でベースラインの実行性能を測定し，クエリの書き方，Spark や JVM の設定の変更，アーキテクチャの NUMA 構成を考慮した OS 設定のチューニングによって，どの程度性能が改善するかを計測し比較していく．1 ノードの POWER8 マシン上で測定した結果，クエリの書き換えにより 20%程度，NUMA の設定を加えることでほぼ全てのクエリで 10%程度，Executor JVM 数を調整することで 20%程度，JVM オプションのチューニングにより 20-30%程度性能が向上した．これらのチューニングを全て合計したものとベースラインを比較すると，クエリによっては 20-40%程度性能が向上することを確認した．

2 節で背景とそれぞれのレイヤでの最適化において必要なことを列挙し，それを踏まえて，3 節で SQL クエリの書き換え，4 節で Spark オプションおよび JVM 数の最適化，5 節で NUMA を用いた最適化について論じる．6 節ではそれぞれのチューニングによって得られた性能向上について説明する．

2　背景

2.1　Apache Spark

Spark は，Scala で実装された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 Hadoop/MapReduce よりも高速に処理可能なシステムとして注目されている．HDFS などに保存された同じ大規模データに対して，従来の Hadoop では扱いにくい機械学習やグラフ解析のような繰り返し処理を多く含むようなワークロードにおいて，入力データや中間データ，結果を再利用可能な形でインメモリ上に置き，出来るだけディスク I/O を発生させないにすることで，低レイテンシな分析処理の実現を目指している．Spark 上で動作する様々なコンポーネントやライブラリの開発も活発であり，機械学習向けの MLlib ライブラリや，グラフ処理向けの GraphX，SQL クエリ処理向けの Spark SQL などがある．

Spark では，RDD (Risillient Distributed Datasets) [8] と呼ばれるイミュータブルな分散コレクションをベースに，RDD に対して様々なオペレーション（例えば map, filter, count, reducebykey など）を適用させて新たな RDD を生成することで処理を実現するプログラミングモデルである．RDD の変換フローは DAG として表現され，それぞれの RDD に定義された依存関係，すなわち，オペレーションとデータの局所性を考慮した上で，ステージに分割されて並列分散実行される．

Spark におけるメインプログラムをドライバと呼び，RDD や DAG，ステージの分割などの情報が管理されている．多数のノードが使えるような大規模な分散システムにおいて，ドライバはクラスタマネージャを介して，アプリケーションの実行に必要な計算資源を確保する．Spark では，Built-in の Standalone,

YARN，Mesos の 3 つのクラスタマネージャが利用可能であるが，本稿では Standalone モードでの実行を前提とする．RDD 内の分割されたデータに対する処理の単位はタスクと呼ばれ，クラスタマネージャによって確保された Executor JVM 上で実行される．Executor JVM に用意された Worker Threads でタスクは実行され，Executor JVM は予め設定された上限に達するまで Worker Threads を並行に起動することができる．

2.2 Spark SQL

Spark SQL [1] は，リレーショナル処理を Spark 上で実現するための分散クエリエンジンであり，最も開発が盛んなプロジェクトの一つである．Shark と呼ばれていた SQL on Spark [7] を発展させ，クエリオプティマイザ (Catalyst) や，スキーマ情報を持ったテーブルを用意する API を用いてリレーショナル処理を従来の Spark アプリケーションの中に組み込むこむことが可能になる DataFrame API を用いることで，複雑なクエリ処理に対してもアグレッシブに最適化されたコードを生成する機能を有する．さらに，Spark SQL は Hive と互換性を持っており，既存の Hive クエリを変更することなく Spark 上で実行可能である．

Spark SQL 上で Hive クエリを実行するには，Spark にインテグレートされている Hive の Thrift Server を起動する．Thrift Server は，Spark アプリケーションとして動作し，クラスタマネージャを介して Executor JVM を確保する．その後，Hive クエリは Beeline コマンドを用いて Thirt Server にサブミットされる．

2.3 TPC-H Benchmark

TPC-H†2は，主に RDBMS に対する性能を評価するために用いられるベンチマークであり，8 つのテーブルデータに対して，22 種類の異なるクエリが定義されている．テーブルデータを生成する DBGEN とクエリを生成する QGEN が TPC から提供されており，Hive や Presto，Impala，Tajo といった SQL on Hadoop なシステムの性能評価にもしばしば用いられる [4]．QGEN では各種 DB 向けのクエリが生成可能であるものの，HiveQL などの SQL ライクなクエリ言語には対応していないため，本稿では，Hive 上で実行可能なクエリとしてオンライン上に公開されている TPC-H on Hive†3をベースラインの評価用クエリとして用いている．

2.4 マルチレイヤでの最適化にむけて

Spark は，生成された RDD をインメモリに出来るだけ保持したり明示的にキャッシュすることで，同一のデータセットに対する繰り返し処理を行うようなワークロードに対しては，Hadoop よりも高い実行性能を達成できる．Hadoop においては，MapReduce のインプットデータや中間データの Read/Write が大量に発生する I/O バウンドな実行モデルであるため，I/O スループットを高めることが性能向上の最も重要な要素であった．一方 Spark においては，ディスク I/O が Hadoop に比べて減ってインメモリでの処理に近づいていくため，依然として I/O は重要な要素ではあるものの，ジョブによっては CPU やメモリ，さらにはネットワークバウンドになることが考えられる．そのため，Spark アプリケーションの実行性能を向上させるためには，アプリケーション自体の特性や，オペレーティングシステム，計算機システムのハードウェア構成などをより一層考慮して最適化していく必要があり，考慮しない場合，期待通りの性能が達成できない可能性がある．本節では，システムからアプリケーションを含めた横断的な最適化をしていく上で重要な要素を各レイヤごとに列挙していく．

Spark アプリケーション

ユーザーに並列分散プログラムの知識がなくとも，オペレータを連結させせることで Spark のランタイム側で自動的にタスクが分散処理され，クエリによっては複数のオペレータを同時に実行するような最適化が適用されるが，アプリケーションの動作と Spark のコア動作を正しく理解してアプリケーションを記述することで，さらなる性能向上が期待される．例え

†2 http://www.tpc.org/tpch/

†3 https://github.com/rxin/TPC-H-Hive

ば，SQL においては，Catalyst による最適なクエリプランの生成，Hive に実装されているコストベースオプティマイザ，クエリ対象となるテーブルのカラムやデータサイズなどを事前に解析して統計情報を生成しておくなど，システム側で利用可能な情報から自動的な最適化を期待するだけではなく，SQL の記述そのものに無駄なものがないかを考慮すべきである．

JVM

Spark は，既存の JVM 上で動作するアプリケーション (ウェブアプリケーション，Hadoop など) と同様，JVM 上のチューニングも重要な要素である．巨大なヒープを必要とし，大量のオブジェクトを生成する Spark においては，従来型のアプリケーションと比べてより一層ガーベージコレクション (以下，GC) の性能によってアプリケーションの性能が大きく左右される．さらには，ユーザーが指定可能な JVM の起動オプションには様々な種類があり，どの設定が Spark にとって適切なのかを判断することは容易ではないため，それぞれのチューニングによってどの程度 Spark の実行性能に影響するかを知ることは重要である．

システム

多数の Worker Threads が並列に動作する Spark においては，多数のコアやソケットを搭載するノードでは，Task を処理する Worker Threads とアクセスするデータのメモリ局所性 が乖離した場合，実行性能に大きな影響を与えることが懸念される．また，SMT により OS 側から認識される仮想コア数は非常に多くなっており，よりノードの CPU，メモリ構成を考慮した効率的なスレッド配置をすることは性能向上させる上で重要である．

その他にも考慮すべき点は多く存在するが，本稿ではこれらのレイヤでの最適化，および最適なパラメータを次節にてそれぞれ説明する．

3 SQL クエリの最適化

3.1 一時テーブルの削除

オリジナルの Hive クエリの中には，一時テーブルを作成し，その結果に対して新たなクエリ操作をするものが存在した．これはサブクエリが使えなかったためであると考えるが，一時テーブルを作成する場合，ファイルへ書き戻さなければならず，さらにジョブが分割されてしまうので非効率である．これらのクエリに対しては，サブクエリを作成して出来るだけ少ないジョブ数になるように変換する．

3.2 JOIN 順序の交換

複数のテーブル A, B の JOIN 操作を行う場合，Spark では，A をロードするステージ 0，B をロードするステージ 1，そしてロードしたデータをシャッフルして JOIN するステージ 2 に分割される．(図 1：左) シャッフルでは二つのテーブルデータがネットワークを介して複数の Executor JVM に送られるため，出来るだけシャッフル時のデータ量を減らすことは，Hadoop と同様，Spark でも有効である．異なるサイズのテーブルを多段に JOIN する場合，大きなテーブルから JOIN すると，大量のデータが何度も Executor JVM 間をシャッフルされるため非常に非効率である．よって，出来るだけ小さなテーブルから JOIN していき，大きなテーブルの JOIN 順序は最後になるよう書き換えを行う．

3.3 LIMIT 句による JOIN の変換

上記で述べたとおり，JOIN 時のシャッフルデータ量を減らすことは重要であるため，Spark においてはテーブルのデータサイズに応じて，ShuffleHashJoin と BroadcastHashJoin の二つの JOIN アルゴリズムが自動的に切り替わる．全データを交換する ShuffleHashJoin については前節で説明した通りであり，通常はこちらが使われる．一方，片方のテーブルがかなり小さいとき，例えば 1MB 程度であった場合，BroadcastHashJoin が使われる．小さいテーブルデータを一度ドライバ側に保存した後，これを全 Executor JVM に配られるが，テーブル A はそれぞれの Worker Threads が個々のファイルシステムからロードするためデータ転送をする必要がなくなり，大幅にシャッフルデータ量を削減することができる．(図 1：右)

サブクエリの結果との結合については，自動的には BroadcastHashJoin を用いた JOIN には変換され

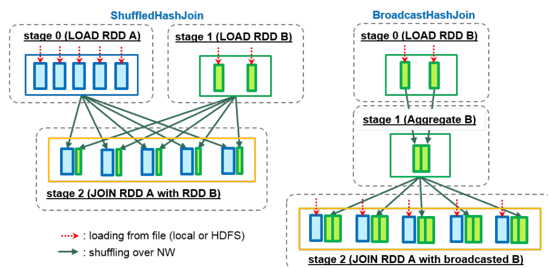

図 1　ShuffleHashJoin と BroadcastHashJoin の処理の流れ

ないが，得られたクエリ結果に対して LIMIT 句をつけることでジョブを分割することができ，明示的に BroadcastHashJoin を用いることが可能となる．サブクエリ結果の行数とデータサイズが予め分かっている，かつ，そのサイズが小さい場合この手法は有効であるが，例えばサブクエリ結果が 100MB 程度である場合，むしろ遅くなってしまうため，LIMIT 句をつけるサブクエリには注意を払う必要がある．

JOIN 順序の変更などは，基本的にはクエリオプティマイザによって自動的に最適化されるものであり，Spark SQL にも spark.sql.codegen オプションを有効にすることで最適なクエリプランを生成するオプションが存在する．しかしながら，今回実験に用いたクエリにおいてこのオプションを有効にした場合，実行時エラーとなってしまうものが多かった．現在開発中の機能であるため，将来的にこれらの最適化の一部は自動的に適用される可能性はあるが，本稿ではこれらのオプションを使用していない．

4　Spark オプションの最適化

4.1　Spark Executor JVM 数の調整

Standalone モードでは，*spark-env.sh* 内でいくつかの環境変数をセットすることができ，SPARK_WORKER_INSTANCES で 1 ノードあたりの Executor JVM 数を，SPARK_WORKER_CORES で 1 Executor JVM あたりの Worker Threads 数を，SPARK_WORKER_MEMORY で 1 Executor JVM あたりのヒープメモリサイズを設定することができ，何も設定しない場合，Executor JVM はただ 1 つのみでノード内の全コアを利用する設定で起動される．

正確には，アプリケーションサブミット時の引数でそのアプリケーションに必要なコア数や Executor JVM のメモリサイズを動的に確保して JVM を起動することが可能であるが，今回，後述する NUMA ノードの構成を考慮して Worker Threads をノード内 CPU コアにバインドさせるため，Standalone モードのノードマネージャに登録する Executor JVM 数などを予め決定して設定しておく．

ノード内で複数の Executor JVM を生成することは，ノード内であっても Executor JVM 間でのデータ通信 (シリアライズ・デシリアライズ) が発生するため，デメリットしかないように思われるが，ノード内で用いる Worker Threads とヒープメモリを固定する場合，1 Executor JVM あたりのヒープサイズによって GC Pause Time も変化する．例えば，巨大すぎるヒープを JVM に割り当てる場合，GC Pause Time は一般的に大きくなる．さらに同じ Executor JVM 内の Worker Threads であっても，NUMA ノードをまたぐように配置されるとメモリアクセスコストによって性能が変わる．これらの影響を踏まえて，Executor Threads の数は 1 物理コアにつき 2 つに制限し，異なる Executor JVM 数であっても合計で 48 Worker Threads となるように調整した上で，複数の Executor JVM を用意した場合にどのような性能変化が見られるかを調べていく．

4.2　Executor JVM のオプション

spark.executor.extraJavaOptions にセットすると，任意の JVM オプションを Executor JVM に追加可能である．Executor JVM では多数の Worker Threads が起動するほか，GC や JIT，各種モニタなど JVM 由来の Threads，Spark の内部動作に必要な Threads が存在するため，例えば，GC アルゴリズムやヒープサイズの調整，GC Threads 数の調整，不要な Profiliing Threads の停止など多岐に渡る．本稿では，GC や JIT のログを元にして System.gc() を止めるオプションと GC Threads 数の調整などを行った．

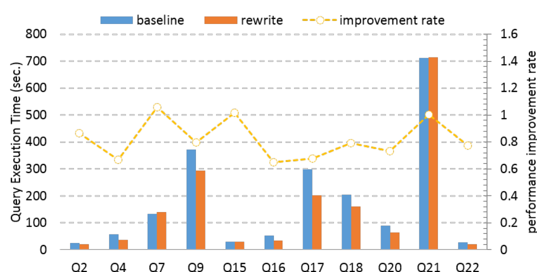

図 2　クエリ書き換えによる性能比較

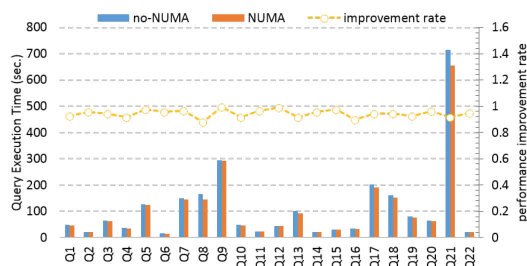

図 3　NUMA 設定を行ったときの性能比較

5　NUMA を考慮した最適化

各 Executor JVM では複数の Worker Threads が動作するが，これらの Worker Threads は，OS 側のスレッドスケジューリングポリシーと，JVM のスレッドポリシー，そして Spark ランタイムでの Task 実行のローカリティポリシーなどが相互に関係する．それぞれが互いのステータスを全て把握した上で実行するコアを決定するわけはないため，複数の NUMA ノードに JVM 内のスレッドが分散してしまう場合，メモリアクセスの遅延が発生し性能低下を引き起こす可能性がある．Linux では numactl コマンドを用いることでプロセスに対する CPU アフィニティを設定することができ，Executor JVM を numactl コマンド経由で起動して，Executor JVM が使える CPU を制限することで，Executor JVM 内で動作する Threads が異なる NUMA ノードで実行されるのを防ぐことを行う．

6　評価

6.1　実験環境

3, 4, 5 節で述べたチューニング方法の効果を調べるため，本稿では 3.3GHz の IBM POWER8 プロセッサ (12 コア) を 2 ソケット (合計 24 コア) 搭載した 1 ノードで実験を行った．メモリは 1024GB，ディスクは 1TB，OS は Ubuntu 14.10 (kernel: 3.16.0-31-generic) であり，SMT8 モードで動作しているため，OS 側から認識されるトータルのコア数は 192 である．ソフトウェアは，Java は IBM J9 JVM (1.8.0, SR1 FP10)，Scala 2.10.4，Spark 1.4.1，Hadoop 2.6.0, Hive 0.13 (Spark 内部でサポート) を用いた．

TPC-H で提供された DBGEN を用いて Scale Factor 100 (100GB) のデータセットを作成し，Hive Table は Parquet 形式で保存され snappy 形式で圧縮した後，Hive Table にロードする．HDFS のブロックサイズは 128MB である．TPC-H クエリは，2.3 節で述べた通り，github 上に公開されている Hive 用のクエリをベースとする．

以下の実験では，特にことわりがない場合，ノード内で用いるトータルの Worker Threads は 48，ヒープメモリは 192GB とし，Executor JVM 数は 8 とする．このとき，個々の Executor JVM では 6 Worker Threads が動作し，ヒープサイズは 24GB である．

6.2　クエリ書き換えによる性能

まずはじめに，ベースラインからクエリを書き換えたときの性能を測定する．図 2 は書き換えた 11 個のクエリの実行時間を比較したものであり，多くのクエリにおいて 20%程度性能が向上することを確認した．Query 11 も書き換えを行っているが，ベースラインのクエリが 100GB データセットでの実行に失敗するため，図からは除いている．Query 11 では BroadcastHashJoin が使われていたが，本来 Broadcast するのに適さない大きなテーブルを Broadcast していたため，非常に長い時間がかかるクエリとなっていた．JOIN 順を適切に入れ替えることで，小さいテーブルを Broadcast することができるようになり，100GB データセットでも実行可能となった．別途 10GB のデータセットを用意し Query 11 の性能比較を行ったところ，20 倍以上高速化されることを確

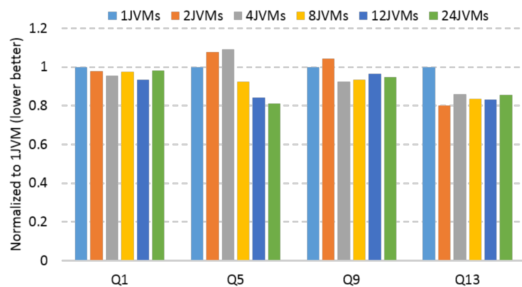

図 4　Executor JVM 数を変えたときの性能比較

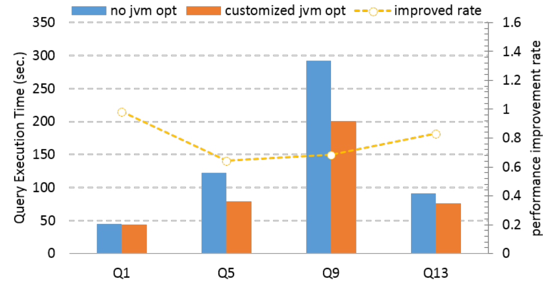

図 5　JVM Option を追加したときの性能比較

認した．Query 9 では，JOIN の順序が変わることでシャッフルされる総データ量が 48.4GB から 36.3GB まで削減可能であり，その結果，21%程度性能が向上した．その一方であまり変化のないクエリも確認できる．Query 7 や 15 ではクエリ書き換えにより一時テーブルを削除してサブクエリ内で処理するようにしたが，サブクエリの結果がそもそも小さく，処理段数が増えることのオーバーヘッドによりトータルで見た実行性能は書き換え前と同程度になっている．また，Qurry 21 では，クエリの書き換えによりシャッフルデータサイズが減る一方，個々の Executor JVM での GC に一時停止時間が多くなっており，結果としてクエリ書き換えの効果が相対的に小さくなってしまっている．しかしながら，これは JVM のオプションを変更することで改善することができ，この結果については 6.5 節にて言及する．

6.3　NUMA 設定による性能

次に 5 節で述べた NUMA の設定を行った場合の性能を測定する．Executor JVM が利用するコアを numactl を用いて個々の NUMA ノードにアラインするよう設定し，メモリのインターリーブ設定は行わず，2.1 節で行った書き換え後のクエリを用いて，numactl でコアを指定をする場合としない場合の性能を比較したときの結果を図 3 に示す．全クエリで numactl を用いた場合に 10% - 15%程度性能が向上することを確認した．とりわけ実行時間の短いクエリよりも長いクエリや，Executor JVM 間でのシャッフルが多く発生するようなクエリにおいて効果が見られた．

6.4　Executor JVM 数による性能変化

次に Executor JVM の数を調整したときの性能を測定する．1 ノードで用いる Execuor JVM 数を 1, 2, 4, 8, 12, 24 と変えていくとき，トータルの Worker Thread を 48，ヒープサイズを 192GB に固定して，個々の JVM に均等に割り振るように設定する．また，個々の JVM に対して NUMA の設定を行い，1 物理コアにつき 2 つの Worker Threads が割り当たるようにする．図 4 は，Query 1, 5, 9, 13 に対し Executor JVM 数を変化させ，それぞれのクエリの 1 Executor JVM のときの結果を基準としたときの性能変化を示している．全てのクエリにおいて，複数の JVM を用いたときのほうが性能が良いという結果が得られた．シャッフルデータ量が少ないクエリ (Query 1) に関してはあまり性能変化は見られないが，シャッフルデータ量が多いその他のクエリに関しては，だいたい 20%程度性能が改善した．

6.5　Executor JVM オプションによる性能変化

これまでの結果を踏まえてシャッフルデータ量が多いクエリの GC ログを分析したところ，実行時間の 15 - 30%程度が GC に費やされおり，また，Global GC が数多く呼び出されていることを確認したため，Executor JVM に対してユーザー側から System.gc() が呼び出されても Global GC を起こさないオプションを追加した．さらに，数あるオプションの中から性能向上に有効であるものを探索し，結果として次のようなオプションを Executor JVM に追加した．(*-Xnocompactgc -Xnocompactexplicitgc -Xdisableexplicitgc -Xmn12g -Xgcthreads12 -Xtrace:none -Xnoloa -Xgc :tlhMax-*

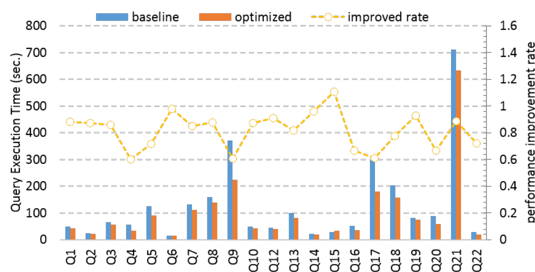

図 6　全ての最適化をおこなった後の性能比較

imumSize=32768 -XlockReservation -Xdump :system:none -Xdump:heap:none) 図 5 は，このときの結果を示したものであり，シャッフルが多い Query 5, 9, 13 においては，20 - 30%程度性能が向上することを確認した．

6.6　全オプション適用時の性能比較

以上の結果を踏まえて，今回行った全ての最適化を適用した後の全クエリの実行性能を比較したものを図 6 に示す．ベースラインはクエリ書き換えをせずに，8 Executor JVMs，6 Worker Threads/JVM，24GB/JVM のヒープを割り当て，NUMA や JVM オプションを追加しないときの結果である．ほぼ全てのクエリにおいて実行性能が 20 - 40%高速にクエリが実行できることを確認した．一方，実行時間が短いクエリ (Query 6, 14, 15) に関しては，ほぼ変わらないか若干性能が悪くなる傾向が観測された．その他のクエリ向けに多数の Reduce タスクに細分化してヒープメモリを圧迫しないようにするため spark.sql.shuffle.partitions をデフォルトの 200 から 400 程度まで増加しているが，タスクの計算時間が少ない場合，計算時間よりもオーバーヘッドのほうが多くなってしまうため，全体の実行時間が短いクエリにおいてはこのオーバーヘッド影響により遅くなる結果となった．

7　関連研究

Spark の公式サイト†4 などに様々なチューニングに関するガイドがいくつか掲載されている．文献[5] では，Garbage Collection のチューニングにフォーカスが当てられており，OpenJDK 上で G1GC アルゴリズムを用いた場合の最適化について述べられているが，我々の知る限り，Spark 上の Executor JVM オプションや JVM 数のチューニング，システムレベルの最適化について論じている文献についてはまだ存在していない．

Spark はクラスタ環境で動作することが想定されているため，ネットワークやシャッフルについての最適化についての研究がいくつか存在する．文献[3] では，シャッフル時にファイル数が Mapper と Reducer の数の積のオーダーで増加していき通信のオーバーヘッドが増加するのを防ぐため，Reducer 数と同等となるようシャッフルデータと統合するためのオプションについて論じている．本稿でもこのオプションは常に有効化した．また文献[6] では，Infiniband 上で RDMA を Spark の Shuffle 通信に用いることで，既存の NIO や Netty ベースの Shuffle 通信を IP over IB ネットワークで実行するよりも高速に実行できることを示している．

Spark SQL 上での TPC-DS の性能測定結果は，Databricks などの Developers Blog などで性能を測る指標として用いられているが，TPC-H に関して言及されている文献は少ない．文献[4] では，Hive on MR, Hive on Tez, Impala など SQL on Hadoop なシステム上で Parquet や ORC などのカラムナーベースのテーブルや，snappy の圧縮を適用した場合などの性能について TPC-H のクエリを題材に論じているが，Spark に関しては行っていない．

8　まとめと今後の課題

本稿では，TPC-H Benchmark を題材として，アプリケーションレベルから Spark や JVM，OS レベルのチューニングによって，TPC-H クエリの実行性

†4　http://spark.apache.org/docs/latest/configuration.html

能がどの程度改善できるかを定量的に示した．その結果，クエリの書き換えにより 20%程度，NUMA の設定を加えることでほぼ全てのクエリで 10%程度，Executor JVM 数を調整することで 20%程度，JVM オプションのチューニングにより 20-30%程度性能が向上した．全てのチューニングを適用させた場合の性能は，何もチューニングを行わない場合と比べて，最大で 40%程度性能が改善することを確認した．今後の課題としては，クエリの傾向ごとに最適なチューニングは異なっているため，クエリごとの特徴を踏まえてより最適な設定を行うことや，これらの知見を Spark ランタイムやスケジューラに適用し，コミュニティにフィードバックしていくことを考えている．

参考文献

[1] 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, 2015, pp. 1383–1394.

[2] Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G. R., Ng, A. Y., and Olukotun, K.: Map-Reduce for Machine Learning on Multicore, *NIPS*, Schölkopf, B., Platt, J. C., and Hoffman, T.(eds.), MIT Press, 2006, pp. 281–288.

[3] Davidson, A. and Or, A.: Optimizing Shuffle Performance in Spark, Technical report, University of California, Berkeley - Department of Electrical Engineer- ing and Computer Sciences, Tec Rep., 2013.

[4] 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(2014), pp. 1295–1306.

[5] Kaczmarek, E. and Yi, L.: Taming GC Pauses for Humongous Java Heaps in Spark Graph Computing, *Spark Summit 2015*, 2015. https://spark-summit.org/2015/events/taming-gc-pauses-for-humongous-java-heaps-in-spark-graph-computing/.

[6] X. Lu, W. Rahman, N. I. D. S. D. P.: Accelerating Spark with RDMA for Big Data Processing: Early Experiences, August 2014.

[7] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I.: Shark: SQL and Rich Analytics at Scale, *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, New York, NY, USA, ACM, 2013, pp. 13–24.

[8] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, USENIX Association, 2012, pp. 2–2.