修士学位論文
並列分散処理フレームワーク Spark を用いた TF-IDF 計算手法の提案

首都大学東京大学院理工学研究科
数理情報科学専攻
西村　建郎

目次

1　序論

1.1　研究背景

現在、シングルコアでの性能向上が技術的な理由で打ち止めとなり、マルチコア化が性能向上の主流となっている。しかし、マルチコア化だけでは年々増加するビッグデータの分析を行うには限界がある。そこで数年前から、コモディティ*1サーバを複数台用いてクラスタ*2を構築し、膨大なデータを高速に処理する並列分散処理フレームワークが開発されるようになった。以前から並列分散処理は行われてきたが、特殊で高価なサーバやストレージを必要としたため、限られた場所でしか使用できなかった。しかし、2008 年頃 Hadoop[1] が登場し、特殊なハードウェアを必要としない、コモディティサーバを用いた並列分散処理が可能になった [2]。Hadoop は当初 MapReduce というフレームワークに沿って処理を行うものであった。MapReduce は強力であったが、すべての処理を Map 処理と Reduce 処理に変換しなければならず、あまり高速化が期待できない処理も多くあった。そこで 2013 年頃、Map 処理と Reduce 処理に制限されないさまざまな処理を可能にした、新しい並列分散処理フレームワーク Spark[3] が登場した。Spark はオープンソースプロジェクトであり、世界中の技術者によって現在も開発が続けられている。Spark は MapReduce と異なり、連続する処理をディスクを通さずメモリ上で行うことが出来る等の理由から、MapReduce よりも高速に動作する [4]。また Spark は、MapReduce 以外の処理も可能になっており、従来の Map 処理、Reduce 処理に縛られず、シングルスレッドのプログラミングと同じ感覚で処理を実装できる。

本研究室では今まで並列処理について研究してきたが、今年度から並列処理のテキストマイニングへの応用に取り組み始めた。テキストマイニングとは、蓄積された膨大なテキストデータを何らかの単位（文字、単語、フレーズ）に分解し、これらの関係を定量的に分析することである [5]。テキストマイニングを行う際、テキストをベクトルに変換することがよく行われる。ベクトルに変換することで、テキスト同士の距離が定義され、クラスタリング*3等が可能になる。最も基本的な変換方法は、各単語をベクトルのある次元に対応させ、その単語の出現回数 (TF:term frequency) を対応する次元の値とするものである。さらに、テキストの持つ特徴をより強調するために、その単語の出現回数に重み (IDF:inverse document frequency) を掛け合わせたものを、対応する次元の値とする TF-IDF と呼ばれる方法もある [5]。この重み付けを行うと、多くのテキストで出現する汎用的な単語に対応する値は小さくなり、少ないテキストでのみ出現する特徴的な単語に対応する値は大きくなる。しかし、並列分散処理でテキストのベクトル化を行う場合、すべてのサーバにおいて、各単語を同じ次元に対応させなければいけないため、簡単には処理できない。Spark には MLlib というライブラリが用意されており、それを用いるとテキストを TF-IDF ベクトルに変換

*1 一般的に入手可能であること
*2 複数のサーバを連結し、全体で一台のサーバであるかのように振舞うシステム
*3 あるデータの集合を、ある共通の特徴を持つ部分集合に分割すること

できる [4]。MLlib を用いた手法は高速に動作するが、生成された TF-IDF ベクトルから重要単語等を抽出することが出来ないため分析方法が制限されてしまうことと、ベクトルに変換する際に同じ次元に異なる単語が対応し衝突する危険性があり、正確さを要求する処理には使用できないという短所があることが分かった。

1.2　今回行ったこと

そこで今回 Spark を用いて、これらの短所を改善し、かつ実行時間の増加を最小限に留めた TF-IDF 計算手法を提案した。提案手法は、TF-IDF ベクトルだけでなく単語辞書という副産物も生成するため、TF-IDF ベクトルから重要単語の抽出を可能にし、また単語辞書を用いたさまざまな分析も可能にした。さらに、単語が衝突する危険性もないため、より正確な分析が可能となる。また、最大 7 台のサーバ（マスターサーバ:1 台, スレーブサーバ:6 台）でクラスタを構築し、提案手法を実装し MLlib を用いた手法とのスケーラビリティを比較した。

2 Spark

本章では、Spark の概要とプログラミング方法について述べる [4]。

2.1 Spark 概要

Spark は並列分散処理フレームワークであり、複数台のコモディティサーバを用いてクラスタを構築し、高速で汎用的な処理を行うことが出来る。処理は、Java、Python、Scala を用いて実装でき、後述する RDD という概念を利用することで、シングルスレッドのプログラミングと同じ感覚で並列分散処理を実装することが出来る。Spark はマスター/スレーブ構成のクラスタを構築し、マスターサーバがスレーブサーバに指示を出し、スレーブサーバが実際に演算をするという関係になっている。

マスターサーバとスレーブサーバの主な役割を以下に列挙する

＜マスターサーバ＞

- アプリケーションの main 関数の実行
- 並列分散処理部分を個々のスレーブサーバが実行するタスクに変換
- スレーブサーバが実行する個々のタスクのスケジュール調整

＜スレーブサーバ＞

- 割り当てられたタスクを実行し、結果をマスターサーバに返す

Spark は、スレーブサーバの数を 1 台〜数 1,000 台まで効率的にスケールアウト[*4]出来るように設計されている。

2.2 クラスタマネージャ

Spark を動作させるためにはクラスタマネージャが必要となる。これは、マスターサーバがタスクのスケジュール調整をするための機構であり、スレーブサーバの CPU やメモリ等のリソース管理を担う。Spark が動作するクラスタマネージャは、Hadoop に付属する YARN[2]、Mesos[6]、Spark 自身が持つ Standalone Scheduler がある。YARN と Mesos は Spark 以外の並列分散処理フレームワークも実行できる汎用的なものであるが、本研究では Spark しか用いないため、Standalone Scheduler を用いる。

[*4] サーバの台数を増やすことで全体の処理速度を向上させること

2.3 HDFS

Spark で処理を行う際、ファイルの入出力は HDFS(Hadoop distributed file system)[2] と呼ばれるファイルシステムを用いる。HDFS はクラスタ内で動作し、各サーバの一部分のディスク領域を束ねて、ひとつの巨大なファイルシステムと見なせるようにしたものである。HDFS を用いると、データがどのサーバに格納されているかを気にすることなく、ファイルシステム上のデータを読み書き出来る。HDFS は巨大なデータを保存するために設計されており、1 つのファイルを複数のブロックと呼ばれる単位 (64MB,128MB 等自由に設定可能) に分割して、複数のサーバに分散して格納する。HDFS は Spark と同じクラスタ上で動作させることができ、マスターサーバには全ファイルのメタデータ*5を、スレーブサーバには実際のデータを格納する。

HDFS は最大で数 1,000 台のサーバを用いた巨大なクラスタ上で動作させることが出来る。しかし、サーバが多くなるほどクラスタ内で不具合が発生する可能性が高まる。この問題に対処するために、HDFS にはレプリケーション数という設定項目があり (初期値:3)、常に全てのブロックはレプリケーション数の複製をクラスタ内に保持する。例えばレプリケーション数が 3 であれば、常に全てのブロックはクラスタ内のスレーブサーバ 3 台に保持され、処理中にスレーブサーバで不具合やデータの損失等が起こった場合は、正常に動作しているサーバからデータをコピーし、処理を継続することが出来る。

2.4 RDD 概要

RDD(Resilient Distributed Dataset) は、Spark の中核をなす概念である。Spark ではデータを RDD という抽象概念として表現し、それに対してさまざまな処理を行っていく。RDD には、ユーザーが定義したものを含め、Python、Java、Scala の任意のオブジェクトのリストを持たせることができ、これらのデータは自動的にクラスタ内で分散され、RDD に対しての操作は自動で並列化される。具体的には、RDD は複数のパーティションと呼ばれる単位に分割されており、各パーティションに対する演算処理が、各パーティションを保持しているサーバ内で実行される。また resilient という言葉は、RDD を保持しているサーバに障害があった場合も、失われた部分を再計算して処理を継続できることに由来する。

Spark を用いたアプリケーションは、次の流れで処理を行う。

(i) RDD の生成
(ii) RDD の変換
(iii) アクション

次節よりそれぞれの詳細について述べる。

*5 ファイル名やデータ形式等、データ自身についての付加的なデータ

2.5 RDD の生成

すべての Spark アプリケーションには、ドライバプログラムというものが含まれる。ドライバプログラムはアプリケーションの main 関数を持ち、マスターサーバで実行される。その main 関数内で 1 つ以上の RDD を生成し、それらに対して処理を行っていく。RDD を生成する方法は 2 種類あり、1 つはドライバプログラムからリスト等を直接ロードすることであり、もう 1 つは外部のデータセットをロードすることである。以下 RDD 生成例を示す。本研究の実装はすべて Java を用いたため、以後のソースコードはすべて Java である。

ソースコード 1　RDD 生成例

```
1 SparkConf conf = new SparkConf();
2 JavaSparkContext sc = new JavaSparkContext(conf);
3 JavaRDD<Integer> input1 = sc.parallelize(Arrays.asList(1,2,3,4,5));
4 JavaRDD<String> input2 = sc.textFile(”hdfs://filepath/file1”);
5 JavaPairRDD<Text,IntWritable> input3 = sc.sequenceFile(”hdfs://filepath/file2”,Text.
      class,IntWritable.class);
```

1 行目: Spark の設定オブジェクトを生成している。Spark の設定を変更する際はここで記述する。

2 行目: SparkContext オブジェクトを生成している。このオブジェクトを通じて、スレーブサーバへの接続を実現する。

3 行目: SparkContext の parallelize() メソッドを用いて、ドライバプログラムで生成したリストを直接ロードしている。

4 行目: SparkContext の textFile() メソッドを用いて、HDFS に格納されているテキストファイルを読み込んで、ファイルの各行を要素とする RDD を生成している。

5 行目: SparkContext の sequenceFile() メソッドを用いて、HDFS に格納されている sequence ファイルを読み込んで PairRDD を生成している。

sequence ファイルとは<key,value>ペア形式を要素とするリストを格納するためのファイル形式であり、PairRDD とは<key,value>ペア形式を要素とする RDD である。

Java において RDD は JavaRDD<A>または JavaPairRDD<K,V>という型で表現される。ここで A,K,V は、RDD の型 (Java オブジェクトの型) を表している。sequenceFile() メソッドを用いるときは、引数としてファイルパスの他に key と value の型を表すクラスファイルを指定しなければならない。sequence ファイルから読み込む際の型は特殊であり、Text は String、IntWritable は Integer とほぼ同等である。

2.6 RDD の操作

RDD に対して、変換とアクションという 2 つの操作が行える。変換は、ある RDD に何らかの処理を行い新しい RDD を生成する。アクションは、ある RDD を基に結果を表示したり、結果をファイルシステム（HDFS 等）に保存したりする。RDD に対する変換、アクションはさまざまなものが用意されており、どちらも RDD に対するメソッドを呼び出すことで実行する。本研究で利用するものについて、図 2,3 にまとめた。図 2,3 の実行結果については、ソースコード 1 の input1 〜input3 を用い、file1 と file2 の内容は図 1 の通りとする。また、map() メソッド、flatMap() メソッドに対する引数にはラムダ式 [7] を用いている。

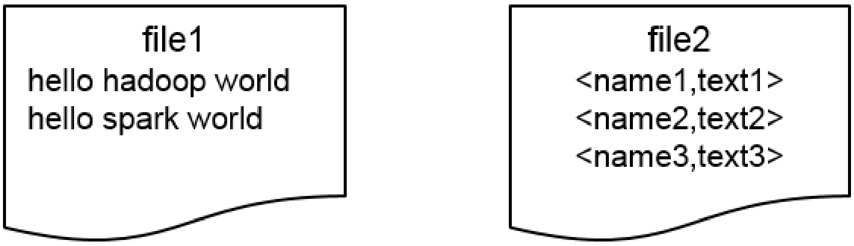


図 1　ファイル例

変換メソッド（引数）	入力型⇒出力型	説明	実行結果
map(f:A⇒B)	RDD[A] ⇒RDD[B]	関数fをRDDの各要素に適用させる	input1.map(x->x+1) ⇒[2,3,4,5,6]
flatMap(f:A ⇒List[B])	RDD[A] ⇒RDD[B]	関数fをRDDの各要素に適用させ、生成した各リストを連結したRDDを生成する	input2.flatMap(x-> Arrays.asList(x.split(" "))) ⇒[hello,hadoop,world, hello,spark,world]
values	PairRDD[K,V] ⇒RDD[V]	PairRDD<K,V>を与え、Vを要素とするRDDを返す	input3.values() ⇒[text1,text2,text3]

図 2　RDD に対する変換例

アクションメソッド	入力型⇒出力型	説明	実行結果
count	RDD⇒int	RDDの要素数を得る	input1.count()⇒5
countByValue	RDD<A> ⇒Map<A,long> *Map,long: Javaのデータ型	RDD<A>に対して、各要素の出現回数をvalueとするMap<A,value>を返す	input2.flatMap(x-> Arrays.asList(x.split(" "))) .countByValue() ⇒{<hello,2>,<hadoop,1>, <world,2>,<spark,1>}
saveAsObjectFile	RDD⇒ファイル	RDDをファイルとして格納する	ファイル生成

図 3　RDD に対するアクション

ソースコード 2　遅延評価の例

```
1 JavaRDD<String> input = sc.textFile("hdfs://filepath/file1");
2 JavaRDD<String> wordList = input.flatMap(x->Arrays.asList(x.split(" "))).persist(
      StorageLevel.MEMORY_ONLY());
3 System.out.println(wordList.count());
4 wordList.saveAsObjectFile("hdfs://filepath");
```

変換とアクションが区別されているのは、RDD に対する Spark の演算処理のやり方が異なるためである。RDD の生成と変換は遅延評価で処理され、アクションでその RDD が使われる時点で初めて実際に生成や変換が行われる。これは、無駄な計算を省くための性質である。しかし、複数のアクションを同じ RDD に対して実行した場合は、デフォルトだとアクションが実行される度に RDD が計算し直されるために、非効率となる。RDD を複数のアクションで利用したい場合は、persist() メソッドを用いることで、メモリやディスク上に RDD を保存し、無駄な計算を省くことが出来る。persist() は変換でもアクションでもなく、RDD をキャッシュ[*6]しておくためのメソッドであり、引数として格納するストレージレベル（メモリのみ:MEMORY_ONLY(), メモリに格納するが、足りなくなったらディスクにも格納する:MEMORY_AND_DISK(), ディスクのみ:DISK_ONLY()）等を与える。

遅延評価の例をソースコード 2 に示す。1～2 行目は RDD 生成と変換であり、この時点では実際の RDD は生成されていない。3 行目で、count() アクションが呼ばれたので、1～2 行目の RDD が実際に処理され、アクションを実行する。ここで、2 行目の RDD では persist() が呼ばれているので、wordList という RDD はメモリにキャッシュされる。このため、4 行目のアクションで wordList を利用する際、再度 RDD を計算し直さず、メモリから直接読み込める。

[*6] 再度利用するために、一時的にデータをメモリやディスクに保持しておくこと

2.7　ブロードキャスト変数

Spark では、ブロードキャスト変数と呼ばれる共有変数が利用できる。ブロードキャスト変数は、データをプログラム中で効率的にマスターサーバからすべてのスレーブサーバに送信し、Spark の処理で使ってもらうためのものである。具体的な使用法は 4 章で述べる。

3　テキストからの特徴ベクトル抽出

本章では、TF-IDF と、MLlib を用いた TF-IDF ベクトル計算手法について述べる [4][5]。

3.1　特徴ベクトル

テキストマイニングを行う際、テキストを特徴ベクトルに変換することがよく行われる。特徴ベクトルとは、分析したいデータに対する複数の特徴を一つにまとめてベクトルとして表現したものである。テキストを特徴ベクトルに変換すると、テキスト同士の距離を定義することができ、クラスタリング等に応用できる。テキストデータに対する最も基本的な特徴ベクトルは、各単語をベクトルのある次元に対応させ、その単語の出現回数 (TF:term frequency) をその次元の値とするものである。この方法で生成された特徴ベクトルを TF ベクトルと呼ぶ。

3.2　TF-IDF

テキストの特徴をより強調するために、特徴ベクトルとして、TF ベクトルに IDF(inverse document frequency) と呼ばれる重みを掛け合わせた TF-IDF ベクトルを用いる方法がある。IDF は、その単語が含まれるテキスト数 (DF:document frequency) に基づいて計算される値であり、多くのテキストで出現する汎用的な単語ほど小さくなり、少ないテキストでしか現れない特徴的な単語ほど大きな値となる。一般的に用いられる TF-IDF の定義を以下に示す。

T:総テキスト数
TF_{wt}:テキスト t 内での単語 w の出現回数
DF_w:全テキストの中で、単語 w が含まれるテキスト数

$$IDF_w = \log \frac{T}{DF_w}$$

$$TFIDF_{wt} = TF_{wt} IDF_w$$

TF-IDF に厳密な定義はなく、TF の値がテキストの長さに依存しないように、TF を各テキストの単語数で割ってから IDF を掛ける流儀等もある。

また、後述する MLlib を用いた手法では IDF の計算の際 DF_w が 0 になる可能性があるため、スムージングという操作を行い、分母分子に 1 を足す処理を加えている。

3.3　MLlib を用いた特徴ベクトル抽出

Spark には、MLlib というライブラリが用意されており、これを用いるとさまざまな処理が簡潔な記述で実装できる。本節では、MLlib を用いた TF-IDF ベクトル計算手法を紹介する。大量のテキストデータを扱うときは、key をファイル名、value をテキストデータとする sequence ファイルを用いることが多い。

入力データを、key をファイル名、value をテキストデータとする sequence ファイルとした時の、MLlib を用いたプログラムを以下に示す。

ソースコード 3　MLlib を用いた TF-IDF ベクトル生成

```
1 JavaPairRDD<Text> input = sc.sequenceFile(inputFile, Text.class, Text.class).values();
2 JavaRDD<List<String>> wordLists = input.map(new WordsList());
3 HashingTF tf = new HashingTF();
4 JavaRDD<Vector> tfVectors = tf.transform(wordLists). persist(StorageLevel.
      MEMORY_ONLY());
5 IDFModel idfModel = new IDF().fit(tfVectors);
6 JavaRDD<Vector> tfIdfVectors = idfModel.transform(tfVectors);
```

1 行目: sequence ファイルから key をファイル名、value をテキストデータとする PairRDD を生成し、それに対して values() メソッドを用いることでテキストデータをのみを要素とする RDD を生成する。

2 行目： MLlib にはテキストを単語に分割する機能は提供されていないため、自分で実装する必要がある。ここではテキストを入力とし、単語に分割したリストを出力する自作関数を、map の引数に与えている。ここでの関数は複雑であり、ラムダ式を用いると見にくくなってしまうため、クラスとして別に定義している。

3 行目: TF ベクトルを生成するためのオブジェクトを生成している。引数として生成するベクトルの次元数を与えることが出来る。デフォルトは 2^{20} となっている。

4 行目: TF ベクトルを要素とする RDD を生成している。この RDD は 5,6 行目で二度利用するため、persist() メソッドを用いてメモリにキャッシュしている。

5 行目: 4 行目で生成した TF ベクトルを基に、各単語に対する IDF を計算する。

6 行目: TF ベクトルに IDF を掛け合わせ、TF-IDF ベクトルを要素とする RDD を生成する。

tfVectors と tfIdfVectors はどちらも MLlib で定義された Vector 型の RDD であるが、この Vector はインターフェイスであり、DenseVector と SparseVector の二種類の実装を持つ。DenseVector はベクトルの要素をすべて格納する形式で、SparseVector は 0 以外の要素とその添字のみを格納する形式であり、疎ベクトルを扱う場合に適している。TF ベクトルや TF-IDF ベクトルはどちらもほとんどの場合疎ベクトルであるため、SparseVector が使われている。

MLlib では特徴ベクトルを生成する際、各単語に対してハッシュ関数を用いて、ある次元に対応させている。この方法だと、高速にベクトル化出来るが、生成されたベクトルから各次元がどの単語に対応しているのかを知ることが出来ないため、テキスト間の距離を計るためだけであれば良いが、ベクトルを用いてその他のさまざまな処理を行うことは難しい（最頻出の単語・重要度の高い単語の抽出等)。また、単語をある次元に対応させる際、同じ次元に異なる単語が対応し衝突する危険性があるため、正確さを要求する処理には使用できない。

4　提案手法

前章より、MLlib を用いた TF-IDF の計算にはいくつかの短所があることが分かった。本章では、それらの短所を改善した Spark を用いた TF-IDF の計算手法を提案する。

入力データとして、テキストファイル名を key、テキストデータを value とする以下のような sequence ファイルを用い、出力は MLlib を用いた手法と同様、SparseVector を要素とする以下のような RDD とする。

＜入力＞　`sequence ファイル : [<filename1, text1>,<filename2, text2>・・・]`
＜出力＞　`RDD : [SparseVector1,SparseVector2,・・・]`

提案手法は以下の 5 ステップから成る。

提案手法におけるデータの流れを図 4 に示す。図 4 ではスレーブサーバ 2 台しか描かれていないが、スレーブサーバが増えた場合も同様である。

説明の中で出てくる Map とは、<key,value>形式の集合を格納することを表す Java のインターフェイス型であり、本提案手法では、Map インターフェイスを継承した HashMap と TreeMap の 2 種類の実装を使い分けている [7]。これらの違いを図 5 に示す。これらを特に区別する必要がないときは総称して Map と呼ぶことにする。

- STEP1（並列分散処理 + サーバ間通信）
  全テキストから単語を抽出し、各テキストの単語リストを生成しキャッシュしておく。次に各テキストで重複しない単語のみを抽出し、それらをマスターサーバに集約し、key を単語、value を DF とする Map（DF マップ）を生成する
- STEP2（マスターサーバ）
  STEP1 で生成した Map（DF マップ）の各単語に通し番号を付与し、key を単語、value を Pair<DF, 通し番号>[*7]とする新しい Map（単語辞書）を生成する
- STEP3（サーバ間通信）
  STEP2 で生成した Map（単語辞書）をすべてのスレーブサーバにコピーする
- STEP4（並列分散処理）
  各スレーブサーバにおいて、STEP1 でキャッシュしておいた単語リストから、各テキストに対して key を単語、value を TF とする Map（TF マップ）を生成する
- STEP5（並列分散処理）
  各スレーブサーバにおいて、STEP4 で生成した Map（TF マップ）と、STEP3 でコピーした Map（単語辞書）を用いて、TF-IDF ベクトルを計算する

[*7] 通し番号と DF のペアを格納するための自作クラス

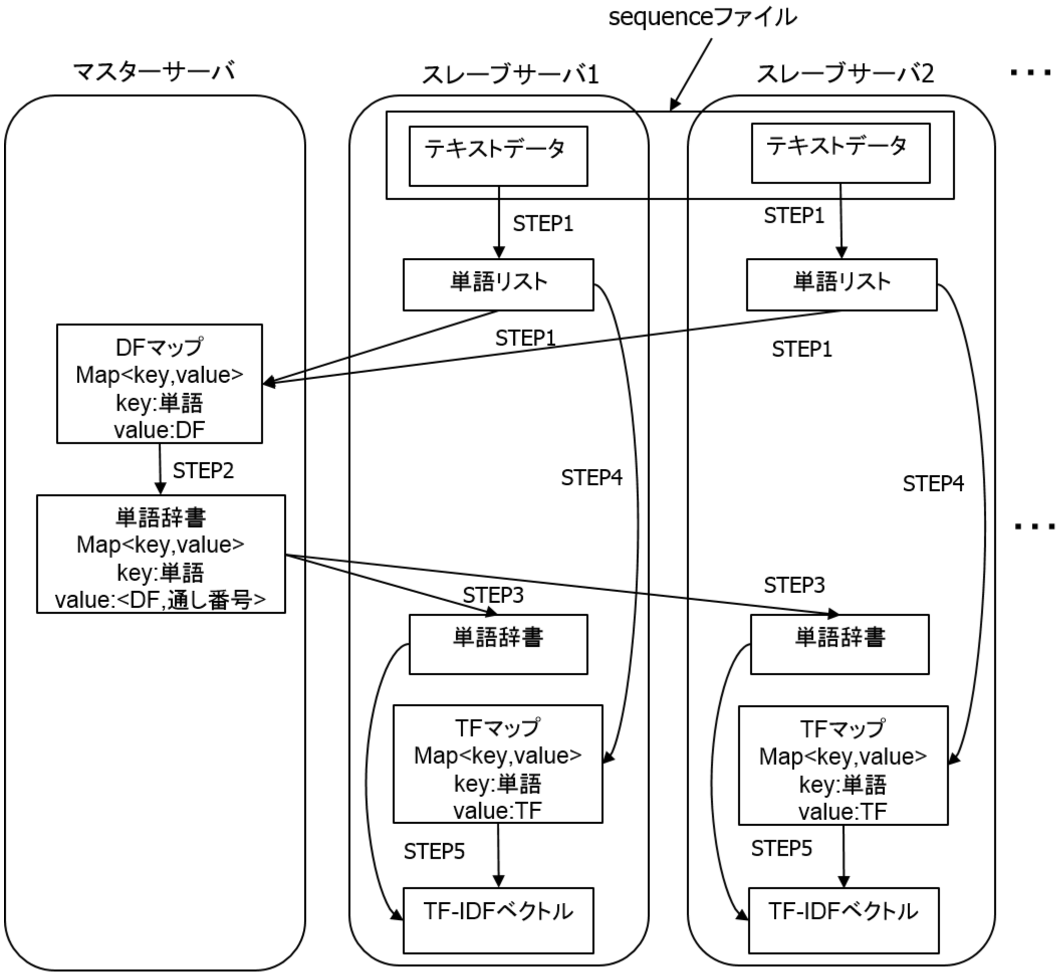


図 4　提案手法のデータの流れ

Map実装	内部構造	利点
HashMap	配列（ハッシュ表）	追加や検索が速い
TreeMap	木構造	任意の比較規則に従った順序でデータを取り出せる（辞書順等）

図 5　本研究で用いた Java の Map 実装

以下それぞれのステップについて実装の詳細を述べる。

4.1 STEP1:DF マップ生成

ソースコード 4 STEP1

```
1 JavaPairRDD<Text> text_rdd = sc.sequenceFile(inputFile, Text.class, Text.class).values();
2 JavaRDD<List<String>> words_rdd = text_rdd.map(new WordsList()).persist(
      StorageLevel.DISK_ONLY());
3 JavaRDD<String> unique_words_rdd = words_rdd.flatMap(new UniqueWordsList());
4 Map<String, Long> word_df = unique_words_rdd.countByValue();
```

1 行目: sequence ファイルから key をファイル名、value をテキストデータとする PairRDD を生成し、それに対して values() メソッドを用いることでテキストデータをのみを要素とする RDD を生成する

2 行目: テキストを入力として単語に分割したリストを出力する関数を定義したクラス (WordsList) を map() メソッドの引数として text_rdd に適用させ、各テキストの全単語のリストを要素とする RDD を生成する。この RDD は 3 行目の他に STEP4 でも利用するため、persist() を用いてキャッシュしている。ここで生成した words_rdd はデータ量が大きいため、ストレージレベルにディスクを選択した場合が一番高速に動作した。

3 行目: 単語のリストを入力として重複を除いた単語のリストを出力する関数を定義したクラス (UniqueWordsList) を flatMap() メソッドに与え、各テキスト内で重複しない単語のリストを連結した RDD を生成する。

4 行目: 3 行目で生成した RDD に countByValue() メソッドを適用し、key を単語、value を DF とする Map（DF マップ）を生成する。

4.2 STEP2:単語辞書生成

STEP2 では、STEP1 で生成した Map（DF マップ）に通し番号を追加する処理を行う。この処理は逐次的な処理であり、マスターサーバが実行する。

ソースコード 5 STEP2

```
1 TreeMap<String,Pair> word_num_df = new TreeMap<String,Pair>();
2 int i = 0;
3 for (Map.Entry<String, Long> e : word_df.entrySet()){
4         word_num_df.put(e.getKey(), new Pair(e.getValue().intValue(), i ));
5         i++;
6 }
```

1 行目: key に単語、value に DF と通し番号のペアとする新しい Map（単語辞書: この Map を出力する際（人が直接見て分析する際）、単語が辞書順で並んでいたほうが便利であるため、TreeMap を用いた）を生成する。Pair は自作クラスであり、DF と通し番号のペアを格納する。

2-6 行目: for 文で STEP1 で生成した Map（DF マップ）から一つずつ要素を取り出し、通し番号を付け加えたものを 1 行目で生成した Map（単語辞書）に格納している。変数 i が通し番号を表す。

ここで付与した通し番号が、特徴ベクトルを生成する際の各単語に対応する次元となるため、MLlib を用いた手法のように異なる単語が同じ次元に対応することはない。

4.3 STEP3:単語辞書の分配

ソースコード 6 STEP3

```
final Broadcast<TreeMap<String,Pair>> BC_word_dict = sc.broadcast(word_num_df);
final Broadcast<Integer> BC_all_document = sc.broadcast((int)words_list.count());
final Broadcast<Integer> BC_unique_words = sc.broadcast(word_num_df.size());
```

上記のようにブロードキャスト変数を生成することで、引数に与えた変数のデータが各スレーブサーバにコピーされ、その後の並列分散処理の中でこれらのデータを参照できる。BC_word_dict は STEP2 で生成した単語辞書、BC_all_document は全テキスト数、BC_unique_words は全テキスト内で出現した単語の種類数を表す。

4.4　STEP4:TF マップ生成

残りの STEP で、STEP1 の 2 行目で生成した各テキストの全単語のリストを要素とした RDD に、map() メソッドを適用させ、TF-IDF ベクトルを要素とする RDD を生成する。以下、この map() メソッドに渡す関数について述べる。この関数は、単語のリストを入力とし、TF-IDF ベクトル (SparseVector) を出力とする。

ソースコード 7〜8 は各スレーブサーバ単体で実行される逐次的な Java プログラムである。

ソースコード 7　STEP4

```
1  public SparseVector call(List<String> wordList){
2          TreeMap<String,Pair> dict = BC_word_dict.value();
3          int document = BC_all_document.value();
4          int unique_words = BC_unique_words.value();
5          HashMap<String,Integer> tf = new HashMap<String,Integer>();
6          int count;
7          for (String word :wordList){
8                  count = tf.containsKey(word) ? tf.get(word) : 0;
9                  tf.put(word, count + 1);
10         }
11 //ソースコード 8に続く
```

2-4 行目: STEP3 で定義したブロードキャスト変数に対して、データを取り出すために、value() メソッドを用いる（図 2 で示した RDD に対する変換メソッド values() とは異なる）。

5 行目: key を単語、value を TF とする Map（TF マップ）を生成する。この Map は STEP5 で TF-IDF を計算するときに使用するが、順序を考慮する必要はないため、より高速にアクセス出来る HashMap を用いる。

6-10 行目: 5 行目で生成した Map（TF マップ）に値を格納していく。入力されたリスト (wordList) から 1 単語ずつ読み込み、Map（TF マップ）のその単語に対応する value を 1 ずつ加算することで TF を計算している。Map は直接 value の値を増やすことは出来ないので、変数 count に現在の TF 値を格納し、それに 1 を加算した値を put() メソッドによって上書きしている。

4.5 STEP5:TF-IDF ベクトル生成

STEP5 では、STEP3 で各スレーブサーバにコピーした Map（単語辞書）と、STEP4 で生成した Map（TF マップ）を用いて、最終的な TF-IDF ベクトルを計算する。

ソースコード 8 STEP5

```
1  //ソースコード 7の続き
2          int [] indices = new int[tf.size ()];
3          double[] values = new double[tf.size ()];
4          int i = 0;
5          Pair pair = new Pair();
6          for (Map.Entry<String, Integer> e : tf.entrySet()){
7                  pair.set(dict.get(e.getKey()));
8                  indices[i] = pair.second;
9                  values[i] = e.getValue() * (Math.log((double)document / pair.first));
10                 i++;
11         }
12         SparseVector sv = new SparseVector(unique_words, indices, values);
13         return sv;
14 }
```

2-3 行目: 12 行目で SparseVector 型の TF-IDF ベクトルを生成するが、そのコンストラクタの引数として、ベクトルの次元の大きさ (int)、値が 0 でない次元の配列 (int[])、0 でない値の配列 (double[]) をとる。ここでは、コンストラクタに与える 2 つの配列を宣言している。

以下のように、0 でない次元の配列は 0 でない値の配列と対応していれば、昇順である必要はない。

（SparseVector 生成例）

indices = [0, 5, 9, 6, 3]
values = [1, 2, 3, 4, 5]
SparseVector(10, indices, values) ⇒ [1.0, 0.0, 0.0, 5.0, 0.0, 2.0, 4.0, 0.0, 0.0, 3.0]

4-11 行目: Map（TF マップ）から一つずつ単語を取り出し、indices には次元として単語の通し番号を、values には TF-IDF として 3.2 節で紹介した式に従って計算した値を代入している。pair.first は DF を、pair.second は通し番号を表している。

12-13 行目: TF-IDF ベクトルを SparseVector 型として生成し、関数の出力としている。

ここで生成された TF-IDF ベクトルは MLlib で生成されたものと同じ形式であるが、単語辞書を参照することで、どの単語がどの次元に対応しているのかが分かるため、分析の幅が広がる。

5　スケーラビリティの比較

本章では、MLlib を用いた手法と前章で提案した手法のスケーラビリティの比較を行う。結果を示す前に、今回用いたテキストデータと単語の抽出方法を説明する。

5.1　用いたテキストデータ

- エンロンコーパス
  - テキスト数：517,392
  - 言語：英語
  - 総データ量：1.35GB

今回はテキストデータとしてエンロンコーパス [8] を用いた。エンロンコーパスは Web 上に無償で公開されているテキストファイル群であり、テキストマイニングの研究等で利用されている。エンロンコーパスは米エンロン社が実際にやりとりしたメールを基に作成されている。

5.2　単語抽出方法

単語抽出の一般的な流れは以下の通りである。

(i)　文章を単語に分割
(ii)　ストップワード除去
(iii)　単語変換

ストップワードとは、前置詞や接続詞等、内容に関わらず多くのテキストで出現する単語のことであり、これらを除去することでクラスタリングの精度を上げることができる。単語変換では、語形が変化している単語を原形に変換するステミング処理や、大文字を小文字に統一させる処理等を行う。

今回 (i) の処理に関しては、テキストが英語であるため空白文字等によって分割した。(ii)(iii) に関しては、行いたい分析によって変更する必要がある。本研究では単語を抽出することのみを対象としているため、ストップワードの除去は行っていない。ただし、エンロンコーパスに対して (i) の処理を行うと、単語として意味のなさない文字列が大量に含まれてしまうことが分かったため、今回は数字を含む単語と 20 文字以上の単語は抽出しないようにした。単語変換に関しては、大文字を小文字に変換する処理のみ行った。

5.3　評価方法

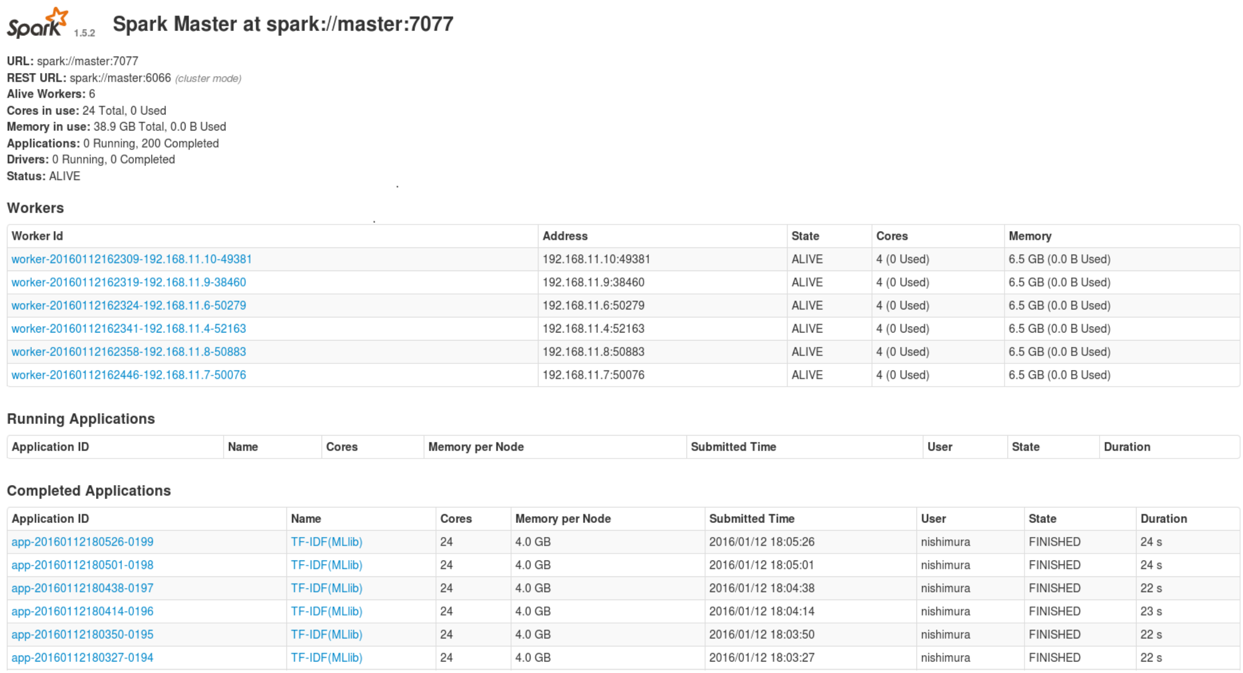
MLlib を用いたアプリケーションと提案手法を用いたアプリケーションを、スレーブサーバ 1〜6 台に対してそれぞれ 100 回ずつ実行し、平均実行時間を比較する。

アプリケーションの仕様はどちらも以下のようにした。

(i) key をファイル名、value をテキストデータとする sequence ファイルを HDFS に格納しておく。
(ii) アプリケーション実行開始
(iii) HDFS に格納された sequence ファイルからテキスト読み込み
(iv) TF-IDF ベクトル (SparseVector) を要素とする RDD 生成し、メモリにキャッシュ
(v) アプリケーション実行終了

実際のテキストマイニングでは、生成された TF-IDF ベクトルを基にさまざまな分析を行っていくため、HDFS から sequence ファイルを読み込んでから、最終結果の RDD（TF-IDF ベクトル）を生成しメモリにキャッシュするまでの時間を測定した（TF-IDF ベクトルを HDFS に格納する必要はない）。

アプリケーションの実行時間はマスターサーバのブラウザからアクセス出来る Spark の WebUI（図 6）で見ることが出来る。さらに図 6 の Application ID からアプリケーションの詳細（図 7）を見ることができ、各アクションの実行時間を見ることが出来る。図 7 の Completed Jobs は実行されたアクションを示しており、さらに各アクションの詳細（図 8）を見ることが出来る。図 8 では各サーバの各 CPU についてタスクの実行状況を詳細に見ることができる。



Spark 1.5.2 **Spark Master at spark://master:7077**

URL: spark://master:7077
REST URL: spark://master:6066 *(cluster mode)*
Alive Workers: 6
Cores in use: 24 Total, 0 Used
Memory in use: 38.9 GB Total, 0.0 B Used
Applications: 0 Running, 200 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160112162309-192.168.11.10-49381	192.168.11.10:49381	ALIVE	4 (0 Used)	6.5 GB (0.0 B Used)
worker-20160112162319-192.168.11.9-38460	192.168.11.9:38460	ALIVE	4 (0 Used)	6.5 GB (0.0 B Used)
worker-20160112162324-192.168.11.6-50279	192.168.11.6:50279	ALIVE	4 (0 Used)	6.5 GB (0.0 B Used)
worker-20160112162341-192.168.11.4-52163	192.168.11.4:52163	ALIVE	4 (0 Used)	6.5 GB (0.0 B Used)
worker-20160112162358-192.168.11.8-50883	192.168.11.8:50883	ALIVE	4 (0 Used)	6.5 GB (0.0 B Used)
worker-20160112162446-192.168.11.7-50076	192.168.11.7:50076	ALIVE	4 (0 Used)	6.5 GB (0.0 B Used)

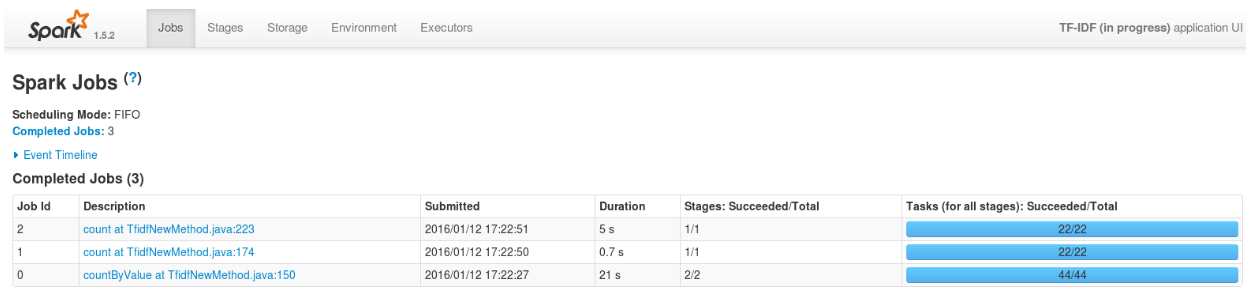
Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160112180526-0199	TF-IDF(MLlib)	24	4.0 GB	2016/01/12 18:05:26	nishimura	FINISHED	24 s
app-20160112180501-0198	TF-IDF(MLlib)	24	4.0 GB	2016/01/12 18:05:01	nishimura	FINISHED	24 s
app-20160112180438-0197	TF-IDF(MLlib)	24	4.0 GB	2016/01/12 18:04:38	nishimura	FINISHED	22 s
app-20160112180414-0196	TF-IDF(MLlib)	24	4.0 GB	2016/01/12 18:04:14	nishimura	FINISHED	23 s
app-20160112180350-0195	TF-IDF(MLlib)	24	4.0 GB	2016/01/12 18:03:50	nishimura	FINISHED	22 s
app-20160112180327-0194	TF-IDF(MLlib)	24	4.0 GB	2016/01/12 18:03:27	nishimura	FINISHED	22 s

図 6　Spark の WebUI



Spark 1.5.2 Jobs Stages Storage Environment Executors **TF-IDF (in progress)** application UI

Spark Jobs (?)

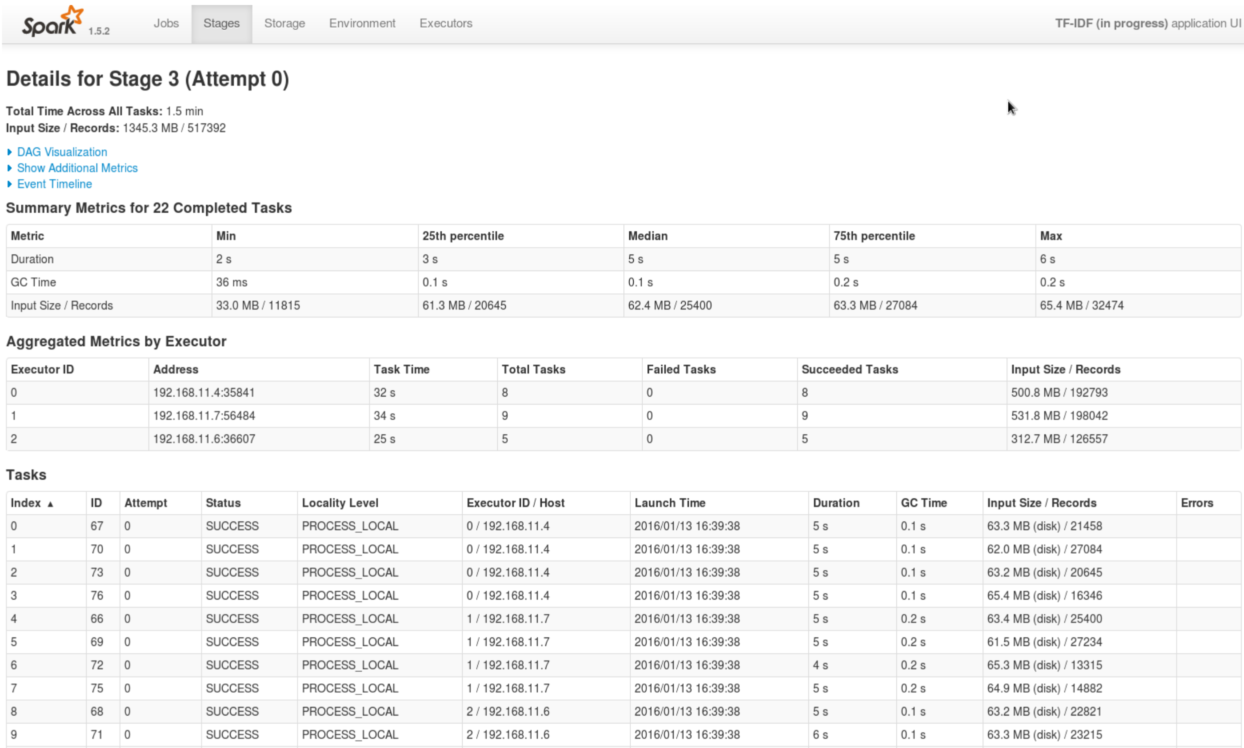
Scheduling Mode: FIFO
Completed Jobs: 3

▸ Event Timeline

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at TfidfNewMethod.java:223	2016/01/12 17:22:51	5 s	1/1	22/22
1	count at TfidfNewMethod.java:174	2016/01/12 17:22:50	0.7 s	1/1	22/22
0	countByValue at TfidfNewMethod.java:150	2016/01/12 17:22:27	21 s	2/2	44/44

図 7　アプリケーションの詳細



Spark 1.5.2 | Jobs | Stages | Storage | Environment | Executors | **TF-IDF (in progress)** application UI

Details for Stage 3 (Attempt 0)

Total Time Across All Tasks: 1.5 min
Input Size / Records: 1345.3 MB / 517392

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 22 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2 s	3 s	5 s	5 s	6 s
GC Time	36 ms	0.1 s	0.1 s	0.2 s	0.2 s
Input Size / Records	33.0 MB / 11815	61.3 MB / 20645	62.4 MB / 25400	63.3 MB / 27084	65.4 MB / 32474

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records
0	192.168.11.4:35841	32 s	8	0	8	500.8 MB / 192793
1	192.168.11.7:56484	34 s	9	0	9	531.8 MB / 198042
2	192.168.11.6:36607	25 s	5	0	5	312.7 MB / 126557

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	67	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.11.4	2016/01/13 16:39:38	5 s	0.1 s	63.3 MB (disk) / 21458	
1	70	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.11.4	2016/01/13 16:39:38	5 s	0.1 s	62.0 MB (disk) / 27084	
2	73	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.11.4	2016/01/13 16:39:38	5 s	0.1 s	63.2 MB (disk) / 20645	
3	76	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.11.4	2016/01/13 16:39:38	5 s	0.1 s	65.4 MB (disk) / 16346	
4	66	0	SUCCESS	PROCESS_LOCAL	1 / 192.168.11.7	2016/01/13 16:39:38	5 s	0.2 s	63.4 MB (disk) / 25400	
5	69	0	SUCCESS	PROCESS_LOCAL	1 / 192.168.11.7	2016/01/13 16:39:38	5 s	0.2 s	61.5 MB (disk) / 27234	
6	72	0	SUCCESS	PROCESS_LOCAL	1 / 192.168.11.7	2016/01/13 16:39:38	4 s	0.2 s	65.3 MB (disk) / 13315	
7	75	0	SUCCESS	PROCESS_LOCAL	1 / 192.168.11.7	2016/01/13 16:39:38	5 s	0.2 s	64.9 MB (disk) / 14882	
8	68	0	SUCCESS	PROCESS_LOCAL	2 / 192.168.11.6	2016/01/13 16:39:38	5 s	0.1 s	63.2 MB (disk) / 22821	
9	71	0	SUCCESS	PROCESS_LOCAL	2 / 192.168.11.6	2016/01/13 16:39:38	6 s	0.1 s	63.3 MB (disk) / 23215	

図 8　アクションの詳細

5.4　実行環境

5.4.1　ハードウェア

- マスターサーバ（1 台）
  - Lenovo ThinkServer TS140
  - CPU : Intel Xeon Processor E3-1226 v3 （4 コア）
  - メモリ : 16GB
- スレーブサーバ（6 台）
  - Lenovo ThinkServer TS140
  - CPU : Intel Xeon Processor E3-1226 v3（4 コア）
  - メモリ : 8GB
- ルーター
  - WSR-300HP
  - 最大転送速度 : 1000Mbps
- ハブ
  - LSW5-GT-8NS
  - 最大転送速度 : 1000Mbps

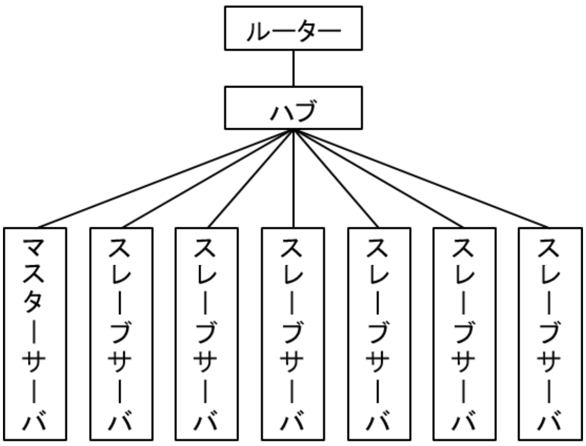


図 9　クラスタ構成図

5.4.2 ソフトウェア

- Java
  - version:1.8.0_65
- Spark
  - version:1.5.2

Spark や HDFS にはさまざなパラメータが用意されており、実行環境や処理内容に合わせて変更することができる。今回用いた、アプリケーションの実行速度に関係するパラメータ設定について以下にまとめた。

Sparkのパラメータ	設定値	説明
spark.driver.memory	8GB	ドライバプログラムが使用できるメモリの量
spark.executor.memory	4GB	各スレーブサーバにおいてSparkの処理に使用できるメモリの量
spark.serializer	org.apache.spark.serializer.KryoSerializer	オブジェクトの*シリアライズ処理に使われるクラス

*シリアライズ:ソフトウェアで扱っているデータを丸ごと、ファイルで保存したりネットワークで送受信することができるように変換すること

図 10　Spark のパラメータ

HDFSのパラメータ	設定値	説明
dfs.blocksize	64MB	分割するブロックのサイズ（2.3節）
dfs.replication	1	レプリケーション数（2.3節）

図 11　HDFS のパラメータ

5.5　結果・考察

結果を図 12 に示す。横軸はスレーブサーバの数を表し、縦軸は 100 回実行した際の平均実行時間を表している。実行時間については、スレーブサーバの台数に関わらず、提案手法は MLlib を用いた手法には及ばなかった。原因としては、MLlib では生成していない単語辞書を生成していることが考えられる。しかしスケーラビリティについては、MLlib を用いた手法では、台数を増やすことで実行時間が増加してしまっている箇所があるが、提案手法では台数を増やすほど実行時間が短縮出来ている。

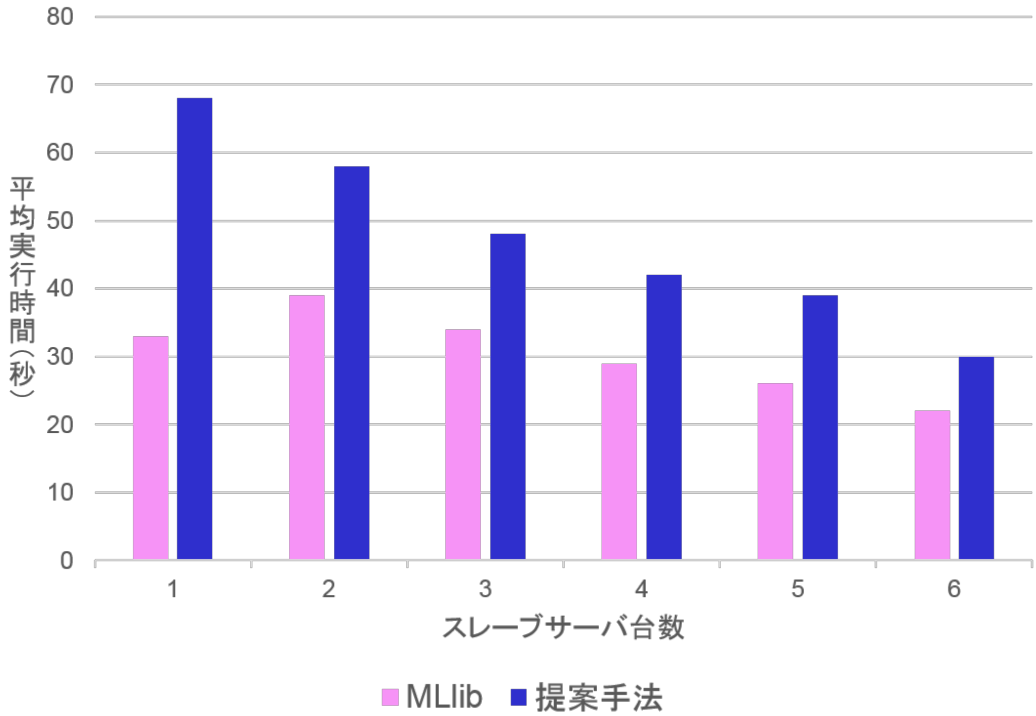


図 12　スケーラビリティ

並列分散処理フレームワークを用いた処理では、サーバ間の通信によるオーバーヘッドが、サーバ数を増やしても処理速度が向上しない主な原因となる。MLlib を用いた手法は、TF ベクトルを計算してからそれを用いて IDF を計算するが、その際にサーバ間の通信によるオーバーヘッドが生じる。一方提案手法は、サーバ間の通信は STEP1 の DF をカウントする処理と STEP3 でデータをコピーする処理で生じているが、MLlib を用いた手法よりもオーバーヘッドが小さく、並列化に向いているアルゴリズムであると考えられる。

次に、各アプリケーションのアクションごとの実行時間を調べてみる。MLlib を用いた手法のアクションは、IDF を求めるまで（ソースコード 3 の 1〜5 行目）の処理と、IDF を TF ベクトルに掛け合わせる処理の 2 つに分けられているが、ほとんどの時間が前者のアクションに費やされ、後

者のアクションはスレーブサーバの台数に関わらず 1 秒以下となっている。一方提案手法のアクションは、大きく分けて DF カウントまでの処理 (STEP1) と、TF-IDF を計算する処理 (STEP4～5) の 2 つに分けられており、これら 2 つのアクションがアプリケーションのほとんどの実行時間を占めている。2 つのアクションの実行時間の比較を図 13 に示す。図 13 から、DF カウントの方が TF-IDF ベクトル計算に比べて台数を増やした時の実行時間の短縮の割合が少なくなっていることが分かる。これは、DF カウントはサーバ間通信のためにオーバーヘッドが生じるためであり、TF-IDF ベクトル計算はサーバ間通信を必要としない独立した並列分散処理が行われているためである。

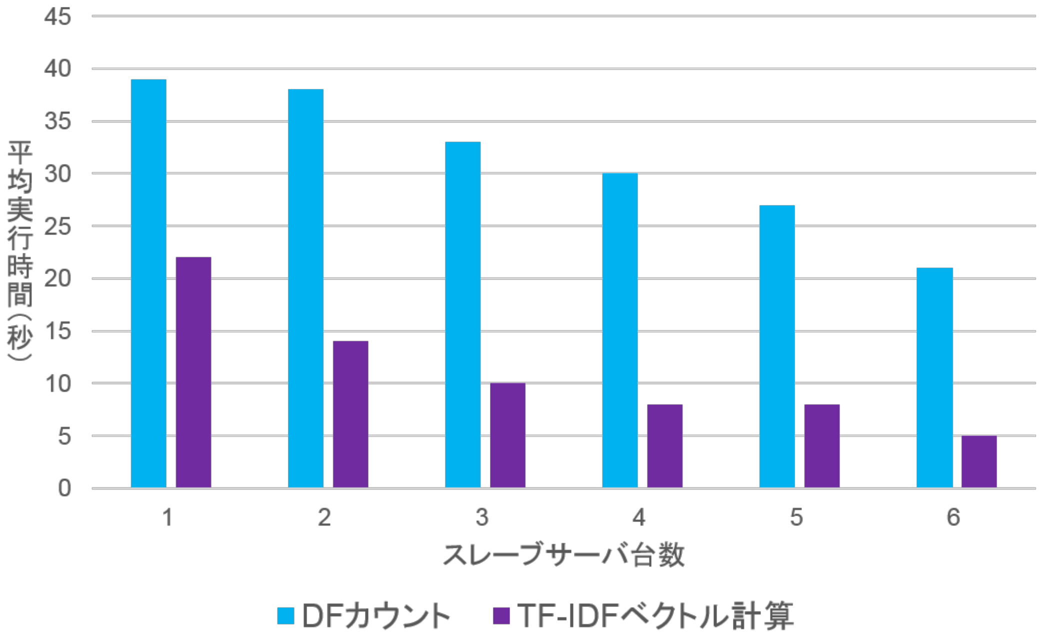


図 13　提案手法のアクション別スケーラビリティ比較

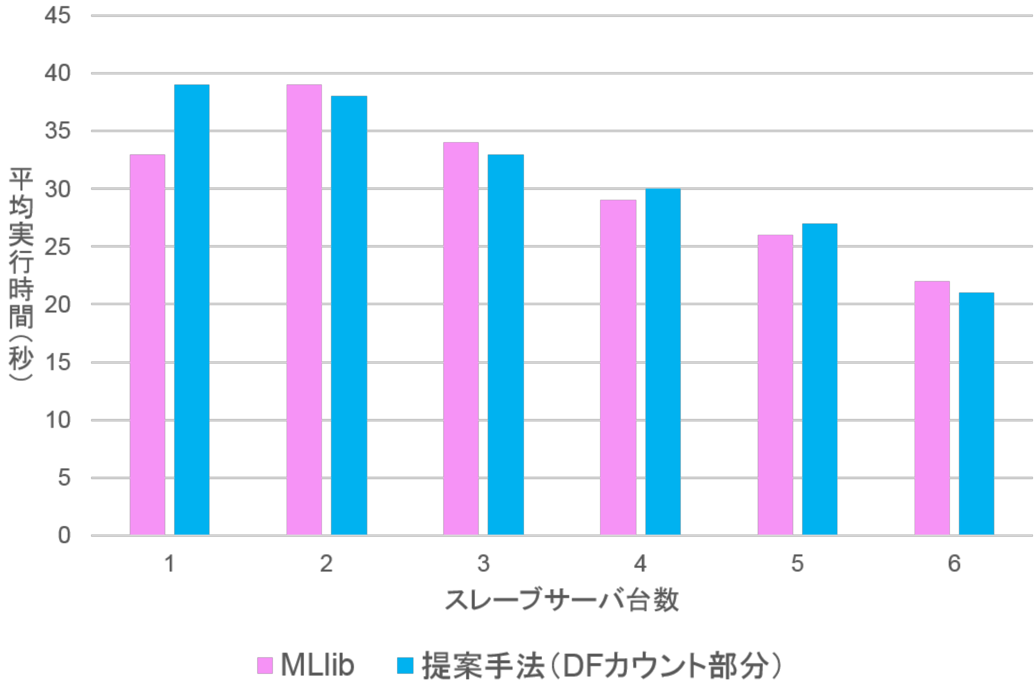


図 14　MLlib と提案手法の DF カウント部分のスケーラビリティ比較

また、MLlib を用いた手法の実行時間と、提案手法の DF カウント (STEP1) の実行時間の比較を図 14 に示す。これら 2 つの処理は、各スレーブサーバ台数において実行時間が近似していることが分かる。スレーブサーバの台数をさらに増やした場合も実行時間が近似すると仮定すれば、スレーブサーバの台数を増やすことで、提案手法の TF-IDF ベクトル計算部分は、完全に独立した並列分散処理のため実行時間がさらに短縮され、提案手法の全体の実行時間が MLlib を用いた手法の実行時間により近づくことが期待される。

6 まとめ

6.1 結論

本論文では、Spark を用いた TF-IDF の計算について、既存のライブラリ MLlib を用いた手法の短所を改善し、かつ実行時間の増加を最小限に留める手法を提案した。提案手法は TF-IDF ベクトルだけでなく単語辞書という副産物も生成するため、MLlib を用いた手法では出来なかった TF-IDF ベクトルからの重要単語抽出等を可能にし、また単語辞書を用いた分析（DF を用いたテキスト群の包括的な分析等）も可能にした。さらに、提案手法は各単語に通し番号を与えることで、MLlib を用いた手法の短所であった異なる単語が同じ次元に対応してしまう危険性を回避でき、より正確な分析が可能となった。また、提案手法の実行時間は MLlib を用いた手法には及ばなかったが、より高いスケーラビリティを持つことが分かった。今回は予算の関係上スレーブサーバ 6 台までしか実行できなかったが、さらに台数を増やすことで MLlib の速度に追いつく可能性も考えられる。しかし、実際に検証するにはさらに巨大な Spark クラスタを用いなければならず、今回は検証出来なかった。

6.2 展望

Spark を用いると簡単に並列分散処理が実装できるが、今回の研究で、同じ処理でも並列化のアルゴリズムの違いによって実行時間に差が出ることが分かった。今回は TF-IDF の計算に特化したが、テキストマイニングにおいて TF-IDF の計算はその後の分析のための下準備に過ぎない。TF-IDF ベクトルは主にクラスタリングに用いられるが、クラスタリングにはさまざまな方法があり、それらのより高速な並列化アルゴリズムを探っていくことが今後の課題である。

7　謝辞

今回の研究を通じて、さまざまなテキストマイニング手法や Hadoop・Spark といった並列分散技術を学ぶことができ、非常に貴重な経験を積むことが出来ました。このような研究に取り組む機会を与えてくださった指導教員福永力教授、株式会社 UBIC 武田秀樹様、蓮子和己様、藤田肇様、小野里拓一様、猪瀬悟史様に深く感謝致します。また、今回の研究に関して多くの助言をくださった工藤健史氏と日高敬介氏に深く感謝致します。

参考文献

[1] Hadoop homepage
http://hadoop.apache.org/（参照 2016/1/27）

[2] Tom White 訳　玉川竜司　兼田聖士　「Hadoop 第 3 版」　オライリー・ジャパン

[3] Spark homepage
http://spark.apache.org/（参照 2016/1/27）

[4] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, 訳　玉川竜司
「初めての Spark」　オライリージャパン

[5] 金明哲　「テキストデータの統計科学入門」　岩波書店　2009

[6] Mesos homepage
http://mesos.apache.org/（参照 2016/1/27)

[7] 井上誠一郎　永井雅人　「パーフェクト Java　第 2 版」　技術評論社　2014

[8] エンロンコーパス
https://www.cs.cmu.edu/ ./enron/（参照 2016/1/27)