Vol.2016-HPC-153 No.21
2016/3/2

Spark RDDのストレージ出力に関する性能評価

谷村 勇輔[1,a)]　小川 宏高[1]

概要：Spark は反復計算や対話的な処理など，Hadoop の MapReduce ベースのアプリケーションが苦手とする処理に対して高速なデータ処理を可能にすることから新しいビッグデータ解析基盤として注目を集めている．Spark の特徴は，RDD（Resilient Distributed Dataset）の仕組みにより耐障害性を確保したインメモリ処理であるが，中間データの再処理に時間がかかる場合やメモリ容量が不足する場合，データのインポートやエクスポートを行う場合などストレージが必要となるケースは様々に存在し，ストレージの入出力性能が全体性能に与える影響も少なくないであろうと考えられる．そこで我々は，Spark アプリケーションの性能チューニングや Spark システムの構築における指針を得るため，RDD のストレージに対する入出力性能について調査を開始した．本稿では RDD の Persistence の確保やチェックポイント実行時のデータの出力性能を調査した結果を報告する．

1. はじめに

近年，ビッグデータの活用分野は大きく広がり，人工知能に相当するような，より高度な「インテリジェンス」の抽出・利用技術への期待も高まっている．そのような中で，Apache Spark（以下，Spark） [1–3] はビッグデータの解析基盤として高い注目を集めている．Spark は，これまでのビッグデータ解析基盤の主流と言える Hadoop [4] の MapReduce [5] が苦手とする反復計算や対話的な処理に向いており，機械学習やグラフ処理を高速に実行できる利点がある．それは再利用される中間データを外部ストレージに書き出さず，メモリに保持することでデータ入出力を削減したことによるものである．一方，中間データの生成手順を記録しておき，障害発生時には，失われた中間データを再処理する方針をとることで耐障害性を確保している．Spark のこの仕組みは RDD (Resilient Distributed Dataset) と呼ばれる．

RDD は有効な仕組みであるが，中間データの再処理に時間がかかる場合には，外部ストレージにデータを保存しておいた方が障害時の実行時間を短縮できる．また，シャッフル処理時の自動的なチェックポイント，処理すべきデータのインポートや結果のエクスポートなど，ストレージへのアクセスは Spark においても依然として重要である．現状の Spark では，中間データのストレージへの保存は一部を除き，ユーザ側のプログラムに委ねられているため，アプリケーションを効率的に実行するために，ユーザはストレージへのデータ入出力性能を知っておくことが望ましい．また，Spark システムを構築・提供する側には，どのようなストレージを用意すべきか，そしてその費用対効果を把握しておくことが求められる．

本研究では，Spark RDD の入出力性能について調査し，ワーカノードに搭載されるディスク，および全ワーカノードで共有される分散ファイルシステムに求められる性能要件を明らかにする．これにより，Spark アプリケーションの性能チューニングや Spark システムの構築のための指針を得ることを目指す．

本稿ではその最初の取り組みとして，persist() メソッドによる RDD の継続保持，および checkpoint() メソッドによる RDD のチェックポイントのぞれぞれにおいてデータの出力性能を調査した．そして，RDD のサイズ，バックエンドのストレージ（すなわち，ディスクや分散ファイルシステム）の性能，シリアライザによる違いを比較した．これらの実験結果により，バックエンドのストレージ性能が与える影響の度合いやシリアライザの選択に関する知見を得ることができた．

2. Spark と RDD の仕組み

2.1 Spark の概要

Spark は University of California, Berkeley の AMPLab [6] の研究プロジェクトにより開発が始まり，のちに Apache ソフトウェア財団 [7] の支援を受けて開発が続けられているオープンソースの大規模データ解析基盤である．Spark は処理エンジンである Spark Core をベースとし，機械学習向けの MLlib やグラフデータ処理向けの

[1] 国立研究開発法人 産業技術総合研究所 人工知能研究センター
[a)] yusuke.tanimura@aist.go.jp

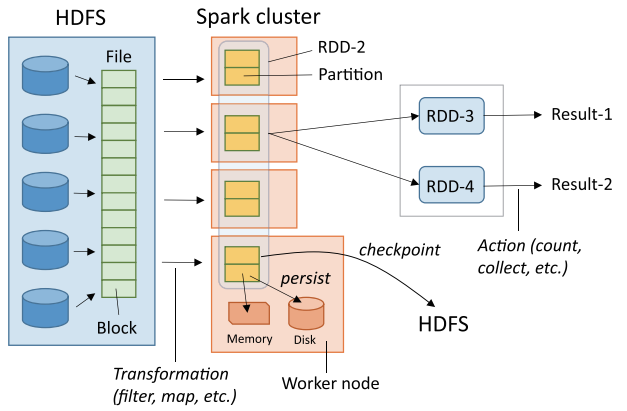

図 1 RDD の概要

GraphX，SQL 処理やストリーミング処理用などの用途別の上位ライブラリ，データ入出力を司るストレージ，実行環境を管理するスケジューラなど複数のコンポーネントから構成される．

Spark におけるデータ入力は，ファイルやデータベースからの一括入力，および Kafka [8] などを介したストリーミング入力が利用できる．ファイル入力の場合は Hadoop Distributed File System（HDFS） [9] や S3 [10] 互換のストレージなどが利用できる．データ出力についてもファイルやデータベースへの一括出力が利用できる．

Spark アプリケーションの実行は，単一マシン上でも可能であるが，大きなデータを扱う場合にはマスタ・ワーカからなるクラスタ環境で行うことになる．クラスタの起動には，SSH を利用した起動スクリプトを用いるスタンドアローンモードのほか，Hadoop の資源管理フレームワークである YARN [11] や Apache Mesos [12, 13] を利用する方法がある．

2.2 RDD の仕組み

Spark では RDD（Resilient Distributed Dataset）と呼ばれるデータ構造が用いられる（**図 1**）．RDD はパーティションに分割された，読み取り専用のレコードの集合である．パーティションに分割されていることにより，それぞれを複数のマシンで並列に処理できる．Spark のユーザは，例えば行単位でレコードを保持する HDFS 上のファイルを RDD として定義し，それに map，filter，join などの操作を適用した別の新しい RDD を定義するというようにデータフローを記述する．ただし，その実行においては各操作は Spark によって遅延処理される．アクションと呼ばれる操作が呼ばれてから各操作の処理が開始される．

各操作が適用された結果，すなわち生成された各 RDD は基本的にメモリ上に展開されて処理が進む．このため，障害発生時にはいくつかの RDD，またはそれらのパーティションの一部が失われる可能性がある．これに対して，Spark

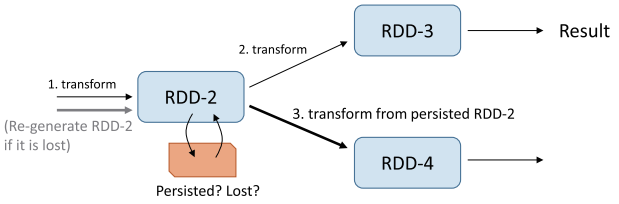

図 2 RDD の再利用の例

は RDD の生成手順 *1 を記憶しておき，パーティション単位で再処理を行って RDD を再生成することで耐障害性を確保している．

加えて，ある RDD に対して反復操作が適用される場合や RDD の生成に時間がかかる場合などのために，その RDD を継続して保持する仕組み（RDD Persistence）が用意されている．RDD を保持するストレージとしては，メモリあるいはディスク，またはその両方を選ぶことができる．また，データの複製の有無や，メモリに保持する場合にシリアライズの有無を指定することもできる．例えば，メモリ上にデータを保持しておくことで，以前の処理で生成した RDD を再利用し，速やかに次の結果を得ることが可能になる（**図 2**）．

2.3 RDD のストレージへの書き込み

RDD をストレージに書き込むケースとしては，前述のディスク上に RDD を継続保持する場合（Persist）とチェックポイント（Checkpoint）がある．書き込む先のストレージにはワーカノード上のローカルディスクや HDFS が利用可能である．Persist については，シャッフルを伴う操作において Spark が自動的にローカルディスクへの書き出しを行う場合と，ユーザが明示的に persist() メソッドを呼び出し，ストレージレベルとしてディスクを指定する場合がある．Checkpoint はユーザが明示的に checkpoint() メソッドを呼び出した場合に実行される．

Persist と Checkpoint はいくつかの点で違いがある．Persist は各 RDD が生成された時点で実行され，Persistence が確保された後も RDD の生成手順を保持し続ける．ただし，Spark ジョブが終了した後は，ディスクへ書き出していたとしてもそのジョブによって作られた RDD は全て削除される．それに対して，Checkpoint はジョブの最後に実行され，チェックポイントが完了した後は RDD の生成手順を保持しない．ストレージに保存された RDD は Spark ジョブの完了後も残り，ユーザが手動で削除を行うまで利用可能である．

3. 評価方法

3.1 評価の指針

本研究では Spark のストレージ・コンポーネントがアプ

*1 lineage graph と呼ばれる．

表 1　評価実験に用いたマシンのスペック

Management node	AMD Opteron 6128 CPU (2GHz, 8 cores), 32GB memory, Intel X520-DA2 (10 GbE), OCZ Vertex2 (100GB) connected through LSI-Logic MegaRAID SAS 9260-8i, CentOS 6.7.
Worker node	Intel Xeon E3-1230 (3.2GHz, 4 cores) CPU, 8GB memory, Intel X520-DA2 (10 GbE), OCZ Vertex3 (240GB)×2 or OCZ Vertex3 (240GB) and Hitachi Travelstar 7K320 (All disks are connected through SATA 2.0.), CentOS 6.7.

リケーションの実行に与える影響を調査し，ユーザが自身のアプリケーションの性能チューニングを行ったり，システム管理者が最適な Spark 実行環境を構築・運用したりするのに有用な指針を得ることを目的としている．具体的には，

- Spark のワーカノードに高速なディスクを搭載することの有用性
- Spark 利用時に，HDFS のインタフェースを持つ共有ストレージシステムに求められる性能
- HDFS のデータノードと Spark のワーカノードをオーバラップさせた場合の効果

を明らかにすることを目指している．

本稿ではその最初のステップとして，Persist および Checkpoint の性能，すなわち各ストレージへの書き出し性能を評価するための実験を行った．そして，RDD のサイズ，用いるストレージデバイス，シリアライザの違いによる性能の違いを明らかにした．

3.2　ストレージへの出力性能の測定方法

Persist や Checkpoint を実行する Spark のプログラムでは，Key-Value ペアに見立てた文字列を指定数行読み込み，map 処理において Value 部分を増やすことで，任意のサイズの RDD を用意することにした．これはドライバプログラムからワーカノードへのデータ転送を減らすとともに，HDFS のブロックサイズに応じたパーティションの自動分割の適用を受けずに，1 つの大きなパーティションの RDD を生成するためである．なお，1 パーティションに限定するのは，処理を単純化し，性能調査を行いやすくするためである．本 Spark プログラムでは，そうして作成した RDD に対して persist() あるいは checkpoint() を実行する．

ストレージへの出力性能の計測は OS またはストレージ側で行う方法を採用した．この方法を採用した主な理由は，いずれのアクセスも Spark 内部で適宜実行されるため，Spark のアプリケーションプログラムでは計測不能であり，また Spark のソースコードを改変しての計測には手間がかかるためである．

Persist によるローカルディスクへのアクセスは，SystemTap [14] を用いて Linux カーネルから open，write，close のシステムコールの情報を取得することにより，ディスクへの書き出し速度を測定した．ただし，Persist の性能は RDD の操作と一体であるため，実際に測定できる性能は map 操作を含んだ値となる．

表 2　ディスク性能

	Sequential write (MiB/sec)	Random write (IOPS)
SSD	228	28902
HDD	49.9	243

Checkpoint による HDFS へのアクセスは，HDFS のネームノードのログからデータブロックの割り当て時刻とファイル操作の完了時刻を取得して性能を計算した．なお，Checkpoint を実行するプログラムでは，RDD をメモリ上にキャッシュする persist() を実行した上で checkpoint() を呼び出しており，RDD の再生成を含まないストレージへの書き出し性能のみを測定している．

また，Spark ではディスクへの RDD の書き出しはシリアライズされるため，用いるシリアライザが性能に影響を及ぼす可能性がある．Spark では Java Serialization と Kryo Serialization [15] が利用できるため，両者についての比較も行うことにする．Spark におけるデフォルトの設定では前者が用いられるが，速度は後者が優れていると言われている [16]．

3.3　評価環境

評価実験は**表 1** に示すスペックのマシンを用いて行った．表中の Management node は 1 ノード用意し，そこで HDFS の Namenode と YARN の Resource Manager を起動した．Worker node は 7 ノード用意し，HDFS の Datanode と YARN の Node Manager を起動した．用いた Spark，HDFS，YARN は全て Cloudera が配布する CDH 5.5 [17] に同梱のものである．Persist や Checkpoint を実行するプログラムは Scala で記述しており，Scala は v.2.11.7 を用いた．Java は v.1.8.0_66 を用いた．

Spark から利用可能な Worker node 上のディスクは SSD（OCZ Vertex3），または HDD（Hitachi Travelstar 7K320）とした．それぞれを Ext4 でフォーマットし，8KiB 単位のアクセスを行った際の各ディスクの性能は**表 2** の通りであり，両者の性能には明らかな違いがある．

HDFS は Worker node 上の別の SSD（OCZ Vertex3）を用いて構築し，レプリカ数を 3，ブロックサイズを 128MiB に設定した．1000MiB のファイルの書き込みスループットは平均 427MiB/sec であった．

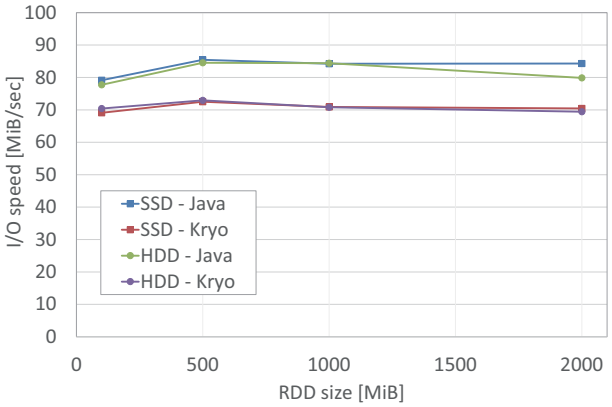

図 3 Persist の測定結果

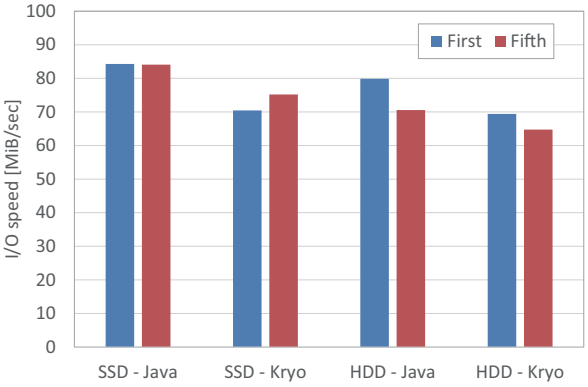

図 4 計 8000MiB の RDD を書き出した後の Persist（2000MiB）の測定結果

4. 評価結果

4.1 Persist の性能評価

本評価ではレコード数を 10,000 と設定して，3.2 節で述べた方法で指定したサイズの RDD を用意した．ただし，Spark RDD では 1 パーティションの上限値が 2GiB であることから，本評価に用いた RDD の最大サイズは 2000MiB とした．

図 3 は，RDD サイズの増加に対して各ケースの性能を示したグラフである．RDD サイズが 100MiB の時に若干低い性能となるのは，サイズが小さいことにより相対的にオーバーヘッドが大きくなっているためと考えられる．シリアライザの比較では，Kryo Serialization よりも Java Serialization の方が良い結果となった．これは用いた RDD が単純な構造であるために，Kryo Serialization の効果が表れなかったのではないかと考えている．

SSD と HDD の比較では，RDD サイズの増加に対して SSD の結果は性能を維持しているが，HDD の結果は若干の性能低下が見られる．表 2 のディスク性能や搭載メモリの量を考慮すれば，HDD の結果は OS 側のキャッシュの効果が含まれていると考えるのが妥当であり，RDD サイズの増加とともにキャッシュ効果が減少したと考えることができる．この仮定を検証するため，2000MiB の RDD を複数個生成し，全てに対して Persist を実行した．**図 4** は，2000MiB の RDD を 4 回書き出した後の 5 回目の RDD の書き出し性能を測定し，1 回目の RDD の書き出し性能と比較した結果である．この結果より，SSD を利用した場合は性能が維持されているが，HDD を利用した場合は 7~12%の性能低下が発生していることが確認できる．

4.2 Checkpoint の性能評価

Checkpoint についても，前節の Persist の評価実験と同様にレコード数を 10,000 に指定して RDD を生成した．**図 5** は RDD サイズに対する各ケースの性能を示している．図中の DFSIO は，Hadoop に含まれる DFSIO ベンチマークにより測定した HDFS の性能であり，Spark アプリケーションが享受できる最大性能を示している．本評価結果においても，Java Serialization の方が Kryo Serialization よりも高速であった．特に，1000MiB の RDD サイズの結果は，DFSIO の結果，すなわち RDD ではないファイルの HDFS への書き出し性能に近い結果であった．

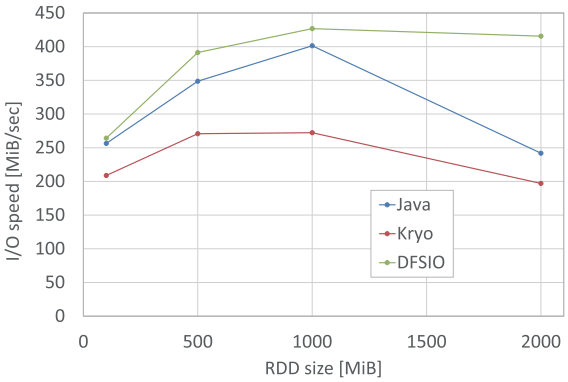

図 5 Checkpoint の測定結果

4.3 議論

本評価では測定を単純にするために RDD のパーティション数を 1 としたが，実際の利用においては多数のパーティションを扱うことになる．パーティションが複数になった場合は通常，複数のワーカノードが用いられ，それぞれでローカルディスクまたは HDFS へのアクセスが行われるだけである．そのうちの 1 つの RDD パーティションの書き込み性能は本評価結果と同様になると考える．HDFS から大きなデータを読み込んで処理するケースでは，ブロックサイズに応じたパーティション数が設定され，それは 128MiB ないし 256MiB が一般的である．2000MiB の大きな RDD を保持するようなケースは，Join などの操作により中間データが増える場合である．2000MiB は大規模な解析処理にとっては小さい値であり，Spark の開発コミュニティに対して修正要求が出されているが，現時点ではまだ対策がなされていない．ユーザ側で RDD のパー

ティションを適切に増やしておくことが必要になる．

SSD と HDD の比較では SSD の方が性能が良い結果となったが，その差はわずかである．コストパフォーマンスでは HDD の方が優れていると言える．

シリアライザに関しては，今回はベンチマーク用の疑似データを用いたこともあり，実際的なアプリケーションデータを用いて追加の検証実験が必要だと考えている．

5. まとめと今後の課題

本稿では Spark の Persist と Checkpoint に焦点を当て，RDD のストレージへの書き出し性能について調査した結果を報告した．Checkpoint の性能はバックエンドのストレージ性能がかなり反映された結果であったのに対し，Persist の性能はバックエンドのストレージ性能があまり反映されない結果となった．これは Spark システムを構築する上で考慮すべき点の 1 つと言えるだろう．

今後の課題としては，実際的な Spark アプリケーションを用いて今回と同様の性能評価を進めるとともに，Persist や Checkpoint が実際にどのような頻度・形態で用いられているのかの調査が必要であると考えている．また，ストレージからのデータの読み込み性能や，Parquet [18] や SequenceFile などのデータフォーマットによる違いについても評価が必要であると考えており，これらについても取り組んでいきたい．

謝辞 本研究の一部は，NEDO の委託業務「次世代ロボット中核技術開発プロジェクト」の支援を受けて実施した．

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 Workshop on Hot Topics in Cloud Computing* (2010).
[2] 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 Symposium on Networked Systems Design and Implementation* (2012).
[3] Apache Spark: http://spark.apache.org/.
[4] Apache Hadoop: https://hadoop.apache.org/.
[5] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th Symposium on Operating Systems Design and Implimentation* (2004).
[6] AMPLab: https://amplab.cs.berkeley.edu/.
[7] The Apache Software Foundation: http://www.apache.org/.
[8] Apache Kafka: http://kafka.apache.org/.
[9] Hadoop Distributed File System: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[10] Amazon S3: http://aws.amazon.com/s3/.
[11] Hadoop YARN: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[12] Apache Mesos: http://mesos.apache.org/.
[13] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. and Stoica, S. S. I.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pp. 295–308 (2011).
[14] SystemTap: https://sourceware.org/systemtap/.
[15] Kryo: https://github.com/EsotericSoftware/kryo.
[16] Tuning Spark: http://spark.apache.org/docs/latest/tuning.html.
[17] Cloudera Distribution for Hadoop (CDH): http://www.cloudera.com/downloads/cdh.html.
[18] Apache Parquet: https://parquet.apache.org/.