Vol.2018-OS-142 No.14
2018/2/28

Apache Sparkのデータ一時保存手法の動的制御方式

藤井 智幸[1]　稲垣 英夫[1]　川島 龍太[1]　松尾 啓志[1]

概要：Apache Spark は入力データや中間データを一時保存するが，保存先としてメモリのみ，ディスクのみ，両者を併用する三つの選択肢がある．計算機クラスタの台数およびメモリ使用量が限界に達している状況下では，一時保存先としてディスクのみを使用すると最も効率的であることが予備評価で分かった．これは，メモリとディスクを併用する場合，メモリからディスクへのデータのドロップが頻発し，一時保存データがメモリとディスク間を頻繁に移動するスラッシング状態になることが原因である．単一データセットの場合はスラッシングへの対策が行われているが，これを複数データセットのワークロードに単純に適用することはできない．そこで本研究では複数のデータセットを対象として，一時保存処理の実装をワークロードの特性に応じて動的に変更し，スラッシングを回避する手法を提案する．具体的には，一時保存データの書き込み時にドロップが頻発するワークロードでは，書き込み時の動作をメモリではなくディスクに直接書き込む動作に変更し，読み出し時のドロップが頻発するワークロードでは，ディスク中の一時保存データの読み出し時に，メモリ上の一時保存領域ではなく直接実行領域に戻すように変更する．メモリとディスク両者のドロップが多数発生するワークロードでは両者の動作をともに変更する．評価結果より，メモリとディスクを併用した既存手法と比較して KMeans を実行した際の実行時間を最大で33%削減できた．

キーワード：ビッグデータ，分散処理, Apache Spark，RDD

1. はじめに

近年，IoT（Internet of Things）の普及により，世界中で生成されるデータの量が飛躍的に増大している [1]．このような大規模データは単一の計算機では処理できないため，計算機クラスタを用いた分散処理が広く利用されている．Apache Spark[2] は，計算機間の通信処理や障害発生時の処理を簡単に記述できる分散処理フレームワークであり，中間データをインメモリで高速に処理することが可能である [3]．

Spark は入力データや中間データを一時保存して再利用することで処理を効率化している．これは，機械学習やグラフ処理のように，データを繰り返し用いるワークロードで特に有効であり [4][5]，ハードディスクなどの二次記憶装置に繰り返しアクセスが生じる Apache Hadoop[6] と比較して 15 倍の性能向上を実現している例もある．しかしビッグデータ処理で見られるような大量のデータを処理する場合は，一時保存するデータがメモリ上の一時保存領域に収まらないため，性能が低下する [7]．

Spark では，中間データの一時保存先として，メモリまたはディスクのみを使用する構成と，両者を併用する構成がある．メモリのみを使用する場合，一時保存するデータは常にメモリ上の一時保存領域に書き込むが，領域に空きがない場合は既に保存されているデータを削除して領域を確保する．そのため削除されたデータが再度必要な場合は再生成する必要がある．そこで一時保存先にディスクを用いることで，データの再生成による性能低下を抑えることができる．メモリとディスクを併用すると，LRU 規則に基づいてデータの保存先が決まるため，効率的なデータアクセスを行うことが出来る．一時保存データの有効な置き換えアルゴリズムについて様々な研究 [8][9] が行われているが，メモリとディスクを併用する場合は，メモリからディスクへのデータのドロップが頻発し，一時保存データがメモリとディスク間を頻繁に移動するスラッシング状態になり性能が低下するという問題がある．

既存の Spark では，単一のデータセットを一時保存する場合はスラッシングを回避する手法が実装されているが，それを複数データセットからなるワークロードに対して単純に適用することはできない．そこで本研究では，複数のデータセットを対象として，一時保存処理の実装をワークロードの特性に応じて動的に変更し，スラッシングを回避

[1] 名古屋工業大学大学院工学研究科
Graduate School of Engineering, Nagoya Institute of Technology

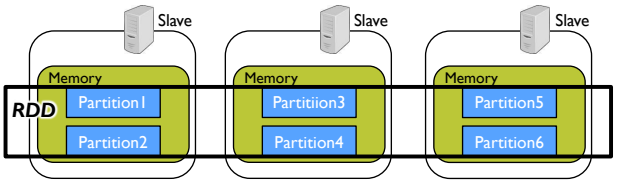

図 1 RDD とパーティション

する手法を提案する．具体的には，一時保存データの書き込み時にドロップが頻発するワークロードでは，書き込み時の動作をメモリではなくディスクに直接書き込む動作に変更し，読み出し時のドロップが頻発するワークロードではディスクにある一時保存データの読み出し時の動作を，メモリ上の一時保存領域ではなく直接実行領域に読み出す動作に変更する．メモリとディスク両者のドロップが多数発生するワークロードでは両者の動作をともに変更する．

本稿の構成は以下の通りである．まず，第 2 章では Spark のデータ一時保存の詳細と問題点を述べ，第 3 章で本研究での提案とその実装について述べる．また第 4 章で性能評価，考察を行い，第 5 章で関連研究を述べる．最後に第 6 章で今後の課題を述べ本稿をまとめる．

2. Apache Spark の一時保存機能

本章では，Spark のデータ形式と一時保存機能を説明し，一時保存における既存実装の問題点を述べる．

2.1 RDD

Spark では RDD（Resilient Distributed Dataset）と呼ばれるデータ形式を用いることで，データに対して繰り返し処理を行う場合でもメモリ上で処理が完結する．RDD は複数のノードに配置された，並列処理可能なデータのコレクションであり，Spark では，ファイルやデータソースから入力データを取得する際に RDD 形式に変換する．図 1 に示すように，内部ではパーティションという単位で分割されている．ユーザは RDD に対する操作を記述することで，内部で自動的に各パーティションに対する処理に変換されるため，容易に分散処理を実現できる．

2.2 一時保存機能

Spark では，入力時に生成した RDD や処理途中の中間データである RDD に属するパーティションを一時保存する機能が実装されている．この仕組みにより，繰り返し処理などの際に同一内容の RDD の再生成にかかる時間を短縮できる．中間データの一時保存先として，メモリまたはディスクのみを使用する構成と，両者を併用する構成がある．メモリのみを使用する場合，一時保存する RDD は常にメモリに書き込むが，領域に空きがない場合は既に保存されている RDD が LRU 方式によって選択され，選択された RDD に属するパーティションのうち新しいものが

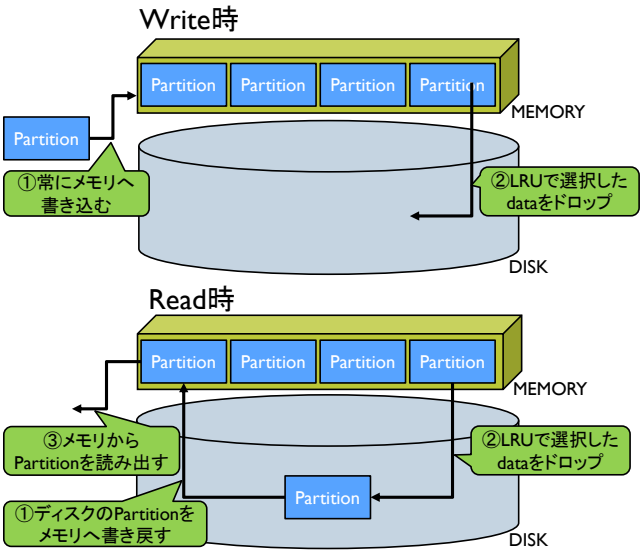

図 2 メモリとディスク併用時の動作

メモリから削除される．そのため削除されたパーティションが再度必要になった場合に再生成する必要がある．

メモリとディスクを併用した場合，図 2 に示すように RDD を一時保存する際（Write 時）は常にメモリ上の一時保存領域に書き込む．領域に空きがない場合は，LRU 規則に基づき別の RDD に属するパーティションを選択するため，ディスクへ移動するドロップが発生する．一時保存データを利用する際（Read 時）は，対象パーティションがディスクにある場合はパーティションをメモリの一時保存領域に格納してから実行領域へ読み出す．この時も，領域に空きがない場合はパーティションのドロップが発生する．

2.3 一時保存実装に関する予備評価

現在の Spark の実装では，Write 時には RDD をメモリ上の一時保存領域に書き込み，ディスクからの Read 時にはメモリ上の一時保存領域と実行領域の両方にデータを移動することで，使用頻度の高いデータをメモリ上に保持できるという利点がある．また，単一の RDD を一時保存する場合，各データは Write 時や Read 時にドロップが発生しないよう Write 時にはディスクに直接書き込み，Read 時にはディスクから直接実行領域に読み出す実装となっている．しかし，複数個の RDD を一時保存する場合は処理に必要な複数個の RDD の依存関係の検出を行うことが難しいため，Write 時/Read 時両方のケースでドロップが発生する．そのため，一時保存するデータを繰り返し利用するようなワークロードでは，メモリとディスクの間でスラッシングが発生するという問題がある．

上記で説明したスラッシングの影響を測定するため，一時保存先を変更した時の実行時間を評価した．評価環境を表 1 に，ベンチマークの設定を表 2 にそれぞれ示す．一時保存されるデータのサイズを 5.0 GB，Spark の一時保存領域を 2.4 GB に設定することで，一時保存されるデータの

表 1　評価環境

評価環境	
OS	Ubuntu 14.04
CPU	Core i5- 3476K (3.20 GHz), 4 cores
Spark が使用するメモリ	4.2 GB（一時保存領域：2.4 GB）
HDD	1 TB
Spark のバージョン	2.0.1
クラスタ数	2 台 (Master 1 台，Slave 1 台)
データソース	HDFS 2.7.3

表 2　ベンチマークの設定

プログラム	設定
k-means	サンプル数 2000 万 サンプルの次元数 20 k=10 一時保存データ 5.0GB （頂点座標の集合：4.0GB, クラスタ中心との距離：1.0GB）

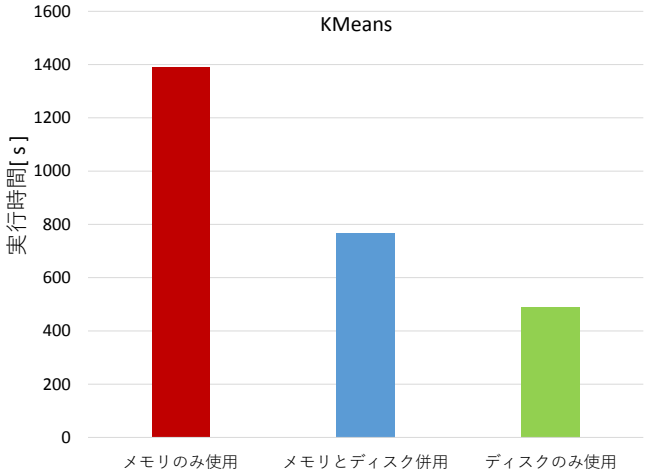

図 3　一時保存先の変更による実行時間への影響

半数程度がドロップするように設定．

評価結果を図 3 に示す．図中のグラフは左から一時保存先としてメモリのみ使用した場合，メモリとディスクを併用した場合，ディスクのみ使用した場合の実行時間をそれぞれ表している．メモリのみを使用した場合の性能が最も低く，これはメモリからドロップしたデータの再生成のコストが大きいためである．一方で，ディスクのみを使用した場合は性能が最も高く，これは大量のデータを一時保存する場合は，RDD を再生成するコストが大きいため，ディスクを用いることでデータの再生成による性能低下を抑えることができるためである．また，メモリとディスクを併用した場合は，ディスクのみを使用する場合よりも悪い結果となった．そこで，メモリとディスクを併用した場合のドロップ数を計測したところ，Write 時のドロップ数は 36 回，Read 時のドロップ数は 3659 回であり，スラッシングが発生していることが速度低下の原因であることを確認した．

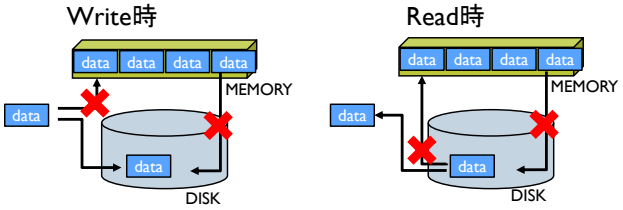

図 4　提案手法の一時保存動作

3. 提案手法

本章では，一時保存先にメモリとディスクを併用した構成において，ワークロードに適した一時保存手法を動的に選択する手法を提案する．これにより，複数のデータセットからなるワークロードにおいても，前章で述べた一時保存機能のスラッシングを回避することが可能になる．

3.1 複数 RDD の一時保存手法の検討

複数個の RDD の一時保存先としてメモリとディスクを併用する場合，現在の実装では Write 時/Read 時の両方においてデータをメモリ上の一時保存領域に保存することで，使用頻度の高いデータがメモリ上に保持される仕組みになっている．提案手法では動作を変更し，データのドロップが発生しないようにすることでスラッシングの回避を図る．

一時保存動作の変更手法について図 4 に示す．まず Write 時には，メモリに空きがある場合は従来と同様メモリに書き込むが，メモリに空き領域がない場合は，既存の Spark ではメモリに書き込んでいたためにドロップが発生していたため，ディスクに書き込むように変更する．次に Read 時には，メモリのデータを Read する場合は従来と同様メモリから読み出す．ディスクのデータを Read する場合は，既存の Spark ではメモリに書き戻してから読み出し，ドロップが発生していたため，メモリに書き戻さずに直接ディスクから読み出す．

3.2 各ワークロードにおける適切な一時保存手法の傾向

一時保存手法の変更の有効性を確認するために，既存の Spark と，一時保存手法を変更した Spark でベンチマークを実行し，実行時間を比較した．比較対象は以下の 5 種類である．

(1) 一時保存にメモリとディスクを併用した従来手法
(2) 一時保存にディスクのみを使用した従来手法
(3) Write 時の動作を変更した時の提案手法
(4) Read 時の動作を変更した時の提案手法
(5) Write 時，Read 時ともに変更した時の提案手法

ベンチマークには KMeans に加えてグラフ処理アルゴリズムの NWeight を使用し，一時保存されるデータのサイズはそれぞれ 5.0 GB，8.0 GB とした．NWeight の設定を

表 3　ベンチマークの設定

プログラム	設定
NWeight	edge: 5000000 degree: 3 max_out_edges: 30 一時保存データ 8.0 GB （頂点と辺の集合）

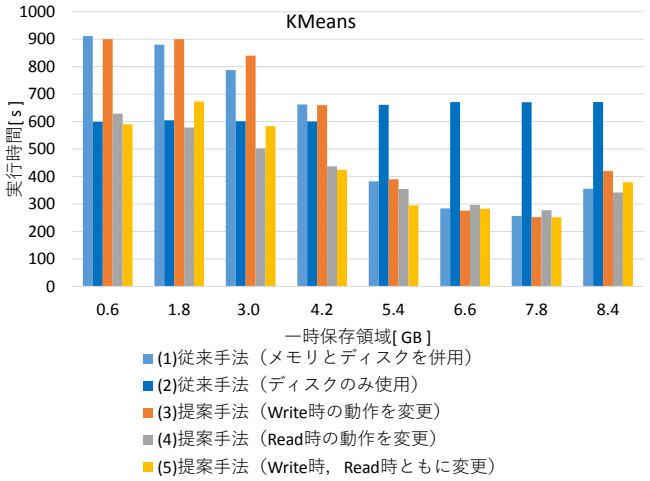

図 5　KMeans 予備評価

表 3 に示す．

KMeans の評価を図 5 に示す．グラフは，メモリの一時保存領域のサイズを 0.6 GB から 8.4 GB まで変更した時の各手法での実行時間を表す．5 本のバーは左から，(1) 一時保存先にメモリとディスクを併用したときの従来手法の実行時間，(2) 一時保存先にディスクのみを使用した従来手法での実行時間，(3) Write 時にデータをディスクに直接書き込むように変更したときの実行時間，(4) Read 時にデータを実行領域に直接読み出すように変更したときの実行時間，(5) Write 時，Read 時ともに変更したときの実行時間を表している．まず (2) の手法では，メモリの一時保存領域のサイズを変化させても実行時間が一定であり，メモリを利用した高速化を行うことができないことが分かる．次に一時保存にメモリとディスクを併用している (1)，(3)〜(5) の手法では，一時保存データが 5.0 GB であるのに対し，一時保存領域を 0.6 GB から 7.8 GB まで増加させると，それに伴い実行時間が削減された．これは，一時保存データがメモリに格納される量が増えたためドロップ数が減少し，一時保存領域が 5.4 GB 以上の時にはメモリに全て格納されドロップが無くなったからである．一時保存領域が 8.4 GB まで増加すると実行時間が増加するが，これは一時保存領域と Spark の実行領域を合わせると 14.0 GB となり，計算機のメモリ 16 GB の大部分を使用するため，プロセスが仮想メモリを使用したためである．次に，一時保存領域が 4.2 GB 以下のとき，一時保存データのドロップが発生しているときの実行時間をみると，(4) と (5) の手法の実行時間が (1) の手法より削減されている．特に，(4) の手法で最も性能が向上し，最大 33%実行時間を削減している．これは図 6 に示すように，KMeans の一時保存データに対する処理は，Write が最初に行われるのみで Read が大部分を占めており，Read 時のドロップがスラッシングの原因となっているため，Read 時の動作を変更することでスラッシングを回避できるからである．

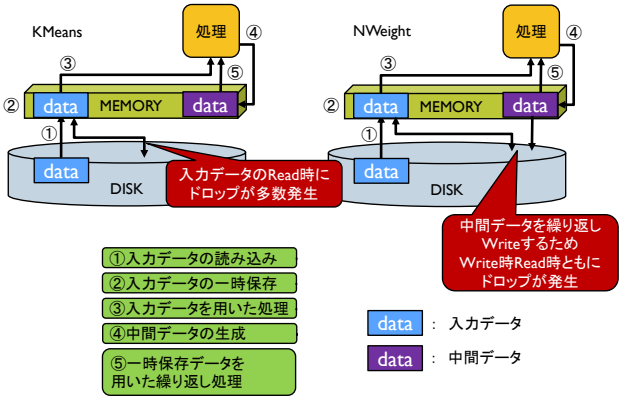

図 6　KMeans，NWeight のデータフロー

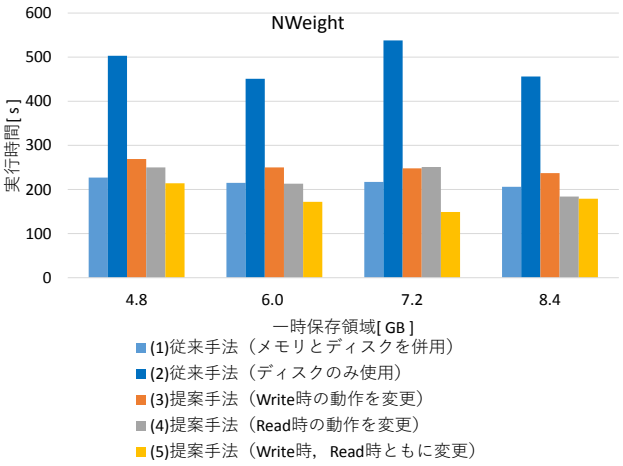

図 7　NWeight の評価結果

次に，NWeight での評価を図 7 に示す．グラフは，一時保存領域を 4.8 GB から 8.4 GB まで変更した時の各手法での実行時間を表す．一時保存領域を 4.8 GB 未満にするとメモリが少なく，ベンチマークが正常に実行できなかった．KMeans と同様に 5 つの手法で評価を行った．まず，(2) の手法ではディスクアクセスの影響により性能が大きく低下していることが分かる．提案手法での実行時間を従来手法と比較すると，Write 時，Read 時ともに変更したときの実行時間が従来手法より削減されていることがわかる．KMeans での，(4) の手法が最も速いという結果とは傾向が異なり，NWeight では Write 時，Read 時の動作ともに変更することで一時保存領域が 7.2 GB の時に最大 32%速度向上するという結果になった．これは図 6 に示すように，NWeight の一時保存データに対する処理は，Write/Read ともに同程度行われ，Write 時 Read 時両方のドロップがスラッシングの原因となっているため，両方の動作を変更

することでスラッシングを回避できるからである．

3.3 一時保存手法の動的変更

以上より，KMeans では Read 時の動作を変更したときが最も速く，NWeight では Write 時，Read 時ともに動作を変更したときが最も速いため，ワークロードによって適切な一時保存手法が異なることが分かった．このような実行前に一時保存手法を変更することは，対象とするワークロードのアクセスパターンが事前に分かっている場合は適用可能である．しかし，実際には様々なワークロードが存在し，実行前に事前にアクセスパターンを知ることは困難である．そこで本研究では，実行途中にワークロードのアクセスパターンを観測し，ワークロードに適した一時保存手法に動的に変更する手法について検討する．動的変更を実装するにあたり，スラッシングの発生を知ることのできる，ドロップの回数を変更の指標として導入する．

表 4 各ベンチマークのドロップ数

プログラム	Write ドロップ数	Read ドロップ数
k-means	40 回	3659 回
Bayes	14 回	48 回
NWeight	171 回	96 回

各ベンチマークにおいて一時保存の Write 時および Read 時のドロップ数をそれぞれ計測した．[*1] 計測結果を表 4 に示す．この結果から，Read 時を変更したときが最も速い KMeans では Read 時のドロップが多く，Write 時，Read 時ともに変更したときが最も速い NWeight では Write 時のドロップと Read 時のドロップが同程度であるというように，Write 時のドロップ数と Read 時のドロップ数の比率に，適切な一時保存手法との関係性があることが分かる．よって本研究では，一時保存手法の変更とともに，ドロップ数をもとにワークロードに適した一時保存手法を動的に選択する手法を提案する．

一時保存手法を，計測したドロップ数を指標にして動的に変更する．Write 時，Read 時のいずれかでドロップが発生した場合，多く発生した方の動作が原因でスラッシングが発生したとみなすことが出来るため，動的変更の条件は，Write 時のドロップ数と Read 時のドロップ数の比率とした．動的変更の基準を表 5 に示す．本実装では，Read

表 5 動的変更の比率

Read 時ドロップの比率	Write 時を変更	Read 時を変更
70%以上		○
20%以上 70%未満	○	○
20%未満，ドロップなし		

[*1] Write 時のドロップ数の計測ではスラッシングに関係するドロップのみを計測するため，実行開始直後はドロップが発生した時点でカウントし，Read 時のドロップの発生後はドロップしたデータが Read された時点でカウントする．

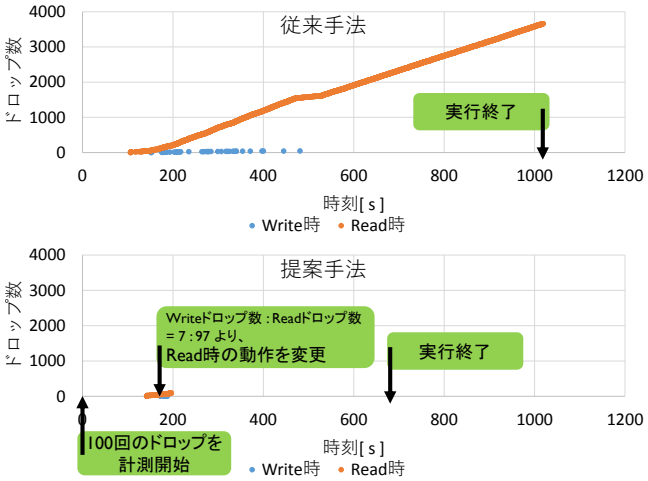

図 8 KMeans のドロップ数の推移

時のドロップが 70%以上を占める時に Read 時の動作のみを変更し，20%以上 70%未満の時には Write 時 Read 時ともに変更することに決める．また，Read 時のドロップが 20%未満の時とドロップ数が Write,Read ともに 0 回のときは，標準の手法であるメモリとディスクを併用した従来手法を用いる．

4. 評価

一時保存手法の動的変更の有効性を確認するため，既存仕様の Spark に追加実装を行い，KMeans，NWeight，Bayes，Pagerank の 4 つのベンチマークを用いて性能評価を行った．評価環境を表 1，KMeans の設定を表 2，Bayes

表 6 ベンチマークの設定

プログラム	設定
bayes	クラス数 100 一時保存データ 6.0 GB
pagerank	ページ数 500000 一時保存データ 4.0 GB

と Pagerank の設定を表 6 に示す．実行開始直後に手法を切り替えるタイミングとして Write 時と Read 時のドロップ数の合計が仮に 100 回となるまで計測して比率を計算し，手法を切り替える．

4.1 評価結果

既存手法と提案手法で KMeans を実行した時のドロップ数の推移を図 8 に示す．上のグラフは既存手法でメモリとディスクを併用した時のドロップ数，下のグラフは提案手法でのドロップ数の推移を表す．既存手法では，Write 時のドロップは実行開始直後に発生するのみで，その後は Read 時のドロップのみが大量に発生していることが分かる．提案手法では，実行開始後の 100 回のドロップのうち 97 回 Read 時のドロップが発生したため Read 時の手法が提案手法の動作に切り替わり，その後のドロップの発生を

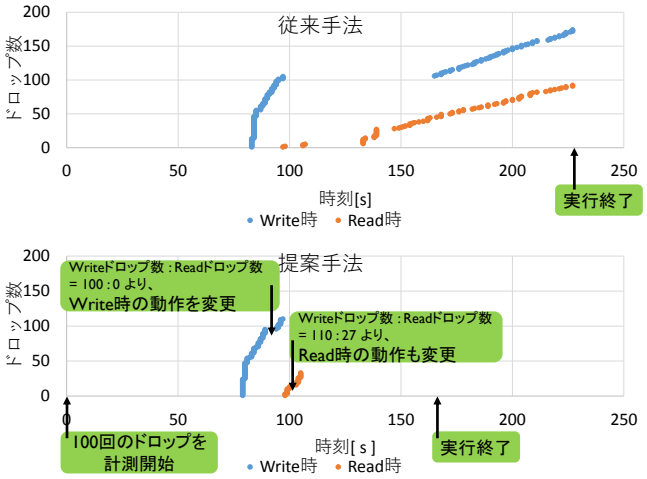

図 9 NWeight のドロップ数の推移

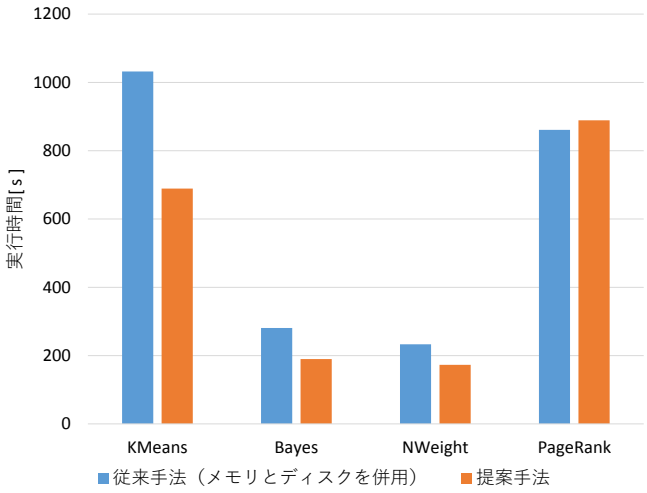

図 10 一時保存手法の動的変更

抑制した．

次に，既存手法と提案手法で NWeight を実行した時のドロップ数の推移を図 8 に示す．上のグラフは既存手法でメモリとディスクを併用した時のドロップ数，下のグラフは動的変更手法でのドロップ数の推移を表す．既存手法では，実行開始 80 秒後に Write 時のドロップが 100 回を超えて発生し，その後は Write 時，Read 時ともにドロップが多数に発生していることが分かる．提案手法では，実行開始後の 100 回のドロップが全て Write 時のドロップであったため，Write 時の手法が切り替わり，その影響でその後は Read 時のドロップが 30 回程発生して Read 時のドロップの割合が 20%を超えたため，Read 時の手法も切り替わり，その後のドロップの発生を抑制した．

各ベンチマークで提案手法を適用した評価を図 10 にまとめた．2 本のバーは左から，従来手法でメモリとディスクを併用したときの実行時間、動的変更手法を実装したときの実行時間を表している．KMeans，Bayes，NWeight，Pagerank の 4 つのベンチマークについて評価をした．KMeans と Bayes のときは，Read 時の動作を変更したときが最も速く，Read 時のドロップが多いため，Read 時の動作を変更した手法に切り替わり，速度が向上している．NWeight では，Write 時 Read 時ともに変更したときが最も速く，両者のドロップが多数発生するため，両方の動作を変更した手法に切り替わり，速度が向上している．Pagerank では単一の RDD のみを一時保存しているため，従来手法でもスラッシングへの対策が行われ，ドロップが発生しなかった．そのため提案手法では手法が切り替わらなかったが，調査では従来手法が最も速かったため，切り替わらないという動作が有効である．以上より，各ワークロードに応じて適切な一時保存手法に動的に変更し，速度を向上していることが分かり，提案手法の有効性を確認した．

5. 関連研究

張ら [10] は RDD の一時保存にメモリとディスクを併用した場合の，データ量に対してメモリサイズが小さいときの性能を評価し，スラッシングと GC が原因で性能が低下することを明らかにした．そこで，一度メモリからディスクにドロップしたブロックについては再度使用する場合でもメモリに書き戻さないという手法を実装することで，40%程度の高速化を実現した．我々の提案手法は，上記の手法に加えて一時保存データの Write 時の動作を変更した手法を実装し，さらにワークロードに適した手法に動的に変更した機構を実装している点に優位性がある．

Jiang ら [11] は一時保存する RDD に対するデータ圧縮によりメモリ領域を節約し，ディスク I/O を削減している．さらに，データ圧縮にかかる時間，ディスク I/O 時間，RDD の再生成にかかる時間を数学的に導出し，一時保存を行うかどうかの選択と，行う場合は 3 つの圧縮アルゴリズムのうち適切なものの選択を実行時に動的に行う手法を提案し，最大 6 倍の高速化を実現している．

6. まとめ

本稿では，複数のデータセットを一時保存する際に生じるスラッシングを回避するために既存の Spark に一時保存方式を変更する手法を追加し，ワークロードに適した一時保存方式を動的に適用する手法を提案した．提案手法の有効性を確認するために，ドロップ数を指標にしてワークロードに適した手法に動的に変更する機構を実装し，ベンチマークを実行した．評価結果から，KMeans と Bayes では Read 時の動作を変更した手法に切り替わり，実行時間がそれぞれ 33%，28%削減された．NWeight では Write 時 Read 時両方の動作を変更した手法に切り替わり，実行時間が 26%削減された．しかし一時保存する RDD が 1 つであるためにドロップが発生しない Pagerank では手法が切り替わらなかった．このような結果から，複数の RDD を一時保存する場合に提案手法が有効であると言える．

今後の課題として，手法の動的変更のしきい値を経験的

に決定しているため，比率だけでなくドロップ数や一時保存データに対するアクセス回数などを考慮することで多くのベンチマークに手法を適用可能にすることが挙げられる．

謝辞 本研究の一部は，科研費基盤研究（C)15K00168による．

参考文献

[1] J.F Gantz. The diverse and exploding digital universe. *An IDC white paper*, 2008.

[2] M. Zaharia. Spark: In-Memory Cluster Computing for Iterative and Interactive Applications. In *Invited Talk. NIPS Big Learning Workshop: Algorithms, Systems, and Tools for Learning at Scale (Granada, Spain.)*, Dec. 2011.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX conference on Networked Systems Design and Implementation (California, USA.)*, p. 2, April 2012.

[4] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, DB. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, Vol. 17, No. 34, pp. 1–7, 2016.

[5] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, Vol. 12, pp. 10–10, 2012.

[6] Apache Hadoop. `http://hadoop.apache.org/`（参照日: 2018-01-26）.

[7] L. Gu and H. Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *Proc. 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, pp. 721–727, 2013.

[8] Y. Yu, W. Wang, J. Zhang, and K.B. Letaief. Lerc: Coordinated cache management for data-parallel systems. *arXiv preprint arXiv:1708.07941*, 2017.

[9] 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, 2016.

[10] 張凱輝, 谷村勇輔, 中田秀基, 小川宏高. Spark RDD の入出力性能の高速化に関する検討. 信学技報（CPSY2016-16）, Vol. 116, No. 177, pp. 77–82, 2016.

[11] Z. Jiang, H. Chen, H. Zhou, and J. Wu. An elastic data persisting solution with high performance for spark. In *Proc. 2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity),*, pp. 656–661, 2015.