Vol.2019-OS-147 No.6
2019/7/25

Apache Sparkにおける ブルームフィルタを用いたSQL処理の高速化

早瀬　大智[1]　稲垣 英夫[1,2]　川島 龍太[1]　松尾 啓志[1]

概要：機械学習やビッグデータの解析を複数の計算機で分散させる手法が一般的となり，そのフレームワークとして Apache Spark が広く利用されている．Spark は頻繁に利用するデータをメモリにキャッシュすることで高速な処理を実現するインメモリ処理を採用しているが，データサイズがクラスタのメモリ総量を超える場合，スラッシングが発生してスループットが低下する．Spark はキャッシュ管理手法に LRU(Least Recently Used) を採用しているが，LRU は短期的なアクセスパターンからキャッシュするデータを決めるため，アクセスパターンが変化した時にメモリ-ディスク間でデータの移動が多発する．より長期的なアクセスパターンを考慮したキャッシュ管理手法として，参照カウントによって頻繁に利用される RDD(Resilient Distributed Dataset) を検出可能な LRC(Least Reference Count) が提案されているが，RDD 内部のパーティション単位でアクセスに偏りがある場合には対応できない．本研究では，パーティション単位のブルームフィルタを導入し，不要なパーティションへのアクセスを禁止することで，頻繁に利用されるパーティションの検出を可能にする．具体的には，パーティション内に含まれるレコードをブルームフィルタに登録し，Master に送信する．Master はそのブルームフィルタを基に，処理対象のデータが含まれている可能性があるパーティションの処理のみを Slave に依頼する．提案手法を性能テストベンチマークである TPC-H で評価した結果，9.8%の性能向上を確認した．

キーワード：並列分散処理, Apache Spark, SQL

1. はじめに

SNS や IoT の普及に伴い，世界中で生み出されるデジタルデータの量は爆発的に増大しており [1]，生み出されたビッグデータを事業の拡大に利活用する取り組みが活発に行われている．ビッグデータの処理には複数の計算機で処理を分散させる手法が広く利用されているが，全ての計算機のリソースを有効活用するプログラムを記述するには障害時の対応や通信処理の知識が必要になるため，並列分散処理を効率的に記述するためのフレームワークとして Apache Spark[2] が広く利用されている．

Spark はデータセットや処理の中間結果などをメモリに保持することで高速な処理を実現しているため，全てのデータをメモリにキャッシュできる場合，高速な処理が可能である．しかし，データサイズがクラスタのメモリ総量を超える場合，メモリ-ディスク間でデータの移動が多発し，スラッシングを起こす可能性があるため，キャッシュ管理手法は Spark の性能に大きく影響を与える [3]．

[1] 名古屋工業大学 Nagoya Institute of Technology
[2] 株式会社小松製作所 Komatsu Ltd.

Spark はキャッシュ管理手法に LRU(Least Recently Used) を用いて，直近のアクセス履歴から将来的に利用される RDD(Resilient Distributed Dataset) を予測する．しかし，短期的なアクセスパターンのみを利用するため，アクセスパターンが変化するとヒット率が低下し，スラッシングが発生する．

そこで，より長期的なアクセスパターンをキャッシュ管理に利用するため，新たなキャッシュ管理手法として LRC(Least Referrence Count) が提案されている [4]．LRC は参照カウントによって頻繁に利用される RDD をより正確に把握するキャッシュ管理手法である．これは頻繁に利用される RDD が 1 個の時は効果的であるが，頻繁に利用される RDD が複数存在し，全てをメモリにキャッシュできない場合，頻繁に使用されるパーティションのみをメモリにキャッシュする必要がある．しかし，RDD 内部のパーティション単位でアクセスに偏りがあると，アクセス頻度の低いパーティションもメモリにキャッシュしてしまう問題がある．そのため，より粒度の細かいパーティション単位で参照カウントを記録する必要があるが，filter 関数実行時は全てのパーティションにアクセスが発生するため，頻

© 2019 Information Processing Society of Japan

繁に利用されるパーティションを正確に特定できない．

本研究では LRC にブルームフィルタを導入することで，不要なパーティションへのアクセスを制限し，高頻度にアクセスされるパーティションのみをメモリにキャッシュする手法を提案する．提案手法ではパーティション単位でブルームフィルタを生成し，生成したブルームフィルタをもとにアクセスするパーティションを決定する．これにより従来より精度の高いパーティションのアクセス履歴を取得することができ，頻繁に利用されるパーティションのみをメモリ上に配置することが可能になる．

本稿の構成は以下のとおりである．2 章で Spark のキャッシュ機構について述べ，3 章では LRU に変わるキャッシュ管理手法として提案されている関連研究について言及する．4 章でブルームフィルタを用いた新たなキャッシュ管理手法を提案し，5 章では提案手法の評価と考察を行う．最後に 6 章で今後の課題を述べ，まとめとする．

2. Spark のキャッシュ機構

Spark では，RDD に対してアクション処理を実行するたびに RDD の再計算が発生する．そのため，複数回利用する RDD が存在する場合，同じ計算を繰り返し行う必要があり，非効率である．そこで，頻繁に利用する RDD を明示的にメモリやディスクにキャッシュすることで，RDD の再計算を抑制し，冗長な処理を削減している．RDD をキャッシュする時の方法として，Memory-Only 方式，Disk-Only 方式，Memory-and-Disk 方式の 3 種類が存在するが，実際には不要になった RDD をディスクに退避できる Memory-and-Disk 方式が広く利用されている [5]．

Spark はキャッシュ管理手法に LRU を採用しているが，複数 RDD をキャッシュする場合，メモリ-ディスク間でパーティションの移動が頻繁に発生するため，スラッシングの発生によってスループットが低下する．そのため，LRU に変わる様々なキャッシュ管理手法が提案されている．

3. 関連研究

本章では LRU に変わる新しいキャッシュ管理手法について述べる．

3.1 参照カウントを用いたキャッシュ管理

LRC は RDD の参照カウントを記録し，アクセス回数が多い RDD をメモリにキャッシュするキャッシュ管理手法である．メモリ上にある参照カウントが最小の RDD とキャッシュしたい RDD の参照カウントを比較して，大きい方をメモリにキャッシュする．ジョブ全体を通して参照カウントを記録するため，一時的なアクセスパターンの変化に影響を受けず，頻繁に利用されるデータを常にメモリにキャッシュすることができる．

LERC(Least Effective Reference Count)[6] は LRC を基に提案されたキャッシュ管理手法で，処理対象となる RDD 全体をひとつのブロックとして参照カウントを記録する．これにより，複数 RDD を入力とする関数実行時に，入力となる一連の RDD を優先的にメモリにキャッシュすることで，スループットの向上を見込む．

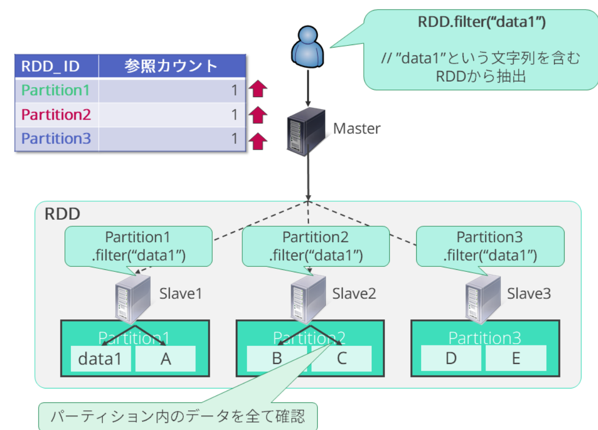


図 1 filter 関数の動作例

しかし，RDD のパーティションにアクセスの偏りがある場合，不要なパーティションまでメモリにキャッシュするため，パーティション単位で参照カウントを記録する必要がある．ここで，filter 関数を用いて RDD 内部から"data1"という文字列を含むデータを抽出する処理を想定する．図 1 に示すように，対象となる文字列の存在を確認するため，Slave は全てのパーティションの中身を確認する必要がある．そのため，処理対象のデータが含まれているパーティションだけでなく，処理対象のデータが含まれていないパーティションへのアクセスも発生し，全てのパーティションの参照カウントが増加してしまう．

このように，filter 関数を実行した場合，パーティション単位の参照カウントを正確に記録することは困難であり，頻繁に使用されるパーティションのみをメモリにキャッシュすることは不可能である．より粒度の細かいパーティション単位で参照カウントを取ることができれば，アクセスの偏りの影響を回避できるが，正確な参照カウントを記録するためには，必要のないパーティションへのアクセスを削減する機構が必要である．本研究では，より正確な参照カウントを記録する手法について検討する．

3.2 データ移動を抑制するキャッシュ管理

Modified I/O behaviors[7] はワークロードに応じてスラッシングの原因となるメモリディスク間の移動を禁止するキャッシュ管理手法である．Modified I/O behaviors は Read/Write 処理時にディスクへのデータドロップが頻発した場合，ディスクへの読み出し/書き込みを許可するように変更することで，スラッシングを抑制する．しかし，Modified I/O behaviors は Read/Write 処理を変更した場

© 2019 Information Processing Society of Japan

合，一度ディスクに格納されたパーティションはメモリに移動しないため，頻繁に利用されるパーティションがディスクに移動した際に性能低下を起こす．また，アクセス履歴を考慮していないため，キャッシュのヒット率が低いという課題もある．本研究では，長期的なアクセス履歴を基にキャッシュするデータを選択することでヒット率の向上を実現する．

4. 提案手法

本章では LRC にブルームフィルタを実装することでより正確な参照カウントを記録する手法について提案する．

4.1 概要

LRC の参照カウントによって頻繁に利用される RDD をメモリにキャッシュできるようになったが，頻繁に利用される RDD が複数存在し，全てのを RDD をキャッシュできない場合，頻繁に利用されるパーティションをメモリにキャッシュする必要がある．しかし，3.1 節で示したように，filter 関数実行時には LRC を用いてパーティション単位の参照カウントを正確に記録することはできない．そこで，filter 関数実行時に対象のデータが含まれるパーティションの参照カウントのみを増加させ，メモリにキャッシュするために，LRC とブルームフィルタを組み合わせたキャッシュ管理手法を提案する．

提案手法の概要図を図 2 に示す．Slave はパーティションごとのブルームフィルタを用意し，パーティション内部のデータを文字列としてブルームフィルタに登録する．その後，Master にブルームフィルタを送信する (図 2a)．Master は Slave から受け取ったブルームフィルタを確認することで filter 関数の対象データが含まれているかを確認できるため，対象データがブルームフィルタに登録されていないパーティションは処理を省略できる．

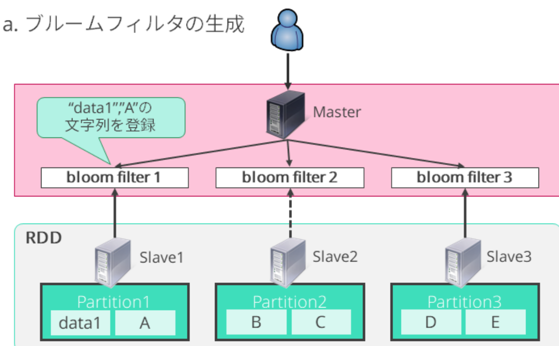
対象データが含まれていないパーティションへのアクセスを禁止することで，処理するパーティション数の削減が行えるだけでなく，対象データが含まれているパーティションの参照カウントのみを増加させることができるため (2b)，より正確な参照カウントが記録できる．

4.2 実装

上記の提案手法を Spark 2.4.0 上に実装した．

4.2.1 LRC

パーティションの参照カウントを記録するためのテーブルとして，パーティションの識別子と参照カウントの 2 つを管理する HashMap, ”LRCMap ” を用意する．パーティションの識別子は全てのパーティションに付与されたユニークな値である BlockId を用いた．LRCMap はデータの読み出しが初めて行われた時に BlockId が登録され，Slave からアクセスされるたびに参照カウントが増加する．



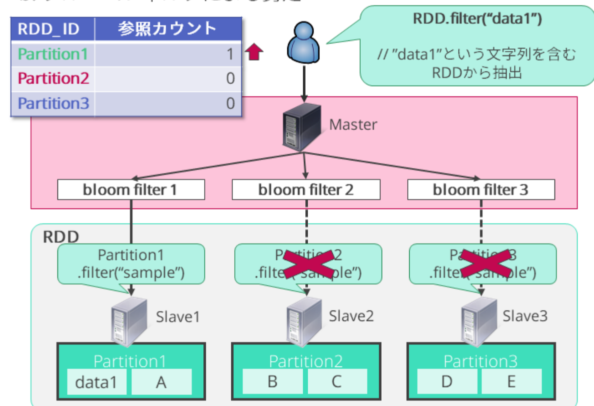


図 2 提案手法の概要図

従来の LRC では，メモリ上にある参照カウントが最小のパーティションとキャッシュしたいパーティションの参照カウントを比較して，大きい方をメモリにキャッシュする仕組みになっている．しかし，データの読み出しが発生するたびにメモリ上に存在する全てのパーティションの参照カウントを確認するのは時間がかかる．そこで，キャッシュしたいパーティションの参照カウントより参照カウントが小さいパーティションすべてを追い出し対象とすることで，実行時間の削減を行う．

4.2.2 ブルームフィルタ

提案手法に用いたブルームフィルタは Spark から提供されているものを利用した．このブルームフィルタはオープンソースである Google Guava[8] を基に設計されたもので，RDD 内部に対象のデータが含まれているかを確認する機能としてユーザに提供している．本研究ではより粒度の細かいパーティション単位でブルームフィルタを用意する．ブルームフィルタのサイズは Spark によって自動設定され，偽陽性はデフォルト値である 3%に設定した．

次に，提案手法におけるブルームフィルタの管理手法について説明する．提案手法では多段の HashMap を利用し，RDD の ID, パーティションの ID，パーティションのブルームフィルタという 3 つの情報を管理する．RDD をキャッシュする関数である persist 関数が実行されたタイミングで空のブルームフィルタを生成し，パーティションを指定したストレージへ格納する doPutIterator 関数が実行

© 2019 Information Processing Society of Japan

されたタイミングで，パーティションの中身のデータをブルームフィルタに登録する．この時，パーティションに含まれるデータは全て文字列に変換され，ブルームフィルタに登録される．キャッシュしたデータを廃棄する unpersist 関数が呼ばれたタイミングでブルームフィルタは破棄される．また，パーティションごとに生成されたブルームフィルタを用いてデータのフィルタリングを行うため，新たに Bloomfilter 関数を作成する．Bloomfilter 関数は検索対象の文字列を引数として取ることで，対象データのみが含まれる RDD を生成する関数とした．

4.3　SparkSQL への応用

SQL では WHERE 句のような特定のデータを抽出するクエリが存在するが，これは提案手法で想定している処理と近い処理である．Spark には SQL 処理用のライブラリとして，SparkSQL[9] が存在するため，SparkSQL を用いた評価も行う．SparkSQL には PartitionPruning 機能という，処理対象となるパーティションを剪定する機能が提供されている．これは上界と下界とを保持し，対象データが当該パーティションが含まれるかどうかを判定し，対象データが含まれていない場合，そのパーティションを処理対象から外すという仕組みになっているが，剪定条件が厳しいためほとんど剪定されないという問題を抱えている．そこで，PartitionPruning 機能に提案手法を実装することで処理対象となるパーティション数の削減を試みる．

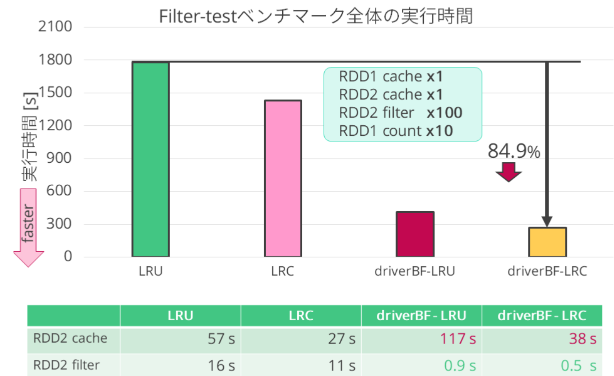
ただし，SQL におけるレコードは一般的に複数の属性を持つため，提案手法では各属性の値をカンマで結合し，一つの文字列として登録するものとする．これにより，属性ごとのブルームフィルタを保持する必要がなくなり，メモリ使用量とブルームフィルタのチェック回数を削減する．

5.　評価

本章では RDD 内の一部のデータに繰り返し利用する filter-test ベンチマークと，性能テストベンチマークである TPC-H[10] の 2 つを用いて評価を行った．評価環境を表 1 に示す．

表 1　クラスタと Spark の実行環境の詳細

評価環境	
OS	Ubuntu 16.04
CPU	Core i5 4460 (3.20 GHz), 4cores
Memory	1.0GB(Storage-memory サイズ 0.4GB)
HDD	1TB
Spark	Version2.4.0
クラスタ構成	Master x1 Slave x1
ファイルシステム	HDFS[11] (ver. 2.7.4)



	LRU	LRC	driverBF - LRU	driverBF - LRC
RDD2 cache	57 s	27 s	117 s	38 s
RDD2 filter	16 s	11 s	0.9 s	0.5 s

図 3　filter-test ベンチマークの実行結果

5.1　Filter-test ベンチマーク

提案手法におけるブルームフィルタの有効性を確認するため，アクセスするデータに偏りがある Filter-test ベンチマークを独自に作成し，性能評価を行った．SQL における Where 句は検索対象のデータを抽出するコマンドであるという性質上，アクセスするデータに偏りが発生するため，現実的なワークロードに近いアクセスパターンで評価を行う．

Filter-test ベンチマークの処理の流れを以下に示す．

1. Hadoop の RandomTextWriter を用いて生成したデータを RDD1 としてキャッシュする．
2. RDD1 で生成した文章を単語単位に分解し，RDD2 としてキャッシュする．
3. 特定の単語を抽出する Filter 関数を RDD2 に対して 10 回実行する．
4. RDD1 内の単語数をカウントする．

上記の 3.4 の処理を 10 回ずつ繰り返し実行し，計 100 回の filter 関数の実行を行う．また，RDD1,RDD2 はそれぞれ 1GB の RDD で，パーティション数は 1000 である．キャッシュ管理手法には LRU,LRC を用いたものと，ブルームフィルタを導入した LRU，LRC(以下，driverBF-LRU,driverBF-LRC) の 4 種類を用いて性能調査を行う．

4 種類のキャッシュ管理手法で filter-test ベンチマークを評価した結果を図 3 に示す．横軸は計測を行ったキャッシュ管理手法を，縦軸は実行時間を表している．

まず，cache 関数について考察する．LRU は RDD のキャッシュ時にメモリとディスク間でデータの移動が発生するため，LRC と比べてキャッシュに時間がかかることが確認できる．また，ブルームフィルタを導入したキャッシュ管理手法はブルームフィルタの生成するコストがかかるため，生成しないものと比べると 50%〜100%ほど追加で時間がかかった．

次に，filter 関数について考察する．ブルームフィルタを導入することで filter 関数一回あたりの実行時間を 1 秒

© 2019 Information Processing Society of Japan

以下に抑えられていることが確認できる．これはブルームフィルタを使用しない filter 関数と比べて，およそ 90%の実行時間削減である．また，driverBF-LRC では頻繁に使用されるパーティションのみにアクセスが発生し，メモリ上に必要なパーティションをキャッシュすることができるため，filter 関数の実行時間が削減された．

LRC とブルームフィルタを用いたキャッシュ管理手法は，LRU と比較して実行時間を最大 84.9%削減した．これはブルームフィルタの生成がオーバーヘッドとなるものの，filter 関数一回あたりの実行時間を大幅に削減できたことが原因であり，ブルームフィルタの生成コストよりも filter 関数実行時に削減できる実行時間が大きいため，この手法は有効であると考えられる．

5.2 TPC-H ベンチマーク

filter-test ベンチマークで LRC とブルームフィルタを組み合わせたキャッシュ管理手法の有用性が確認されたため，次に，より現実的なワークロードである TPC-H で評価を行う．SparkSQL 上でベンチマークを動作させるために RDD に対する処理を一部変更した TPC-H を使用し，処理中に等号条件文の含まれる Query02,Query03,Query08,Query17 の 4 種類で実行時間の計測を行った．計測は従来のキャッシュ管理手法である LRU と，filter-test ベンチマークで最も良い結果を示した driverBF-LRC の 2 種類を用いた．キャシュするレコード数は 100,000，パーティション数は 1,000 とした．

TPC-H を用いて LRU と driverBF-LRC を評価した結果を図 4 に示す．横軸は計測したクエリを，縦軸は実行時間を示している．また，グラフの緑色の部分はキャッシュ関数の実行時間，黄色の部分は実際のクエリを処理する実行時間を表している．

評価結果より，driverBF-LRC を用いることで，Query03 では LRU と比較してクエリ全体の実行時間が 3.6%増加した．Query03 における実行時間の増加は，ブルームフィルタの生成コストによって cache 関数の実行時間が 3.6%程増加したにも関わらず，不要なパーティションを剪定することができなかったためである．これは処理するベンチマーク内に等号条件文が含まれなかった場合も同じ傾向になると考えられる．

一方で，Query02 は 5.1%，Query17 では 6.2%の実行時間の削減を確認した．これは PartitionPruning 機能によって処理対象となるパーティションの数を削減されたためである．特に，Query17 では 1000 個のパーティションのうち 89.8%のパーティションを剪定することに成功した．

上記の評価は cache 関数 1 回に対して filter 関数を 1 回実行した結果である．しかし，実際のワークロードでは一度キャッシュしたデータを繰り返し利用して，クエリ複数回を実行することが多いため，本処理の実行時間を削減する

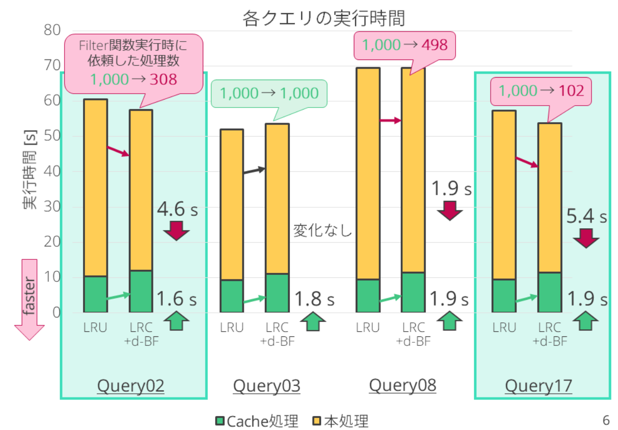


図 4 TPC-H ベンチマークの評価結果

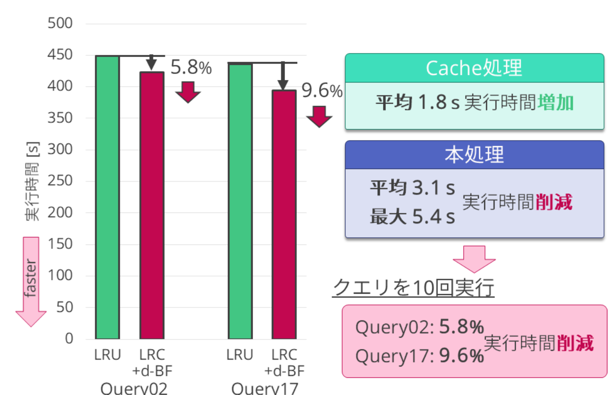


図 5 本処理を複数回実行した場合の評価結果

提案手法はより優位に働く．そこで，パーティションの剪定数が多かった Query02 と Query17 を 10 回を実行する場合の実行時間を計測する．評価結果を図 5 に示す．グラフより，Query02 の全体の実行時間を 5.8%削減し，Query17 で 9.6%削減されたことがわかる．以上から，等号条件分を含むワークロードでは提案手法は有効であることが確認できた．

6. まとめと今後の課題

本論文では，LRC とブルームフィルタを組み合わせるキャッシュ管理手法を導入することで，頻繁に利用されるパーティションの特定を可能にした．提案手法では，不要なパーティションを剪定することでアクセスするパーティション数を削減できるようになっただけでなく，より細かい粒度であるパーティション単位で頻繁に利用されているものが検出が可能になり，メモリの有効活用ができるようになった．RDD 内の一部のデータに繰り返し利用する filter-test ベンチマークでは，ブルームフィルタの生成コストよりも Filter 関数で削減できる実行時間が大きいことが確認され，従来の Spark と比較して 84.9%実行時間を削減することができた．また，提案手法を SparkSQL に応用

© 2019 Information Processing Society of Japan

し，データベースの性能テストベンチマークである TPC-H を用いて性能を評価した結果，最大 9.6%の実行時間の削減が確認された．

提案手法はアクセスするパーティションの剪定によって実行時間の削減を行っているため，ユーザの設定したパーティション数の値によっては十分に性能が出せない可能性がある．今後の課題として，最適なパーティション数を調査が挙げられる．

謝辞 本研究の一部は, 科研費基盤研究（C）18K11324 による．

参考文献

[1] Inc., C. S.: Cisco Visual Networking Index: Forecast and Trends, 2017-2022, White paper of enterprise (2019).
[2] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster computing with working sets., *HotCloud*, Vol. 10, No. 10-10, p. 95 (2010).
[3] Kaihui, Z., Yusuke, T., Hidemoto, N. and Hirotaka, O.: Toward Improving I/O Performance of Spark RDD, *CPSY2016-16*, Vol. 116, No. 177, pp. 77–82 (2016).
[4] Yu, Y.，Wang, W.，Zhang, J.，Letaief, K. B.：LRC: Dependency-Aware Cache Management for Data Analytics Clusters (2017).
[5] Karau, H., Konwinski, A., Wendell, P. and Zaharia, M.: *Learning spark: lightning-fast big data analysis*, " O'Reilly Media, Inc." (2015).
[6] Yu, Y., Wang, W., Zhang, J. and Letaief, K. B.: LERC: coordinated cache management for data-parallel systems, *GLOBECOM 2017-2017 IEEE Global Communications Conference*, IEEE, pp. 1–6 (2017).
[7] Inagaki, H., Fujii, T., Kawashima, R. and Matsuo, H.: Adaptive Control of Apache Spark's Data Caching Mechanism Based on Workload Characteristics, *Proc. 5th International Symposium on Big Data Research and Innovation (BigR&I-2018)*, pp. 64–69 (Barcelona, Spain, Aug. 2018).
[8] Google: Google Guava, https://github.com/google/guava[参照日時 2019/06/19].
[9] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A. et al.: Spark sql: Relational data processing in spark, *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, ACM, pp. 1383–1394 (2015).
[10] valuko: TPC-H, https://github.com/valuko/spark-tpch[参照日時 2019/06/19].
[11] Shvachko, K., Kuang, H., Radia, S., Chansler, R. et al.: The hadoop distributed file system., *MSST*, Vol. 10, pp. 1–10 (2010).

© 2019 Information Processing Society of Japan