バッファキャッシュを用いた Spark シャッフル処理の最適化に向けて

吉村　剛[1,a)]　千葉　立寛[1,b)]　堀井　洋[1,c)]　小野寺　民也[1,d)]

概要：Spark の利用は商用の大規模データ解析やバッチ処理において急速に広まっており，最適化による計算資源利用の効率化は重要である．Spark は特にシャッフル処理によるオーバヘッドが性能低下の原因と知られている．本研究では，シャッフル処理の最適化に向けて，バッファキャッシュ利用の重要性を示すことを目的とする．そのために TPC-H ベンチマークのクエリ 22 種類の性能を分析し，特徴的なクエリをさらに掘り下げて分析をする．現状で得られている知見は，バッファキャッシュ利用率を上げるためにヒープ使用量をできるだけ削減することが重要であることがわかっている．特に，spill が発生してもバッファキャッシュ利用率を優先すべき場合もあることや，Spark/JVM のメモリがバッファキャッシュを圧迫して性能が低下する場合があることを確認している．

1. はじめに

Spark [1] の利用は商用の大規模データ解析やバッチ処理において急速に広まっている．例えば，IBM は Spark に対して 3 億ドルの投資を決めており [2]，データ分析に関するサービスの主要な基幹ソフトウェアとなっている．ワークロードの最適化による計算資源利用の効率化はそのようなサービスの品質の向上やユーザエクスペリエンスの向上にとって重要な課題となっている．

Spark において，シャッフル処理の最適化は現在も課題となっており，様々な研究がされてきている [3] [4] [5]．また Spark 自身も Tungsten プロジェクト [6] 以来，大きく実装が変化し，バージョンが 2.0 となりさらなる最適化が進んでいる．

本研究は Spark のシャッフル処理のさらなる最適化に向けて，バッファキャッシュ利用の重要性を示すことを目的とする．そのために，TPC-H ベンチマークを動作させたときのメモリ利用およびディスク I/O の特性を分析する．TPC-H は商用のデータ分析のためのワークロードを模した標準のベンチマークのひとつであり，現実にあるワークロードの代表的な特性を反映しているとされている．

本論文では著者が構築した環境において発生した，バッファキャッシュと Spark に関わる興味深い特性について集中して分析した結果を報告する．特に Java ヒープに関する設定を 4 通り利用し，シャッフルで余分なディスク I/O の要因となる spill の発生の有無に注目する．ヒープに関する設定はヒープの最大容量およびシャッフルの利用するヒープを Java ヒープ内か Java ヒープ外のオフヒープと呼ばれる領域を利用するかについて注目する．

はじめに TPC-H ベンチマークにある 22 種類のクエリを全て動作させ，そこでより特徴的であったクエリを選択し，さらに掘り下げた分析をする．分析は現在進行形で進んでおり，現在確認できている知見をまとめると以下になる．

(1) Spark の Java ヒープの設定により，OS の管理するバッファキャッシュ利用率が増加し，性能が大きく改善する場合がある．さらに，ヒープの設定が小さく，spill が発生する状況であっても，バッファキャッシュの利用効率を高めたほうが性能が向上する場合がある
(2) Spark/JVM を再起動せず，連続して処理を行うと空きメモリが圧迫されてバッファキャッシュが減少し，性能が低下する場合がある
(3) オフヒープの利用により，OS へのメモリ返却が増加する結果，バッファキャッシュの利用率が向上する
(4) Spark 2.0 の最適化により，メモリ領域が積極的に解放されバッファキャッシュの利用率が向上している

本論文ははじめにバッファキャッシュに注目する動機づけを行う予備実験の結果と得られた疑問を 2 節で述べる．次に，3 節において TPC-H ベンチマークを用いた調査について，その方法の詳細や利用するツールを述べる．4 節

[1] IBM 東京基礎研究所
[a)] tyos@jp.ibm.com
[b)] chiba@jp.ibm.com
[c)] horii@jp.ibm.com
[d)] tonodera@jp.ibm.com

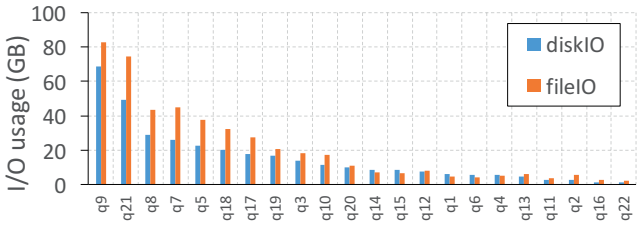

図 1 TPC-H クエリごとのファイルとディスク I/O バイト数

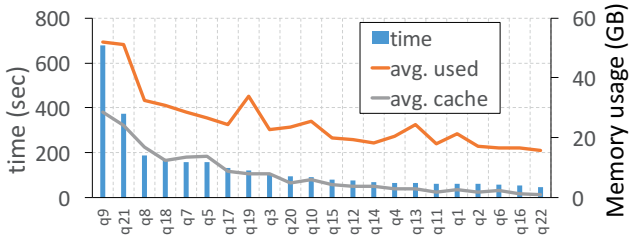

図 2 TPC-H クエリごとの処理時間とバッファキャッシュバイト数

ではいくつかの設定で実際にベンチマークを動作させ，得られた分析結果を報告する．5 節ではベンチマークの中でもより興味深い挙動をしたクエリについてさらに掘り下げて分析を行う．その後，関連研究を 6 節で述べ，最後に本研究のまとめを 7 節で述べる．

2. 研究の動機

Spark はインメモリ処理による最適化がよく知られているものの，実際は最初と最後のデータ入出力以外にもシャッフルでディスクへの書き込みを行う実装になっている．具体的にはシャッフルするデータを載せたメモリが不足した場合に spill を起こす場合とシャッフルで送信の準備が完了したときの 2 つに分かれる．送信準備が完了したとき，最終的にファイルに書き出し，後で次のステージのタスクが送信を要求したときに再度ディスクから読み出して送信をしている．

2.1 予備実験

Spark のシャッフル処理の特性を端的に示すため，TPC-H ベンチマークを 3 ノードで測定する．ここでは Spark 1.6.2, Hadoop 2.6.4 の HDFS 上にある 100GB の Hive テーブルを入力とし，Java ヒープ量の設定は各ノードの RAM のサイズ 31GB に対して最大 80% 程度の利用率となるようにする．設定や実験環境の詳細は 3 節で述べる．測定はノードごとに OS から得られる I/O バイト数などを用いて，クエリごとの特性を比較するため 3 ノードの合計または平均を分析する．また Spark はチューニングやデバッグを簡単にするために，ファイル書き出しバイト数やレコード数などの基本的なメトリクスを測定しているため，それも利用する．

図 1 はクエリごとの Spark が追跡しているファイル I/O

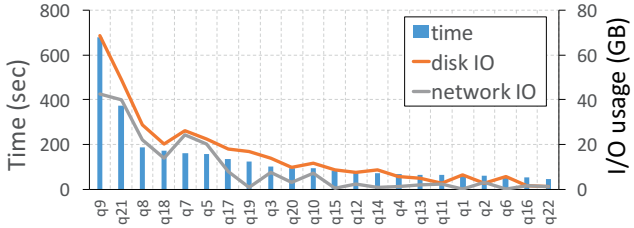

図 3 TPC-H クエリごとの処理時間と I/O

バイト数と実際のディスク I/O バイト数を示している．TPC-H ベンチマークはクエリごとに大きく性能特性が変わるものの，多くのクエリで OS がディスク I/O をキャッシュし，最大 20GB から 30GB の I/O を削減することができている．図の q9, q21 は join を何度も行うクエリになっており，シャッフル処理を繰り返すクエリほどファイル書き出しが多いワークロードになっている．

図 2 が示すメモリのキャッシュ利用量と比較して見ると，I/O が増えるほど OS 内のキャッシュが増えており，その結果実際のディスク I/O が減少していると考えられる．また，図 3 はクエリごとの所要時間と I/O のバイト数に相関があることを示しており，またネットワーク I/O と比較してディスク I/O のほうが性能と相関がより強く表れている．このように，Spark は Linux のキャッシュ機構がうまく利用できる場合，大幅にディスク I/O を減らし性能を改善することができると予想される．逆に，Spark のワークロードの I/O 特性は Spark のログだけではなくシステムレベルの測定も必要であることがわかる．

以上から，OS のキャッシュをうまく使うことで Spark のシャッフル処理の性能を大きく向上させる可能性があることが予想される．しかし，spill の発生などを考慮すると，単純に Java ヒープの容量を小さくしたほうが性能が良くなるとは限らないことに注意したい．また Spark は様々な設定項目を用意しており，設定が変わればワークロードの性質も大きく変わる可能性がある．考えられる疑問をまとめると以下のようになる．

(1) Spark のヒープ設定はディスク I/O の増減に対してどのように影響を与えるのか？

本実験では十分なヒープ量であったために spill は発生していないものの，これ以上ヒープ容量を小さくして spill が発生するとそれだけファイルの読み書き量は増加してしまう．ただし，そこで増加したファイルも OS が管理するメモリ上に配置されるため，実質のオーバヘッドはオブジェクトをシリアライズする部分だけになるかもしれない．また，ヒープ容量の減少はガベージコレクションの増加による性能低下などが避けられないことも考慮しなければならない．

(2) Spark や JVM でのキャッシュと OS のバッファキャッシュはどちらが重要か？

Spark では Java のオーバヘッドを削減するように, JVM を起動し続けたままクエリを何度か処理することもできる．既存の Spark の性能を分析した研究 [7] もその状況を前提として分析を行っている．その場合，一般的には処理時間は短くなると知られているものの，メモリの使用量は JVM を毎回起動するよりも多くなるため，バッファキャッシュを圧迫する恐れがある．特に，Spark 1.6.2 のように Java ヒープの最小量を最大量と同じになるように JVM が起動されている場合に問題になる可能性が高い．

(3) オフヒープを利用した場合にバッファキャッシュ利用の特性は変化するのか？

Spark はガベージコレクション対象となる通常の Java ヒープ領域だけでなく，sun.misc.Unsafe を利用した，C 言語と同じようにメモリ確保・解放が可能なメモリ領域を持つ．その領域はオフヒープ領域と呼ばれ，Tungsten プロジェクト以降に利用できるようになっている [*1]．Spark デフォルト設定ではシャッフル処理でオフヒープを使うようにはなっていないため，予備実験ではその影響を計測できていない．しかし，Spark は通常のヒープと同様に設定した最大利用量から spill するかどうかを制御するため，オフヒープでもバッファキャッシュ利用への影響は発生すると考えられる．

(4) Spark のバージョンによってシャッフルの性能は変化するのか？

Spark 2.0 から Whole-stage Codegen が導入され, シャッフルの最適化が進んでいる．最適化が本研究のケースでも効果があるのか，またそれはなぜかを知ることは今後の Spark の改良に向けて重要な知見になると考えられる．

本研究では以上の 4 つの疑問に対して答えることを目指し，TPC-H ベンチマークの性能をより深く分析する．

3. 分析方法

3.1 分析環境

本研究では Spark 公式サイトからダウンロードできるビルド済みのバイナリパッケージを利用する．論文執筆時の最新である Spark 2.0.1 と Spark 1.6.2 を利用する．分散ファイルシステムは Hadoop 2.6.4 の HDFS を公式サイトからビルド済みのパッケージをダウンロードして利用している．そのため，Spark のバイナリはいずれも Hadoop 2.6 に対応したものを利用する．

環境は予備実験と同じく，3 ノードの Ubuntu 16.04 で構成されたクラスタを利用する．3 ノードの Spark, HDFS の Java プロセスをまとめると図 4 となる．それぞれのノードで, HDFS のデータノードを動作させ，shortcircuit による最適化以外はデフォルト設定のままにする．ネームノード，セカンダリノード，データノードの 3 種類はヒープ最大利用量 1GB とする．Spark は分析全体に渡り圧縮コーデックに LZ4，シリアライザには KryoSerializer を使い，ドライバのヒープ最大値は 4GB に固定する．その他の Spark マスタ，ワーカも 1GB をヒープの最大使用量とし，エクゼキュータのヒープを主に制御する．また，Spark 2.0.1 では，Java のヒープ最小使用量 (-Xms) を設定できるようになっているものの，Spark 1.6.2 と同様に最大使用量と同じになるように設定する．

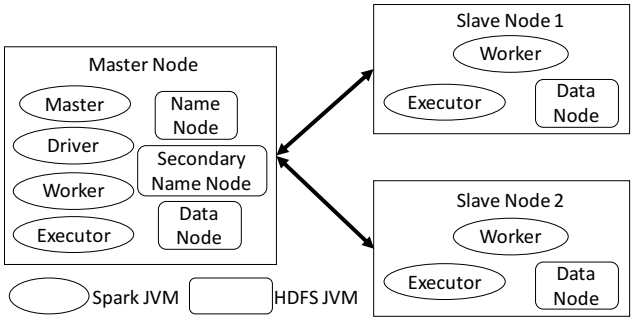

図 4 クラスタで動作する Java プロセス

また，性能を測定する前に必ず OS のキャッシュメモリを一度無効にし，さらに全ての JVM を再起動することで測定前のキャッシュの影響が起きないようにする．ここで利用する入力データは dbgen でスケールファクター 100 で生成したデータを Parquet フォーマットに変換したものを利用している．Parquet フォーマットはカラムナ型のデータ表現方法で，Spark でも様々な最適化が施され広く使われているため利用している [8].

利用したハードウェアは Intel Xeon E5-2680 (2 ソケット，16 論理コアの合計 32 コア, 2.70GHz) マシンを 3 台利用する．RAM は 3 台とも 31GB，ストレージは 1TB の SCSI ハードディスクドライブ，ネットワークは 10Gbps インフィニバンドを利用する．

3.2 分析ツール

本研究では性能分析のためにツールとして dstat, Java GC の統計量を利用する．dstat は CPU 利用率，ディスク I/O 量，メモリ使用量，ネットワーク I/O 量など様々なシステムレベルの統計量を収集できる．Java GC の統計量はガベージコレクション自体の性質のみならず，Spark が実際に使用していたメモリ量を観測することができる．dstat から見ると Java プロセスのメモリ使用量は JVM が管理している領域が使用中ということのみ判断できるものの，GC の情報と組み合わせることで実際にプログラムが必要としていたヒープ使用量などを計測できる．

4. TPC-H ワークロードの性能分析

2 節で示した疑問に答えるため，TPC-H ベンチマークについて Spark 1.6.2, 2.0.1 上で，4 種類のヒープ設定で

[*1] 対して通常の Java ヒープ領域をオンヒープと呼ぶ

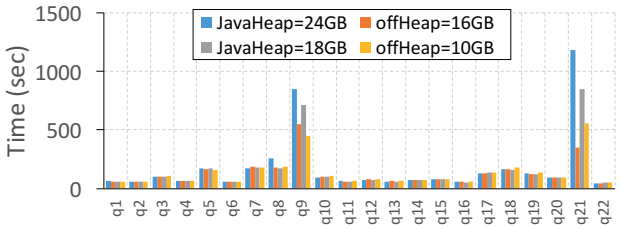

図 5 Spark 1.6.2 の TPC-H クエリごとの処理時間

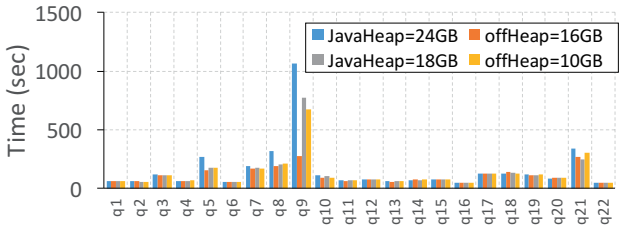

図 6 Spark 2.0.1 の TPC-H クエリごとの処理時間

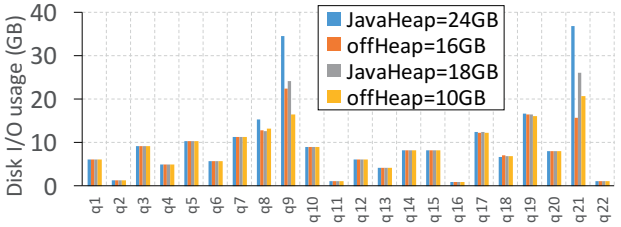

図 7 Spark 1.6.2 の TPC-H クエリごとのディスク読み出し

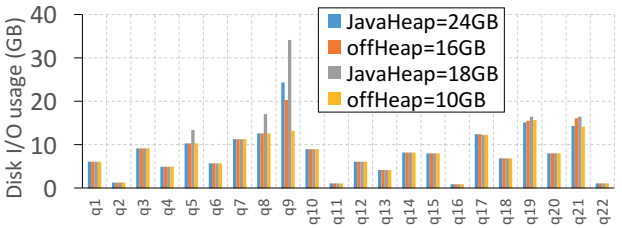

図 8 Spark 2.0.1 の TPC-H クエリごとのディスク読み出し

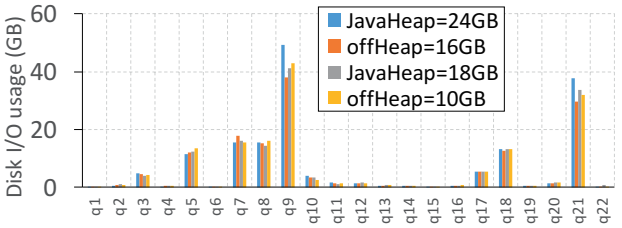

図 9 Spark 1.6.2 の TPC-H クエリごとのディスク書き出し

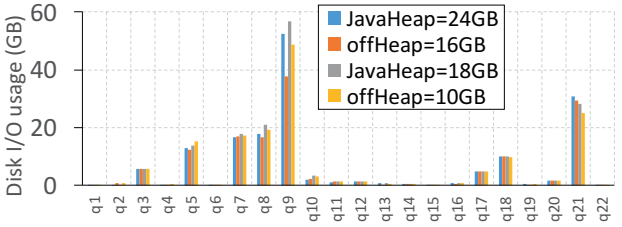

図 10 Spark 2.0.1 の TPC-H クエリごとのディスク書き出し

計測・比較する．使用した設定はオフヒープを設定せず Java ヒープ 24GB, 18GB を設定した場合と Java ヒープ 8 GB に対してオフヒープを 10GB, 16GB に設定した場合の 4 種類とする．また，2 回連続で同じクエリを実行した場合の結果も分析し，Spark/JVM でのキャッシュの効果を分析する．そのとき，バッファキャッシュに関しては無効にし，入力データが必ず HDFS から読み出されるようにする．

4.1 ヒープ使用の制御による性能変化

図 5, 6 はクエリの処理時間を示している．Java ヒープ 24GB のときに比べてヒープが少ないほど性能が向上しているクエリと，ほとんど変化のないクエリに分けられることがわかる．特に q9 についてはいずれのバージョンでもオフヒープによる効果が最も見られる．Spark 2.0.1 と Spark 1.6.2 を比べると，Spark 2.0.1 はあまり大きな性能向上は見られず，むしろ q9 などの性能は低下している．

図 7, 8 はディスク読み出しバイト数を示している．ヒープ設定の変更によって処理時間にあまり変化が見られなかったクエリはいずれもディスク読み出しの量も変化していない．一方，q9 や q21 はディスク読み出しがメモリ設定が小さいほど大きく減っている．Spark 2.0.1 でもこの傾向は変わらず，q9 以外のクエリはディスク読み出しの量はあまり変化していない．ただし，q21 ではオフヒープを利用したほうがディスク読み出し量は増えてしまっている．

図 9, 10 はディスク書き出しバイト数を示している．ディスク読み出しに比べて全体的にディスク書き出し量は少なく，また書き出し量の変化は少ない．ただし，q9 に関しては例外的にオフヒープを利用した場合に書き出し量が増加している．これはシャッフルの spill によるものであることが，Spark のログから確認されている．オフヒープ利用により q9 の処理時間が短縮した原因は，spill が増加したことに比べてディスク読み出しを減らす効果のほうが大きかったためと考えられる．実際，ディスクの読み出しバイト数の減少量はディスクの書き出しバイト数に比べて大きい．

マスタノードでは Java ヒープ 24GB の設定のとき，Java ヒープメモリ利用の合計値がシステムの物理メモリ量を超えてしまう可能性がある．スワップの設定は Linux のデフォルト設定としており，最大数 MB 単位でページイン・アウトが発生していた．しかし，この値は全体のディスク I/O の量と比較して十分小さく性能への影響は少ないと考えられる．

以上まとめると，2 節であげた疑問の中の，(1) Spark のヒープ設定のディスク I/O の増減に対する影響は次のように考えられる：Spark のヒープ設定はできるだけ小さい方

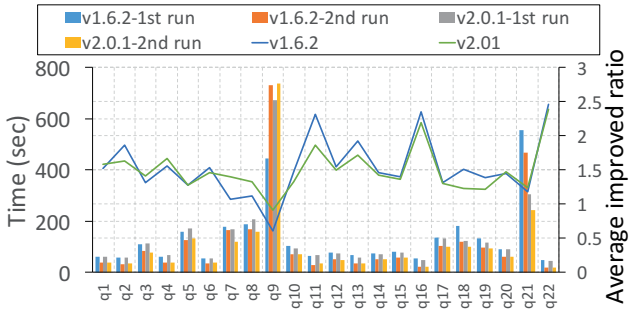

図 11 2 回目の TPC-H クエリごとの処理時間の変化

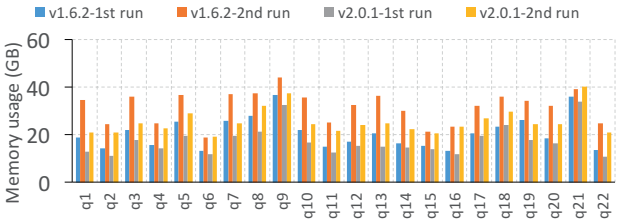

図 12 2 回目の TPC-H クエリごとのメモリ使用量の変化

が OS のバッファキャッシュによってディスク読み出しを減らすことができる．一方，ディスク書き出しは spill によって増加するものの，ヒープをあまりにも小さくしない限り，spill よりもバッファキャッシュによるディスク読み出し削減の効果が上回る場合が見られる．クエリ 9 に関しては 5 節においてより詳細に分析を行う．

4.2 Spark/JVM キャッシュの効果

図 12 は同じクエリを 2 回連続で動作させたときのクエリの処理時間の変化を示しているいずれも Java ヒープ 8GB, オフヒープ 10GB の結果で，ほとんどのクエリは 1.5 倍から 2.5 倍近くの処理時間の短縮が見られる．一方，q9 のみ性能が悪化している．その原因は，図 ?? を見るとメモリの使用量が 2 回目の動作時に増加していることから，バッファキャッシュが圧迫されていることが予想される．

2 節であげた疑問の (2) Spark や JVM でのキャッシュと OS のバッファキャッシュに関する回答は次のようになる：TPC-H ベンチマークのほとんどでは 1.5 倍から 2.5 倍ほどの大幅な性能改善を引き起こす効果が見られ，多くの場合は JVM を再起動せずに連続で動作させる効果は大きい．しかし，本研究で特に詳細に調べたクエリ 9 に関しては例外的に OS のバッファキャッシュ利用を優先したほうが性能が向上するかもしれない．ただし，あくまで予想であるため Spark の内部のさらなる調査が必要となると考えている．

5. TPC-H クエリ 9 の性能分析

2 節で示した中で，最もディスク I/O の負荷が大きいクエリは q9 (クエリ 9 と呼ぶ) であった．また，4 節でもクエリ 9 がその他のクエリと異なるディスク I/O の特性を示しているため，クエリ 9 に焦点をあててより詳細な分析を行う．本節ではそのワークロードについて，いくつかの異なる設定を試行し，それによる性能特性の変化を分析する．それによってシャッフル処理とバッファキャッシュの関係をより深く理解することを目的とする．

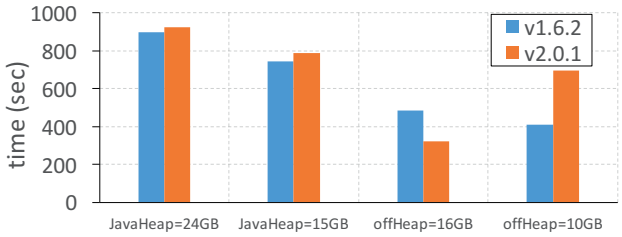

図 13 ヒープ設定ごとの処理時間

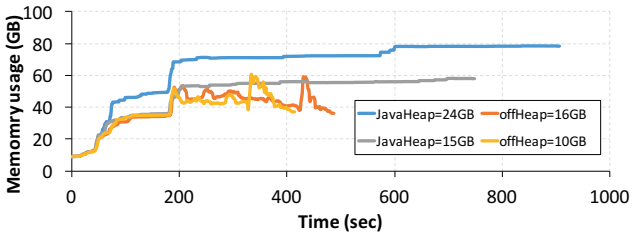

図 14 ヒープ設定ごとのメモリ使用量の時間変動

そこで TPC-H クエリ 9 について Spark 1.6.2, 2.0.1 上で，4 種類のヒープ設定で計測・比較する．使用した設定はオフヒープを設定せず Java ヒープ 24GB, 16GB を設定した場合と Java ヒープ 8 GB に対してオフヒープを 10GB, 15GB に設定した場合の 4 種類とする．

図 13 はクエリの所要時間を示している．Spark 1.6.2 は処理時間は Java ヒープを使う方が短く，またできるだけ容量を小さく設定した方が短くなっている．一方，Spark 2.0.1 ではオフヒープの容量は多いほうが性能が向上している．spill は Java ヒープが 24GB のときと オフヒープが 15GB のときは発生しておらず，それ以外は Spark ログから spill の発生を確認している．逆に Java ヒープ，オフヒープを大きく取るほどバッファキャッシュのための空きメモリは大きくなり，4 種類の設定は異なるバッファキャッシュ利用の特性を反映すると予想される．この結果についてそれぞれ分析を行う．

5.1 ヒープ使用の制御による性能変化

はじめに，Spark 1.6.2 での結果を分析する．結論から述べると，性能差は図 15 が示すように，バッファキャッシュのためのメモリ使用量が増加したためと考えられる．およそ 200 秒を超える辺りまでいずれの設定も同じメモリ使用量の変動の傾向であったのに対し，それ以降に差異が見られる．Java ヒープ 24 GB の設定に関してはメモリ使用が 3 ノード合計で 80 GB 近くまでずっと増加し続け，

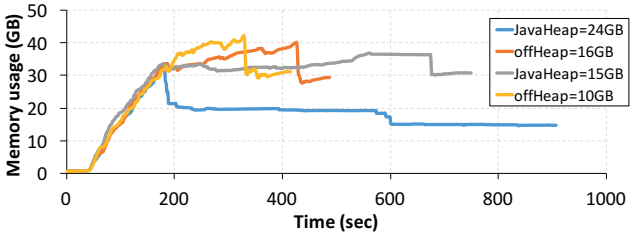

図 15 ヒープ設定ごとのバッファキャッシュ量の時間変動

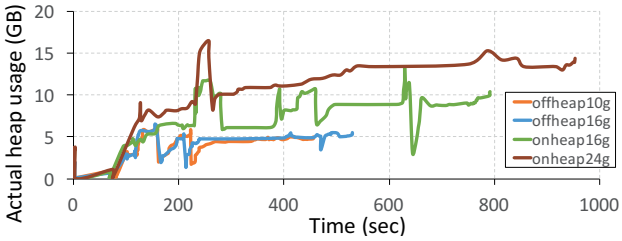

図 16 ヒープ設定ごとの GC 計測の Java ヒープバイト数
この測定値はマスタノードでの 1 台の実際のヒープ使用量を示す.

バッファキャッシュの使用メモリが大きく減少してしまっている．ヒープ量を減らした Java ヒープ 16 GB についても同様の傾向を示している．一方，オフヒープに設定した 2 つのワークロードは 20GB ほどの幅で使用メモリが増減しており，その結果バッファキャッシュを多く確保できていることがわかる．

図 16　はガベージコレクションのトレースから得られる，実際のヒープ使用量になる．Java ヒープ 24GB の場合は最大 15GB 近くまでヒープを消費するものの，それは一時的であることが確認でき，Java ヒープの量を 15GB まで減らした場合でも同様の傾向を示している．オフヒープの場合，この急激な上昇は発生しておらず，シャッフルに使用するオフヒープメモリの増減がなくなっていることが確認できる．ヒープ最大値が削減された結果ガベージコレクションの頻度が高くなっている．しかし，全体の処理時間に対する影響は小さい．

Java ヒープのメモリ使用が増加し続ける原因のひとつは Spark 1.6.2 では Java の最小メモリ使用量が必ず最大メモリ使用量と一致するように常にエクゼキュータの JVM が起動することがあげられる．最小メモリが小さく設定されていれば，オフヒープほどではないにしても JVM が OS へメモリを返すため増加は抑えられる．ただし，この設定方法は Java ヒープ領域を拡大するオーバヘッドが削減される効果もあるため，ヒープ使用の増減が安定しているワークロードでは性能が上がる傾向にある．

図 17, 18 は実際のディスク I/O 量の時間総量を示している．ヒープ設定によって 200 秒以降のシャッフル処理でのディスク読み出し量に変化が起きていることが確認できる．オフヒープを設定したワークロード同士を比較した場合はディスク読み出しの量はあまり変わらない一方，ディスク書き出しの量で差が見られる．この差はオフヒープの

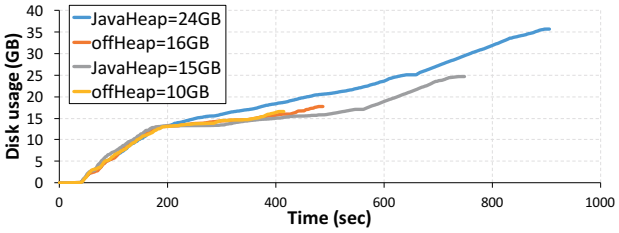

図 17 ヒープ設定ごとのディスク読み出しバイト数の時間総量

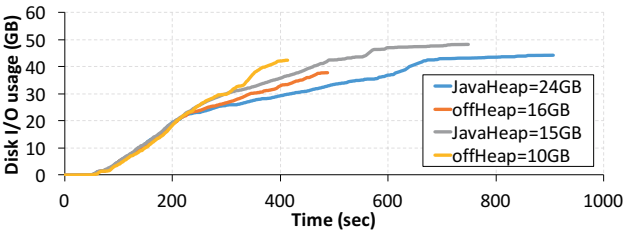

図 18 ヒープ設定ごとのディスク書き出しバイト数の時間総量

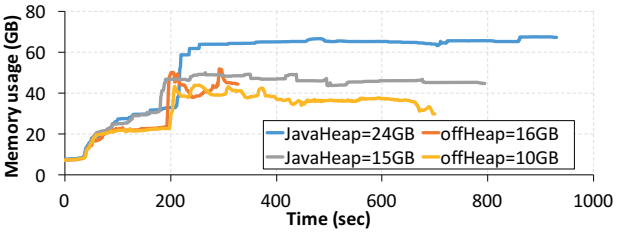

図 19 ヒープ設定ごとのメモリ使用量の時間変動

量が少ないために spill が発生しているためとなっており，実際に Spark のログでも spill が発生していることが確認できている．また，図 16 にあるように，Java ヒープの量を 16GB にしたときの実際のヒープ使用量からも spill に伴うメモリの増減が確認出来る．それにも関わらず spill が発生している 10GB の設定の方が処理時間が短くなっており，バッファキャッシュによる性能向上がより強く出ていることがわかる．spill の発生は Java ヒープの設定を 16GB にした場合も見られ，同様の傾向になっている．

以上から，q9 においてヒープ使用は最大値はできるだけ少ないほうがキャッシュの利用が増加し，ディスク I/O を減少させ結果的に性能が大幅に改善することがわかる．オフヒープについても GC を抑制する効果は説明されてきたものの，シャッフル処理の負荷が大きいワークロードの場合はメモリ利用効率が向上し，バッファキャッシュ利用が増加するという点で性能を改善する方法としても重要であることがわかる．

5.2 Spark 2.0 と Spark 1.6.2 の比較

Spark 2.0.1 における実験では，オフヒープのサイズが小さいほうが性能が良くなっている．図 19 はメモリの使用量の時間変化となっており，バッファキャッシュ量の変動は図 20 から確認できる．確かにバッファキャッシュの

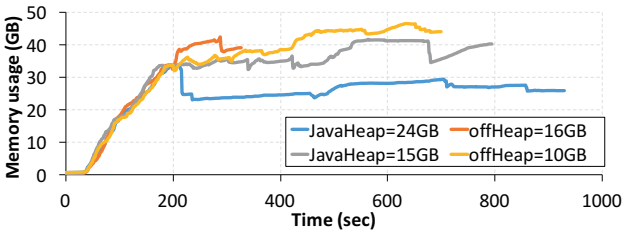

図 20 ヒープ設定ごとのバッファキャッシュ量の時間変動

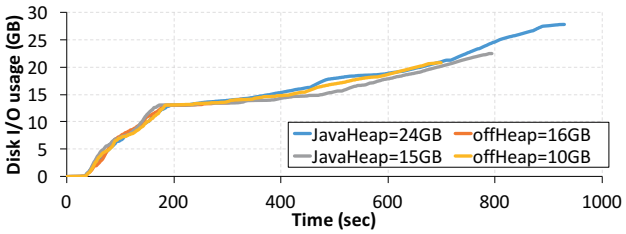

図 21 ヒープ設定ごとのディスク読み出しバイト数の時間総量

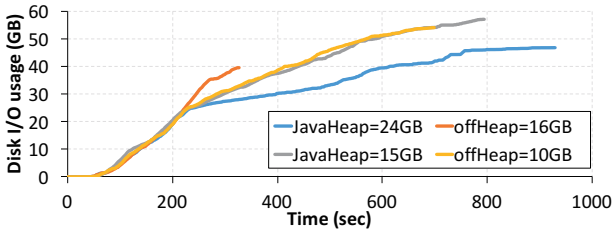

図 22 ヒープ設定ごとのディスク書き出しバイト数の時間総量

総量は最終的にヒープの大きさを狭めたほうが大きいものの，全体の処理時間はオフヒープを大きくしたほうが短くなっている．これは spill 発生があるかないかの差が原因となっていると予想される．図 19 でも 200 秒以降，spill によって何度かのメモリ使用量の増減が発生している．一方，1.6.2 ではオフヒープ 16GB と大きく設定し， spill を抑制した場合でもあまりメモリを解放せず，バッファキャッシュのための空きメモリが少なくなってしまっていることがわかる．

Spark 2.0.1 では Whole-stage code generation により，多くのステージをひとつのステージにまとめて処理を短縮しているため，メモリ使用が一時的に増加するものの，すぐさま解放されていると予想される．この結果から，バッファキャッシュによって性能を向上させるためには処理の効率を上げてできるだけメモリを長く保持しないようにすることが重要になると考えられる．しかし，この最適化があるにも関わらず，オフヒープを利用しない場合の性能は Spark 1.6.2 とあまり変わらない．

図 21 は Spark 2.0.1 のときのディスク読み出しの時間経過を示している．Spark 1.6.2 と比較して，Java ヒープが 24GB の時のディスク読み出し量が 5GB 程度削減できている．これはメモリ使用量が削減され，結果的にバッファキャッシュが 10GB 近く増加したことが原因となっている．図 22 のディスク書き出しの傾向を見ると，全体的に Spark 2.0.1 のほうが書き出し量は増えていることがわかる．例外的に，最も性能の良いオフヒープが十分な状況ではディスク書き出し量にあまり変化が見られない．その結果，全体の所要時間にも影響を与えていることが予想できる．

以上，まとめると 2 節の疑問 (3) の オフヒープを利用した場合のバッファキャッシュ利用の特性については次のようになる：オフヒープを利用した場合，Spark はより積極的にメモリを解放するようになる．その結果，一時的にバッファキャッシュの使用を増加させることができる．Spark 1.6.2 のデフォルト設定となっている，ヒープ最小使用量の設定のために JVM がほとんど OS にメモリを返さないこともバッファキャッシュの利用効率を下げている．

(4) Spark のバージョンによってシャッフルの性能は変化するのか？については，Spark 2.0.1 はシャッフル処理が最適化された結果，メモリ使用量は一時的に大きく増えてしまうものの，すぐに解放される．その結果，(3) の効果を引き起こすことが確認されている．

6. 関連研究

Spark における TPC-H ベンチマーク性能特性は千葉ら [9] が行っている．特に，JVM の設定および NUMA アーキテクチャ環境下の OS 設定に関する性能特性の分析結果を報告している．本研究は分析をシャッフル処理に集中してより細かい分析を行っている．

Ousterhout らによる Spark の性能分析 [7] は，block time に基づく分析により，CPU がボトルネックになりつつあることを示している．この研究の前提とする環境は，十分 Spark が長時間動作し，メモリなどが潤沢な環境を想定している．本研究は Spark が起動した直後の状態で，かつメモリの圧迫が厳しい状況を想定している点で異なる．Block time の算出方法も厳密には Spark がトレースを取れる範囲での性能であり，2 節で示したようにシステムレベルで得られるメトリクスとは異なることもわかっている．

Chen ら [10] は Hadoop MapReduce の商用のワークロードについて大規模な実態調査を行っている．データアクセスパターンやスキュー，データ局所性など MapReduce の様々な指標について，商用で利用されたワークロードのトレースを用いて調査している．本研究は Spark のシャッフル処理について対象を絞り，ディスク I/O やメモリ使用などのより細かいシステムレベルの指標について分析をしている点で異なる．

Harter らは一般のデスクトップアプリケーションのディスク I/O 特性 [11] および，HBase の性能特性 [12] を分析している．それぞれファイルのアクセスパターンなど，システムレベルの指標について詳細な分析を行っている．しかし，本研究と異なりバッファキャッシュに関連した分析

はなされていない．

7. 結論

本研究では Spark のシャッフル処理について，システムレベルで得られる情報をもとに分析を行った．特に，バッファキャッシュによる性能向上について注目し，TPC-H ベンチマークについて特にクエリ 9 の分析を行った．4 種類の Java ヒープの設定を行って分析をした結果，シャッフル処理においてバッファキャッシュの利用が性能を大きく改善する場合があることを示した．ただし，本研究の結果はあくまでこのケースの限りであることに注意したい．Spark のシャッフル処理の最適化に向けて，本研究からさらに Spark とバッファキャッシュの関係について詳細に分析を進めていくこと重要になると考えている．

参考文献

[1] 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)* (2010).

[2] room, I. N.: IBM Launches Industry's First Development Environment for Apache Spark - Delivered in the Cloud for Rapid Adoption, https://www-03.ibm.com/press/us/en/pressrelease/49866.wss (2016).

[3] Davidson, A. and Or, A.: Optimizing shuffle performance in spark, Technical report, University of California, Berkeley - Department of Electrical Engineering and Computer Sciences (2013).

[4] Daikoku, H., Kawashima, H. and Tatebe, O.: On Exploring Efficient Shuffle Design for In-memory MapReduce, *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR '16)*, pp. 6:1–6:10 (2016).

[5] Nicolae, B., Costa, C., Misale, C., Katrinis, K. and Park, Y.: Towards Memory-Optimized Data Shuffling Patterns for Big Data Analytics, *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '16)*, pp. 409–412 (2016).

[6] Xin, R. and Rosen, J.: Project Tungsten: Bringing Apache Spark Closer to Bare Metal, https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html (2015).

[7] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S. and Chun, B.-G.: Making Sense of Performance in Data Analytics Frameworks, *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI '15)*, pp. 293–307 (2015).

[8] Michael Armbrust, P. W. and Xin, R.: Announcing Apache Spark 1.6, https://databricks.com/blog/2016/01/04/announcing-apache-spark-1-6.html (2016).

[9] 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 '16)*, pp. 112–121 (2016).

[10] Chen, Y., Alspaugh, S. and Katz, R.: Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads, *Proceedings of VLDB Endowment*, Vol. 5, No. 12, pp. 1802–1813 (2012).

[11] Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pp. 71–83 (2011).

[12] Harter, T., Borthakur, D., Dong, S., Aiyer, A., Tang, L., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Analysis of HDFS Under HBase: A Facebook Messages Case Study, *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*, pp. 199–212 (2014).