Sparkにおけるディスクを用いたRDDキャッシングの高速化と効果的な利用に関する検討

張　凱輝[†,††]　谷村　勇輔[††,†]　中田　秀基[††,†]　小川　宏高[††]

† 筑波大学　〒305–8577 茨城県つくば市天王台 1-1-1
†† 産業技術総合研究所　〒305–8560 茨城県つくば市梅園 1-1-1

あらまし　Sparkは機械学習やデータマイニングなどの反復計算を高速に実行できる並列データ処理フレームワークである．RDD（Resilient Distributed Dataset）と呼ばれる仕組みを利用してインメモリでの並列処理や中間データのキャッシング，耐障害性の確保を実現している点に特徴がある．扱うデータが大きくメモリ容量が 不足する場合には，一部または全部のデータを処理ノードのディスクに置いて処理を行うことも可能である．しかし，ディスクを用いることにより，Sparkアプリケーションの実行性能が低下する可能性がある上，このディスク利用の有無をユーザが指示しないといけない問題がある．そこで本研究では，ディスク利用時のRDDキャッシングの効果と性能を調査し，ディスクを効果的に利用するために，メモリとディスクを併用する方式において明らかになった性能低下の問題の解決を試みた．また，ディスクを利用したキャッシング操作自体の高速化を検討するため，性能の異なるディスクを用いた場合の性能比較を行った．これらの結果により，ディスクを用いたRDDキャッシングを利用する際の問題点や注意点を明らかにした．

キーワード　ビッグデータ，Spark，RDDキャッシング，ディスクI/O，性能解析

Performance Improvement and Effective Use of Disk-based RDD caching in Spark

Kaihui ZHANG[†,††], Yusuke TANIMURA[††,†], Hidemoto NAKADA[††,†], and Hirotaka OGAWA[††]

† University of Tsukuba　Tennoudai 1–1–1, Tsukuba, Ibaraki, 305–8577 Japan
†† National Institute of Advanced Industrial Science and Technology　Umezono 1–1–1, Tsukuba, Ibaraki, 305–8560 Japan

Abstract　Spark is a parallel data processing framework that can perform iterative calculation, such as machine learning and data mining, in high speed. Spark achieves parallel processing and fault tolerance by using a mechanism called RDD (Resilient Distributed Dataset) and RDD caching allows the Spark programs to reuse intermediate data. When data to be cached is too large to hold in memory of the Spark nodes, some or all of the data can be placed on disks of the nodes. However, use of the disks might degrade execution performance of the Spark applications and it is also a problem that the application users must instruct whether or not the disks are used for caching. In this study, execution performance of the Spark application using disks for caching was investigated by comparing the cases of using different storage levels and disks of different performance. This report summarizes insights form the results for improving I/O performance of the RDD caching using disks and guidelines of when to use disks and we improved the execution performance of the benchmark by the proposed method.

Key words　Big Data, Spark, RDD Caching, Disk I/O, Performance Analysis

1. はじめに

Apache Spark（以下，Spark） [1] は機械学習やデータマイニングを高速に実行できるオープンソースの並列データ処理フレームワークとして，ビッグデータ解析や人工知能分野にお

NOTICE: xSIG 2017 does not publish any proceedings, and this manuscript is provided only to the xSIG 2017 attendees during the workshop. Thus, the TPC expects that acceptance in xSIG 2017 should not preclude subsequent publication in conferences or journals.

いて高い注目を集めている．Spark は Hadoop [2] が実装する MapReduce [3] と同様の利点を持つ一方，中間データを HDFS（Hadoop Distributed File System）に置くのではなく，ワーカノードのメモリやディスクに保持するなどの工夫により，反復操作や対話処理において Hadoop より優れた性能を提供する．Spark において各データは RDD（Resilient Distributed Dataset) として抽象化され，それぞれの中間データも RDD の１つとして扱われる．Spark の主な処理はインメモリで行われるが，シャッフル操作の一部やユーザが指定した場合にワーカノードのディスクが用いられる．例えば，RDD を再利用や障害対策のためにキャッシュしておく場合に，メモリではなくディスクを選択することが可能である．しかし，ディスクの利用は Spark アプリケーションの実行性能の低下を招く恐れがあり，慎重に行う必要がある．その一方で，ディスク利用の選択はユーザに委ねられており，ディスク利用の是非を適切に判断することが容易ではないという問題がある．

我々はこれまでの研究 [4,5] において，RDD キャッシングやチェックポイントにおけるストレージへの書き込み性能の調査を行ってきた．本論文ではそれをさらに発展させ，機械学習のプログラムをベンチマークとして用い，RDD キャッシングにおけるディスク利用が Spark アプリケーションの実行性能に与える影響について調査を行った結果を報告する．具体的には，RDD キャッシングのストレージレベルの違いによる実行性能の比較，メモリとディスクを併用してキャッシングを行う方式の性能調査と改善，4 種類のストレージデバイスを用いた場合のキャッシングの性能比較である．これらの結果をもとに，RDD キャッシングにディスクを利用する際の問題点や注意点を明らかにし，高速化の可能性についてまとめた．

2. Spark の RDD キャッシングの仕組み

2.1 Spark の概要

Spark [1] は University of California, Berkeley の AMPLab（Algorithms, Machines, and People Lab）の研究プロジェクトにより開発が始まったオープンソースの並列データ処理フレームワークである．Hadoop と同様に MapReduce のアルゴリズムに基づいた分散処理が可能であり，コンピューティングとストレージの両機能を持つノードからなるクラスタで実行可能である．Spark は Hadoop MapReduce の利点に加えて，中間データをメモリに保持することで，機械学習やデータマイニングなど反復操作を必要とするアプリケーションを効率良く実行できる特徴を持つ．全体の計算量に対して反復計算が占める割合が大きいほど，Spark を利用するメリットは大きくなる．

Spark は Scala や Java，Python など複数のプログラミング言語をサポートし，Spark-Shell を用いた対話型計算，YARN などの資源管理フレームワークを用いた一括実行が可能である．また，Hadoop エコシステムとの親和性が高く，Hive，HBase，HDFS などと組み合わせて利用することもできる．

2.2 Spark の動作と RDD

Spark では RDD（Resilient Distributed Datasets）と呼ばれる，読み取り専用の分散データ構造が用いられる．RDD は

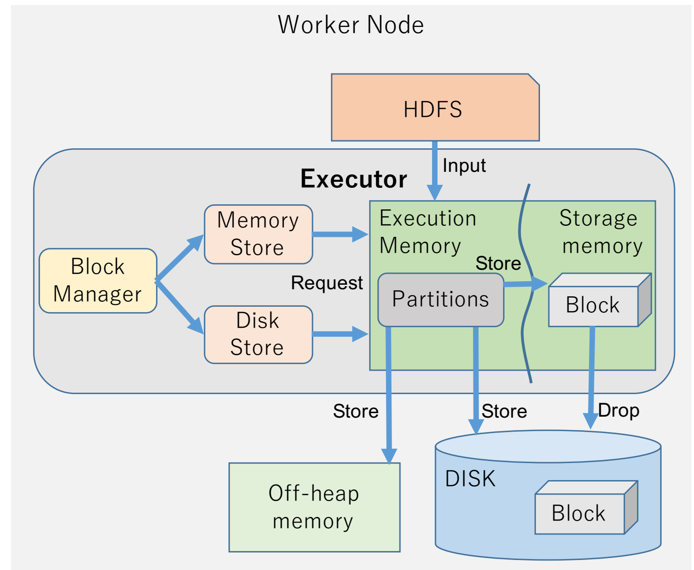

図 1　Spark の Executor の構成

内部的にパーティションに分割され，各パーティションをデータ処理の単位として分散並列実行が可能である．各 RDD に対しては map，flatMap，filter，join などのデータ操作を適用できる．ただし，変換操作（map，filter など）はすぐに処理が実行されるわけでなく，アクション操作（count，first など）が呼ばれてからジョブとして投入され，データのロードおよび処理が開始される．これは遅延処理と呼ばれる．

Spark における耐障害性の実現方法は RDD の保存（キャッシングまたはチェックポイント）と再構築の２つからなる．RDD を生成したノードは，自身が生成したパーティションを明示的にメモリあるいはディスクに保存することができる．Spark ではこれを RDD キャッシングと呼ぶ．RDD キャッシングが実行された RDD を保持するノードに障害が発生した場合は，Spark は RDD の生成手順を記録した「Lineage」を参照し，失われたデータのパーティションのみを再度実体化するようにタスクをスケジューリングすることで耐障害性を実現している．またノードに障害があっても全体の処理速度の低下を引き起こさないように，データを複数のノードに複製しておくこともできる．

RDD はアクションが実行されるたびに計算し直されるのが基本的な動作であるため，同じ RDD に対して複数の操作を適用する場合に，速やかに次の結果を得るためにもキャッシュされた RDD が活用される．

2.3 RDD キャッシングの仕組み

RDD キャッシングの実行は，RDD に対して persist() メソッドを呼び出し，引数にストレージレベルを指定することで行う．デフォルトの指定はメモリであるが、ディスク，あるいはメモリとディスクを併用する方式などを指定可能である．

図 1 は Spark のワーカノードで動く Executor の構成を示した図である．Executor のメモリは，MemoryManager によって内部的に ExecutionMemory と StorageMemory の 2 つに動的に分割されて管理されている．ExecutionMemory が RDD を

処理するのに用いられる一方，StorageMemory は BlockManager により管理され，RDD パーティションを Block という形で保存するのに用いられる．つまり，RDD キャッシュをメモリに保存する場合は StorageMemory が利用される．

メモリを使用するストレージレベルが設定されていても，RDD はパーティションに分割されているため，RDD の全てがメモリに保存されるわけではない．メモリ内の各 RDD パーティションに対しては Least-Recently-Used（LRU）規則が適用され，空きメモリに収まらない RDD のキャッシングが行われる際，参照されていない時間が最も長いパーティションは順にドロップされる．この時，メモリのみを使用するストレージレベルが設定されている場合は該当パーティションは失われるので，再度該当パーティションが必要になった際には再計算を行うことになる．メモリとディスクを併用するストレージレベルが設定されている場合には，該当パーティションはディスクに書き出され，再度の参照が可能である．なお，ユーザは unpersist() メソッドを明示的に呼ぶことにより，利用されていない RDD をキャッシュから削除することができる．

以下に RDD キャッシングを行う際に指定できる主なストレージレベルとその挙動を説明する．

NONE (NOCACHE): RDD キャッシングを行わない方式である．生成した RDD を使い捨てるため，再度同じ RDD が参照される場合は再処理が必要になる．

MEMORY_ONLY: RDD キャッシュをメモリのみで保持する方式である．RDD キャッシングが実行されると，BlockManager は MemoryStore のインタフェースを通して StorageMemory に必要分のメモリを確保し，RDD パーティションを Block という形で保存する．メモリスペースを節約するため，シリアライズしてメモリに保持する MEMORY_ONLY_SER と呼ばれる方式もある．

OFF_HEAP: RDD キャッシュを Off-Heap メモリのみで保持する方式である．RDD キャッシングが実行されると，BlockManager はメモリ上のパーティションをシリアライズし，MemoryStore のインタフェースを通して Off-Heap メモリに保存する．Off-Heap メモリを使うことで GC の影響を削減できる利点がある．

DISK_ONLY: RDD キャッシュをディスクのみで保持する方式である．RDD キャッシングが実行されると，BlockManager はメモリ上のパーティションをシリアライズし，DiskStore のインタフェースを通してローカルディスクに保存する．

MEMORY_AND_DISK: メモリとディスクを併用して RDD キャッシュを保持する方式である．メモリを優先的に利用し，まずは StorageMemory 内に Block として保存することを試みるが，メモリに収まりきらない場合は LRU に基づき，新しい Block が必要とするメモリ量を賄うのに必要なだけの古い Block をシリアライズしてディスクに移動する．ディスクに保存された RDD を参照する際は，BlockManager が利用可能な StorageMemory と StorageMemory 上に該当データを展開するのに必要な容量を確認し，ディスクから Block を読み込み，StorageMemory に戻す操作を行う．本方式の利点はメモリかディスクの選択をユーザ側に求めず，システム側で自動的に選択するという点である．

表 1　評価実験に用いたマシンのスペック

CPU	Intel Xeon CPU E5-2620v3 2.40GHz, 6 cores x2
Memory	128 GB
Network	10 Gbps (for HDFS connection)
NVMe-SSD	Intel SSD DC P3700
SSD	OCZ Vertex3 (240GB, SATA6G I/F)
HDD	Hitachi Travelstar 7K320 (SATA3G I/F)
OS	Ubuntu 14.04 (Kernel v.3.13)
File System	Ext4

3. 評価実験

3.1 評価実験の方法と環境

本研究では先にも述べたように，ディスクを用いた RDD キャッシングを利用した際の Spark アプリケーションの実行性能を調査し，ディスクを用いた RDD キャッシングの高速化と効果的な利用方法を明らかにすることを目的としている．そのため，以下に述べる 3 種類の評価実験を行なった．最初の評価実験では RDD キャッシングの有無やストレージレベルを変えて性能測定を行い，ディスクを用いた RDD キャッシングの結果と比較した．次の評価実験ではメモリとディスクを併用する方式において，メモリからディスクへのドロップが発生する状況下の性能を測定し，Spark が効果的に両者を利用できるかを調査した．最後の評価実験では I/O 性能の異なる複数のストレージデバイスを用いて，ディスクを用いた RDD キャッシング操作自体に要する時間を測定し，ストレージデバイスの高速化による性能向上の効果を調査した．

評価用のプログラムとしては，最初の 2 つの評価実験には，Spark の機械学習ライブラリ（MLlib）[6] に含まれる SparseNaiveBayes と DenseKMeans のベンチマークを用いた．各ベンチマークの入力データは HiBench(6.0) [7] を用いて生成し，データサイズには large を指定した．最後の評価実験には，Spark のワーカ処理において任意のサイズの RDD を生成し，RDD キャッシングを行うプログラム（RDDTest）を用いた．

評価実験には表 1 に示すマシンを用い，ローカルモードで Spark の各ベンチマークプログラムを実行した．本来，Spark は分散環境で実行するものであるが，本評価実験は各ワーカノードの RDD キャッシングの性能にのみ着目するため，実験環境を簡略化して 1 ノードで実験を行っている．RDD キャッシュにディスクを利用する場合には，RAM ディスク，NVMe 接続の SSD，SATA 接続の SSD および HDD の 4 種類を利用した．Spark は v.2.1.0 を用い，Scala v.2.10.6 および Java v.1.8.0_66 を用いてビルドし，実行した．用いた Spark は，GC の影響緩和のための独自のメモリ管理機構の導入やデータがキャッシュに乗ることを意識したデータ構造やアルゴリズムの導入など Project Tungsten [8] の成果を含んだバージョンである．

3.2 評価実験に用いたベンチマークの詳細

SparseNaiveBayes ベンチマークのデータフローは図 2 の通

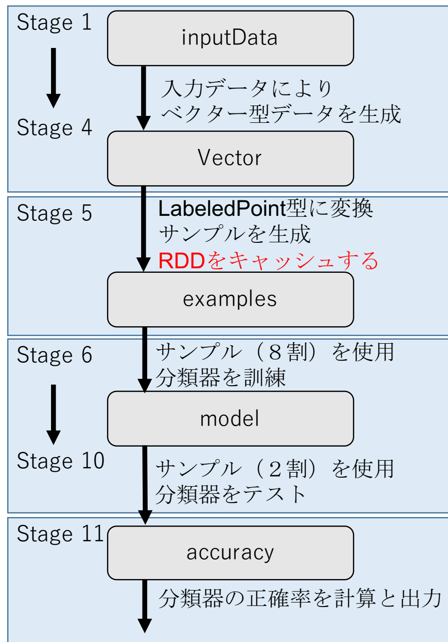

図 2　SparseNaiveBayes の各実行ステージにおける処理内容

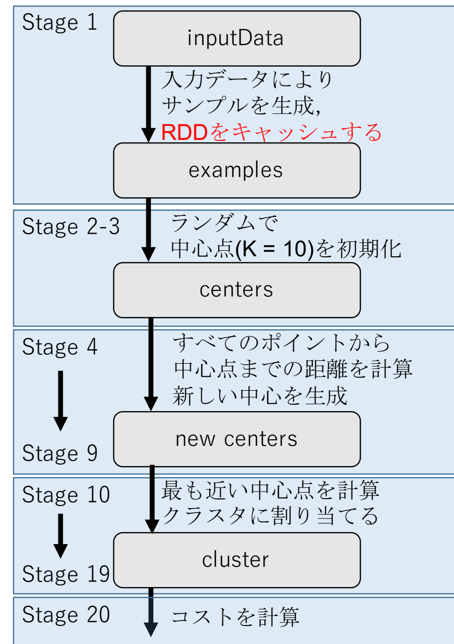

図 3　DenseKMeans の各実行ステージにおける処理内容

りである．最初に入力データよりベクター形式のデータを生成し，LabeledPoint 型のサンプルデータに変換する．そして，サンプルデータに RDD キャッシングを適用する．その後，サンプルデータの 8 割を用いて分類器を訓練し，2 割を用いてテストを行い，正解率を計算し，出力する．評価実験に用いた入力データのサイズは 370MB，RDD サイズ（中間データ）は 788MB である．

DenseKMeans ベンチマークの図 3 の通りである．最初に入力データより，サンプルデータを生成し，これに RDD キャッシングを適用する．次にランダムに k 個の中心点（今回は k=10）を初期化する．その後，全てのポイントから中心点までの距離を計算し，新しい中心を生成する処理を繰り返す．最後に得られた結果の SSE（Sum of Squared Errors）を計算し，出力する．評価実験に用いた入力データのサイズは 4015MB，RDD サイズ（中間データ）は 8546MB である．

3.3　ディスクを用いた RDD キャッシングの効果

RDD キャッシュの効果とディスク利用による影響を調べるため，スレッド数を 1，Executor に割り当てるメモリを 60GB に設定し，RDD キャッシングを行わない場合と行う場合の 4 つのストレージレベルにおいて SparseNaiveBayes と DenseKMeans のベンチマークを実行した．DISK_ONLY の実験では SATA 接続の SSD を用いた．図 4 は各ストレージレベルについて，SparseNaiveBayes のステージ毎の実行時間を比較した結果である．RDD キャッシングを行わない場合，ステージ 5 を高速に実行できるが，ステージ 6 以降は中間データの再生成が必要になるため，実行時間が長くなっているのが分かる．RDD キャッシングの重要性を示した結果といえる．MEMORY_ONLY，MEMORY_ONLY_SER，OFF_HEAP と DISK_ONLY の比較では，ステージ 5 の実行において性能差が見られるが，全体的な実行時間の差は小さい．RDD キャッシングにディスクを利用することへの影響は少ないといえる．

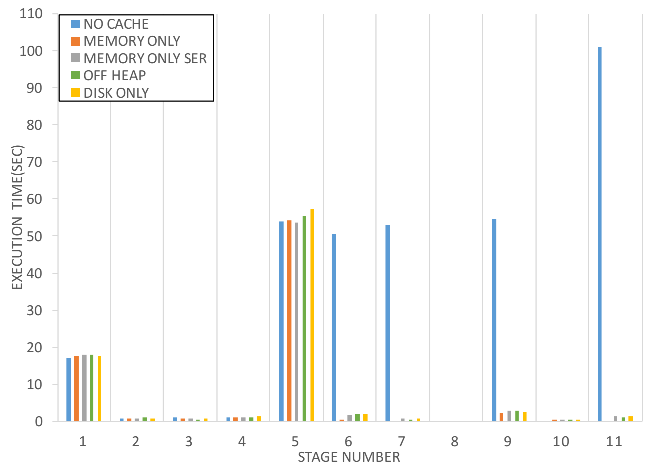

図 4　ディスクを用いた RDD キャッシングの効果（SparseNaive-Bayes）

同様に，図 5 は各ストレージレベルについて，DenseKMeans のステージ毎の実行時間を示した結果である．RDD キャッシングを行わない場合，ステージ 1 では RDD キャッシングを行う場合よりも短い時間で処理を完了しているが，以降のステージではより長い時間を要しているのが分かる．逆に，RDD キャッシュを行う場合にはステージ 1 には時間がかかるが，以降のステージでは短い時間で処理が完了している．MEMORY_ONLY，MEMORY_ONLY_SER，OFF_HEAP と DISK_ONLY の比較では，MEMORY_ONLY が他のストレージレベルより良い結果となり，実行時間の差は倍以上となった．MEMORY_ONLY_SER や OFF_HEAP の結果は DISK_ONLY と同等であり，DenseKMeans においては中間データのシリアライズがボトルネックの多くを占めているといえる．

SparseNaiveBayes と DenseKMeans の結果を比べると，後

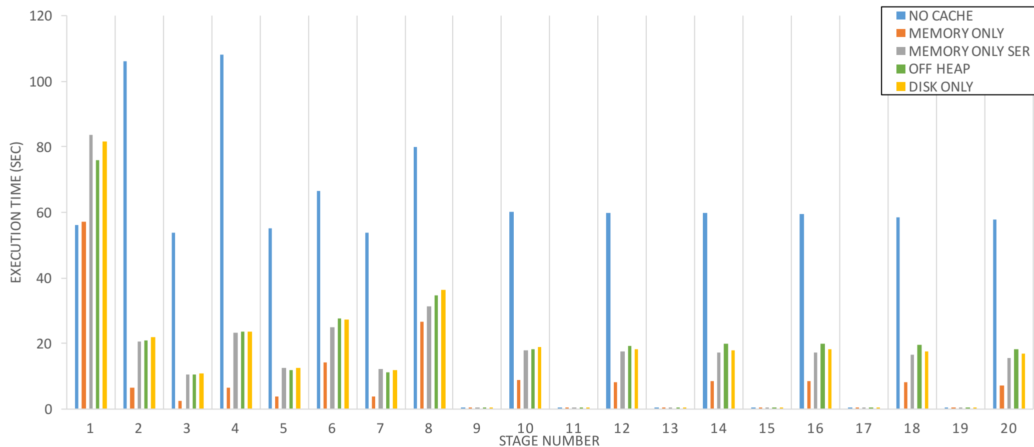

図 5　ディスクを用いた RDD キャッシンングの効果（DenseKMeans）

者の方が生成される RDD の合計サイズが 10 倍以上大きく，また各ステージの実行に時間を要しているため，RDD キャッシングの効果がより大きく表れたといえる．

3.4　メモリとディスクの併用による影響

3.4.1　実行時間とディスクへのドロップの関係

RDD キャッシングにおいてメモリとディスクの併用による影響を調べるため，ストレージレベルを MEMORY_AND_DISK に設定し，Executor に割り当てるメモリ量を 3 つのパターン（512MB，1GB，2GB）に設定して，MEMOLY_ONLY，OFF_HEAP および DISK_ONLY の場合を比較した．MEMORY_ONLY と DISK_ONLY の場合，割り当てるメモリ量は 60GB に設定した．ただし，DenseKMeans の DISK_ONLY においては 512MB だけを割り当てた場合も試した．

図 6 は SparseNaiveBayes のステージ毎の実行時間を示した結果であり，表 2 は MEMORY_AND_DISK の各実行において MemoryStore からドロップしたブロック数とその合計サイズを示した表である．表 2 から分かるように，MEMORY_AND_DISK_2G ではドロップが発生していないため，図 6 の MEMORY_ONLY と同様に，ステージ 6 以降は他の MEMORY_AND_DISK の結果より高速に実行できたと考えられる．MEMORY_AND_DISK_512M と DISK_ONLY はほぼ同等の結果を示しており，MEMORY_AND_DISK におけるドロップの発生による遅延はほとんど見られなかったと考えることができる．

表 2　MemoryStore からのドロップの発生頻度と量（SparseNaiveBayes）

	512MB	1024MB	2048MB
#Blocks	70	59	0
Size(MB)	1,508	1,183	0

同様に，DenseKMeans の結果を示したのが図 7 および表 3 である．SparseNaiveBayes とは異なり，MEMORY_AND_DISK_512M が DISK_ONLY に比べて倍以上の処理時間を要しており，MEMORY_AND_DISK の実行性能が非常に悪い結果となった．MemoryStore からのドロップの状況を確認すると，MEMORY_AND_DISK_512M は MEMORY_AND_DISK_1G や MEMORY_AND_DISK_2G に比べてメモリからドロップしていたブロック数が少ないものの，ガベージコレクションの発生率が高くなっており，それが性能低下の原因であると考えられる．MEMORY_AND_DISK_1G と MEMORY_AND_DISK_2G はトロップの発生回数が多く，サイズも非常に大きい．その原因はドロップされたブロックが再び MemoryStore に戻され，再度ドロップされる状況になっていたためである．

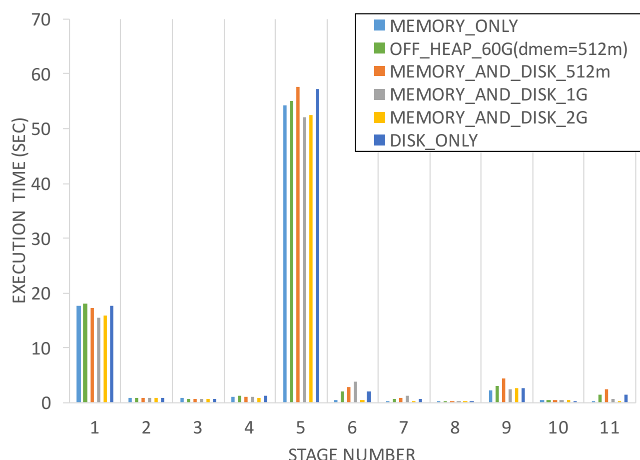

図 6　メモリとディスクの併用による影響（SparseNaiveBayes）

OFF_HEAP はドロップの発生頻度が少ないために MEMORY_AND_DISK_512M より良い性能を示したが，シリアライズのオーバーヘッドがあり，DISK_ONLY（dmem=512m）に近い結果となった．

表 3　MemoryStore からのドロップの発生頻度と量 (DenseKmeans)

	MEMORY_AND_DISK			OFF_HEAP
Driver memory (MB)	512	1024	2048	512
#Blocks	600	746	715	305
Size(MB)	2,577	26,793	20,936	1,332

3.4.2　同じブロックに対する再トロップの抑制

前節で述べたように，Executor へのメモリ割り当てが不十

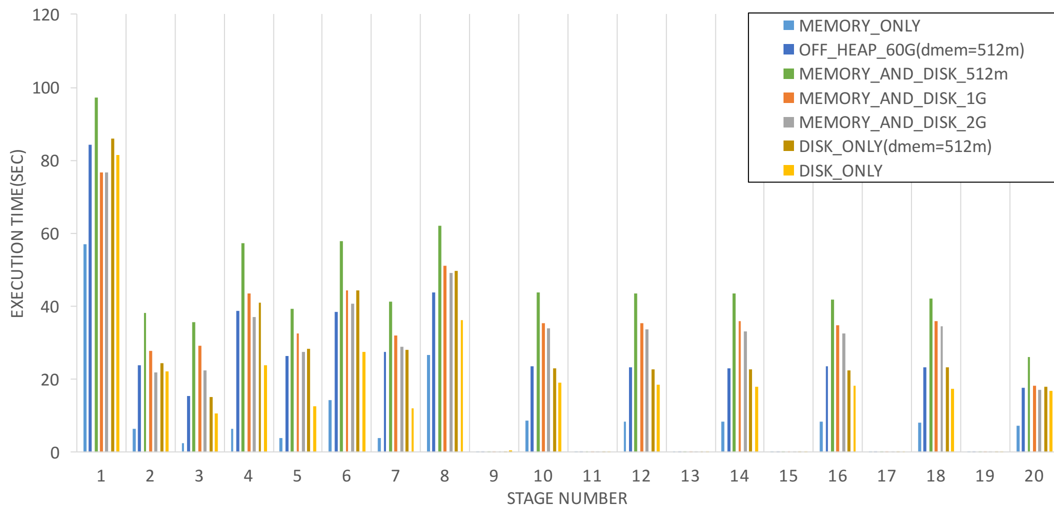

図 7　メモリとディスクの併用による影響（DenseKMeans）

分な場合，RDD キャッシュを構成するブロックが繰り返し MemoryStore に入る状況が発生しうることが確認できた．この問題を解決するため，一旦ディスクにトロップされたブロックについては MemoryStore に戻さず，以降はそのままディスクに保持したままとする修正を加えた．

表 4 は前節の DenseKmeans と同じ実験について，修正前（Original）と修正後（Modified）の実行時間を比較した結果である．Modified の実行時間は Original の 6 割程度まで短縮し，再ドロップを抑制することで性能が向上することが確認できた．表 5 は Original と Modified のトロップの発生回数とドロップされたブロックの合計サイズを示した結果である．再ドロップの抑制によってドロップの発生回数と合計サイズが大きく削減できているのが分かる．

表 4　再ドロップの抑制による実行時間の短縮

	Original	Modified
MEMORY_AND_DISK_512MB	670 s	422 s
MEMORY_AND_DISK_1G	533 s	331 s
MEMORY_AND_DISK_2G	489 s	318 s

3. 5　ストレージデバイスによる違い

ディスクを利用して RDD キャッシュを保持する際のストレージデバイスによる性能の違いを調査するため，RDDTest を用いて 1,000MB の RDD を RDD キャッシュにする実験を行った．実行するスレッド数は 1 である．図 8 は RDDTest の結果と Fio ベンチマーク [9] を用いて各ストレージデバイスの性能を測定した結果を比較した表である．この結果より，各ストレージデバイスの性能差が RDD キャッシュを生成する性能差にあまり影響を与えていないことが分かる．RAM ディスクや NVMe 接続の SSD のようにストレージデバイスが高速である場合にはシリアライズ処理がボトルネックとなり，デバイスの性能を活かすことができていない．また，HDD のようにストレージデバイスが低速であっても，OS のバッファキャッシュの効果により RDD キャッシュの生成性能への影響が小さくなっている．

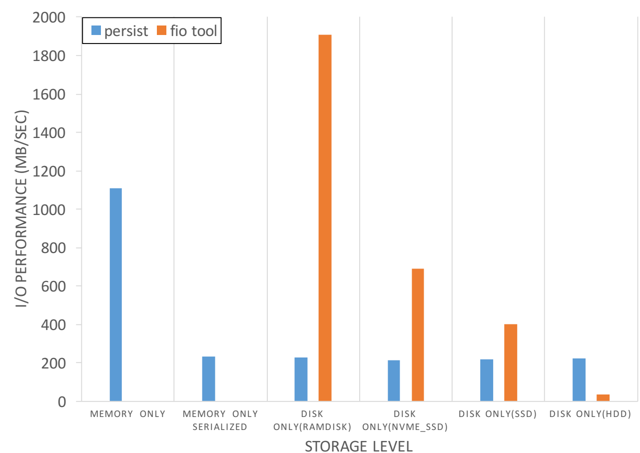

図 8　RDD キャッシュを生成する速度の比較（1 スレッド）

図 9 は同じ RDDTest を用いて，実行するスレッド数を 1, 4, 8, 12 と変化させた結果である．また，各スレッドの RDD のサイズも 500MB，1,000MB，1,500MB，2,000MB と変化させた．図 9 より RDD サイズおよびスレッド数を増やすことにより，OS のバッファキャッシュの効果が小さくなっていることが確認できる．RDD サイズの増加に対して，RAM ディスクと NVMe 接続の SSD の結果は RDD キャッシュを生成の I/O 性能を維持しているが，HDD ではスレッド数が 12，RDD サイズが 2,000MB より RDD キャッシュ生成の I/O 性能の低下が見られる．これらの結果より，複数のタスクを同時に処理して RDD キャッシュ生成される時に初めてディスク性能が重要になってくることが分かる．

4.　議　　論

本節では 3. 節の結果をもとに，RDD キャッシングにおけるディスク利用の指針と高速化について検討した結果を述べる．

RDD キャッシングの効果はキャッシュの生成時間と RDD の再生成がトレードオフの関係にあるが，今回試したベンチマークにおいては再生成のコストが圧倒的に高く，キャッシングが非常に有効であった．一方，メモリにキャッシュする方式

表 5　再ドロップを抑制した際のドロップの発生回数と合計サイズ

	Original		Modified	
	#Blocks	Size(MB)	#Blocks	Size(MB)
MEMORY_AND_DISK_512MB	660	2,577	133	610
MEMORY_AND_DISK_1G	746	26,793	33	486
MEMORY_AND_DISK_2G	751	20,936	58	473

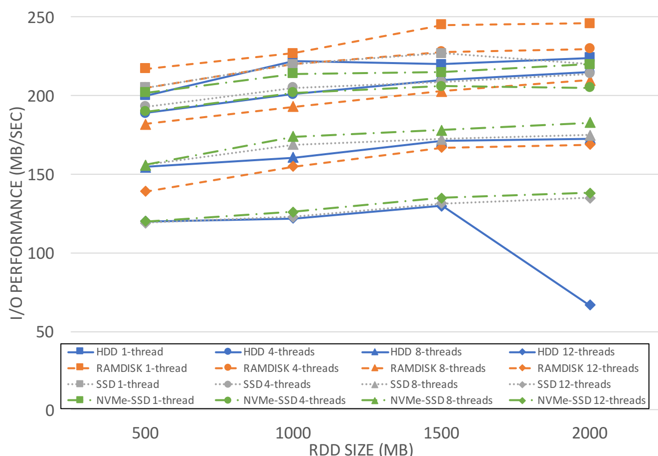

図 9　RDD キャッシュの生成速度の比較（複数スレッド）

とディスクにキャッシュする方式では，アプリケーションの全体の実行時間に比べてキャッシュを生成する時間差はわずかであり，ディスク利用は有効な選択肢の１つである．

メモリとディスクを併用する方式はシステム側で両者を使い分けるためにユーザにとって利便性が高いが，メモリが不足する状況において，同じ RDD キャッシュのディスクへのドロップが繰り返されることにより，ディスクのみを用いてキャッシュする方式より実行性能が低下する現象が見られた．一方，再ドロップを抑制する修正を施すことにより，その問題が解決できることも確認できた．

Off-Heap メモリの利用はガベージコレクションの影響を削減できる利点があるが，シリアライズのオーバーヘッドが大きく，今回の評価実験ではディスクのみを用いてキャッシュする方式に対する優位性を確認することができなかった．なお，Executor に割り当てるメモリが十分でない場合にどのような GC アルゴリズムを利用するべきか，また GC アルゴリズムに与えるパラメータによってアプリケーションの実行性能が大きく変わり得るかなどの調査は Spark の一般的な利用において重要であり，その結果が今回の評価実験結果にも適用されるのかは今後検討を進めていきたい．

RDD キャッシングにおけるディスク利用については，OS のバッファキャッシュがあるためにディスクの性能はボトルネックになりにくい．１つのワーカノードにおいて複数のタスクを並列に実行し，各タスクが同時に RDD キャッシュを生成する場合やメモリの空きが少ない場合においてのみディスク性能の影響がアプリケーションの実行性能に現れると考えられる．それ以外のケースでは，高速なディスクよりも容量の大きい HDD などを用いた方がコストパフォーマンスに優れている．また，RDD キャッシュの生成を高速化するには，高速なディスクを用いてもあまり効果がないため，シリアライズなど Spark 内部の実装の改善が必要であると考える．

5. 関 連 研 究

Spark においてメモリと CPU 利用の効率化を図る試みの１つに Project Tungsten [8] がある．JVM のオブジェクトモデルやガベージコレクションのオーバヘッドの削減，オフヒープの実装，シリアライズの高速化等により RDD API や DataFrame を用いた Spark プログラム，および Spark SQL の実行速度の改善を実現している．また，RDD キャッシュ用のメモリが不足する際にドロップする RDD を選択する方法については，RDD パーティションの生成コストを考慮した Weight Replacement アルゴリズムの研究が行われている [10]．これらの研究では RDD をメモリにのみキャッシュする方式に焦点が当てられているのに対し，本研究ではディスクを利用した RDD キャッシングの性能解析や高速化を目的としている点に違いがある．

J. Shi らは Spark のタスク実行をプロファイリングして性能解析を行っており，RDD キャッシングの効果やストレージレベルの影響，ボトルネックの所在について調査を行っている [11]．K-means 実行時の RDD キャッシングの効果に関する調査など本研究と類似する取り組みが見られるが，Hadoop MapReduce と Spark の性能差の理由や耐障害性をアーキテクチャや実装の観点から明らかにすることに主眼が置かれている．

6. お わ り に

本論文では Spark においてディスクを用いた RDD キャッシングについて調査を行い，ディスク利用が与える影響やディスクの性能による違いを明らかにし，ユーザがディスク利用を選択する指針やキャッシュの高速化について検討した結果を述べた．今後の課題としては，MLlib のベンチマークだけでなく他のアプリケーションにも適用して，今回得られた指針を一般化することやより大きなデータセット，クラスタ環境を用いての評価実験，RDD キャッシュの生成性能の改善などが挙げられる．

謝　　辞

この成果の一部は、国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られたものです。本研究は JSPS 科研費 JP16K00116 の助成を受けたものです。

文　　献

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M.

McCauly, M.J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12), pp.15–28, 2012.

[2] "Apache Hadoop". http://hadoop.apache.org/

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04), pp.1–13, 2004.

[4] 谷村勇輔，小川宏高，"Spark RDD のストレージ出力に関する性能評価，" 情報処理学会研究報告，第 2016-HPC-153 巻，pp.1–5，2016.

[5] 谷村勇輔，小川宏高，"Spark における中間データ用ローカルストレージの構成方式の検討，" 2016 年ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集，p.54，2016.

[6] X. Meng and etal., "MLlib: Machine Learning in Apache Spark," Machine Learning Research, vol.17, no.1, pp.1235–1241, 2016.

[7] "HiBench Suite". https://github.com/intel-hadoop/HiBench

[8] "Project Tungsten (SPARK-7075)". https://issues.apache.org/jira/browse/SPARK-7075

[9] "Fio (Flexible I/O Tester)". https://github.com/axboe/fio

[10] M. Duan, K. Li, Z. Tang, G. Xiao, and K. Li, "Selection and Replacement Algorithms for Memory Performance Improvement in Spark," Concurrency and Computation: Practice and Experience, vol.28, no.8, pp.2473–2486, 2013.

[11] J. Shi, Y. Qiu, U.F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics," Proceedings of the VLDB Endowment, vol.8, pp.2110–2121, 2015.