RogueWave SOFTWARE

Accelerating Great Code

Hadoop MapReduce アプリケーションでの JMSLライブラリの使用

「ビッグデータ」は、この 10 年以上の間に業界の流行語になりました。その意味するものは人によって異なるとはいえ、確かにデータサイズは爆発的に増え、データの複雑さも増大してきました。全世界のデジタルデータは 2020 年までに 44 ゼタバイトになるとの予測もあります［CIO Insight, 2014 年］[1]。インターネット、スマートフォン、IoT（モノのインターネット）のためのデバイスに加え、安価な市販品として入手可能になったハードウェアの普及によって、より多くの、より新しいタイプのデータが収集、処理、保存され、そして分析されています。

今日のデータ規模は前例がありませんが、このコンピュータ処理上の大きな問題に対する研究はしばらく前から行われています。その標準的な戦略は、まず問題を管理可能なサイズの部分に分割してから、各々独立したワーカーに送り、並列に処理するというものです
各ワーカーは、結果を中央集中型のコントローラーに伝え、コントローラーで情報を作業可能な形に結合します。分散コンピューティングとは、問題を部分に切り分け、ネットワーク上でそれらの各部分が連携することに対して一般的に使用される用語です。一方、並列計算とは、より規模の大きい問題のパーツを同時に処理することを意味します[2]。

MapReduce は、そのような戦略に対応したプログラミングモデルです。MapReduce は、大規模データを処理する際の課題に対応するために、Google の科学者によって 2004 年[3] に開発されました。MapReduce プログラムの ２つの基本ステップは、map と reduce です。map ステップは入力データを読み込み、reduce ステップ用にデータを照合し、reduce ステップでは有意義な方法でデータを組み合わせます。MapReduce モデルで最も広く使用されている実装の 1 つが Apache Hadoop です。
Apache Hadoop は、分散コンピューティング用の Java オープンソースプロジェクトです。Hadoop には、HDFS（Hadoop 分散ファイルシステム）と MapReduce プログラミングおよびジョブ管理フレームワークという ２つの主要コンポーネントがあります。

JMSL数値計算ライブラリは数値演算のための100% Java ライブラリであり、広範囲にわたる先進的な数学/統計計算、チャート表示を Java 環境で実現します。コアの Java Numeric が拡張され、開発者は Java アプリケーションに先進的な分析をシームレスに統合できます。

分散コンピューティング環境において、JMSL はクラスタの各ノードで利用可能であり、データのセクションを処理し、論理的な方法で収集と結合が可能な形式での出力を生成できます。以降のセクションでは、Hadoop MapReduce アプリケーションで JMSL を使用する方法を説明します。

[1] 1 ゼタバイトは 1,000 エクサバイト = 100 万ペタバイト = 10 億テラバイト。44 ゼタバイトは 440 億テラバイトです！
[2] 並列計算を、同一 CPU 上での複数スレッドの使用と定義する著者もいます。
[3] http:research.google.com/archive/mapreduce-osdi04-slides/index-auto-0006.html

Hadoop MapReduce アプリケーションの構造

Hadoop MapReduce アプリケーションのプロトタイプは少なくとも 1 つの Mapper クラス、1 つの Reducer クラス、および 1 つの driver クラスを持ちます。Mapper クラスは入力データを読み込み、中間データを作成します。Reducer クラスは、中間情報を累積し、結合した結果を書き出します。Driver クラスは Hadoop のConfigured クラスを拡張し、Hadoop の Tool インターフェイスを実装し、ジョブの設定と実行に使用されます。

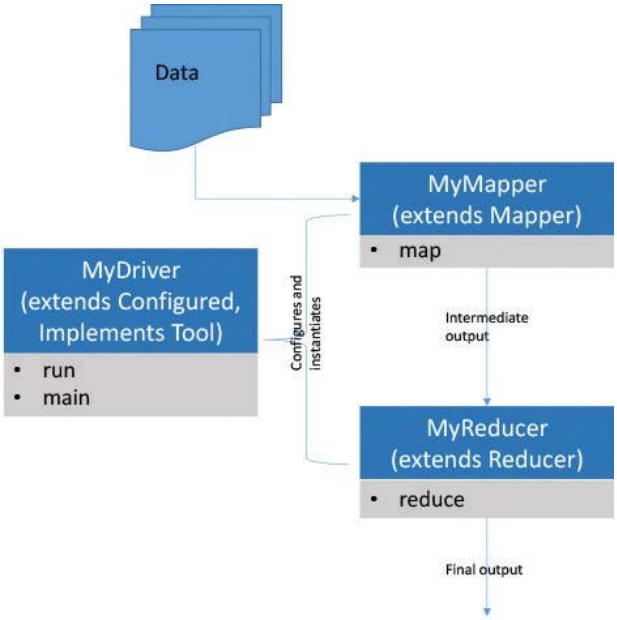

図1:単純な Hadoop アプリケーションの構成要素

プログラマは、データの形式、データの処理や結合方法の指示、望ましい出力形式を取り扱うためにメソッドを作成（上書き）する必要があります。これらはすべて、プログラマが解決しようとしているデータ分析問題の性質によって異なります。最初の例は、ごく単純な問題で始めます。キーごとの数値の集計です。

例 1：JMSL Summary

MapReduce は〈key, Item〉ペアに組織化されたデータ上で動作します。各データ “アイテム” は 1 つだけの “キー” に属すると仮定します。これは、Mapper が期待する入力データの形式です。この最初の例では、〈ABC, 10〉のように、キーはテキストであり、アイテムはスカラー値とします。コンマ区切りの入力ファイルの内容は次のようになります。

```
BDK,  20
ABC,  10
EAF,  50
AGC,  10
IJK,  90
DAL,  40
```

JMSL Summary クラスは、カウント、平均、分散、その他の記述統計などの要約統計情報を生成します。Hadoop クラスタ上でデータをキーごとに集計するために、SummaryDriver、SummaryMapper、および SummaryReducer の ３つのクラスを作成します。

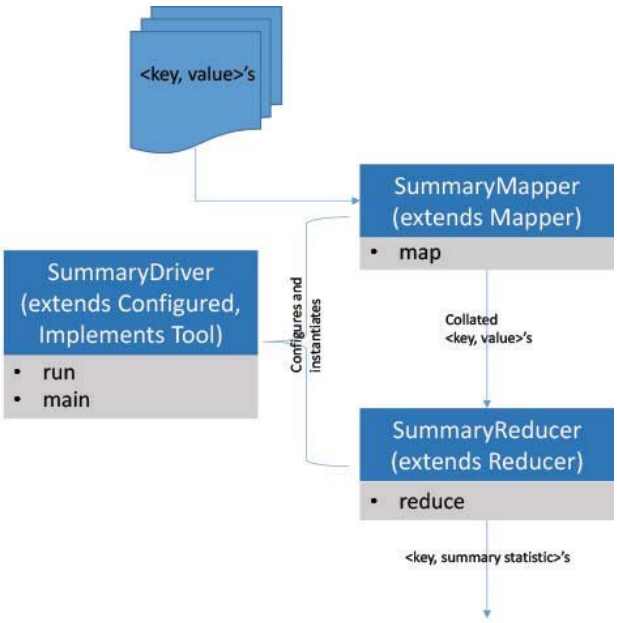

図2:Summary のコンポーネント

SummaryMapper は、Hadoop のMapper クラスを拡張し、map() メソッドを上書きします。map() メソッドは、入力データを読み込み、SummaryReducer に送信するための中間出力を書き出します。この中間出力は、Hadoop で管理され、ローカルのファイルシステムに書き込まれます。

```
public static class SummaryMapper extends
              Mapper<Object, Text, Text, DoubleWritable> {

     Text word = new Text( “key” );
     DoubleWritable val = new DoubleWritable();

     @Override
     public void map(Object key, Text value, Context context)
              throws InterruptedException, IOException {

          String columns[] = value.toString().split( “,” );
          word.set(columns[0]);
          val.set(Double.parseDouble(columns[1]));
          context.write(word, val);
     }
}
```

SummaryReducer は、Hadoop の Reducer クラスを拡張し、reduce() メソッドを上書きします。reduce() メソッドは、一意のキーそれぞれに対して、すべての中間値を結合し、出力をマージします。reduce() メソッド内では、アイテムの結合方法を定義するために JMSL Summary クラスを使用します。

```
public static class SummaryReducer extends
          Reducer<Text, DoubleWritable, Text, DoubleWritable> {

     Text out = new Text();
     DoubleWritable result = new DoubleWritable();

     @Override
     public void reduce(Text key, Iterable<DoubleWritable> values,
               Context context) throws IOException, InterruptedException {

          // JMSL
          Summary summary = new Summary();
          // For manual calculation.
          double sum = 0;
          double count = 0;

          for (DoubleWritable val : values) {
               // JMSL
               summary.update(val.get());
               // Manual calculation, just for validation.
               sum += val.get();
               count += 1;
          }
          sum = sum / count;
          out.set(key);
          result.set(summary.getMean());
          System.out.println(key + “,” + sum);

          context.write(out, result);
     }
}
```

大規模クラスタでは、特定のキーに対するデータが複数のノードに存在する場合があります。map タスクは、reduce() メソッドを呼び出す前に、特定のキーのすべてのデータを一緒に取り出すように処理します。reduce() メソッド内の summary. update() は一度に 1 つの値を受け取るため、追加のデータ配列を作成する必要はありません。

もちろん、平均値の計算は簡単です。手動で計算することが可能ですし、結果の確認のために、Hadoop でキーごとに結果を追跡し続けることもできます。しかし真に有益なのは、Hadoop が通信オーバーヘッドを管理することを踏まえつつ、分散ネットワーク上のより高度な解析に対して JMSL メソッドを利用する場合です。

次に、SummaryDriver の run() メソッドがMapReduce ジョブの設定と構成を扱います。まず、以下の行で新しいジョブをインスタンス化します。

```
Job job = new Job(super.getConf(), “calculate mean”);
```

以降の行で、Mapper と Reducer のための入出力値と入出力パスを設定します。

```
    @Override
    public int run(String[] args) throws IOException, InterruptedException, ClassNotFound
            Exception {

        // Set up the configuration for the MapReduce job.
        Job job = new Job(super.getConf(), “calculate mean”);
        job.setJarByClass(getClass());
        job.setMapperClass(SummaryMapper.class);
        job.setReducerClass(SummaryReducer.class);

        // Set the input and output data types.
        // Note that Mapper output key and value must match
        // reducer input key and value.
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(DoubleWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(DoubleWritable.class);

        // Set the input path and output path.
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ?0 :1);
        return 0;
    }
```

ジョブを実行する文は次のとおりです。

```
      System.exit(job.waitForCompletion(true) ?0 :1);
```

SummaryDriver の main() メソッドがrun() メソッドを呼び出します。

```
public static void main(String[] args) throws IOException,
            InterruptedException, ClassNotFoundException,
            Exception {

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
                    args).getRemainingArgs();

        if (otherArgs.length != 2) {
            System.err.println( “Usage:SummaryDriver <in> <out>” );
            System.exit(2);
        }

    int res = ToolRunner.run(conf, new SummaryDriver(), args);
}
```

クラスタ上では、コマンドラインからコードを実行するための形式は次の通りです。

```
$hadoop jar <nameOfJarFile>.jar <class_directory/nameOfDriverClass> /
<Hadoop options> <args to driver class>
```

JMSL をクラスタのノードで利用可能にするには、Hadoop の libjars オプションを使用します。たとえば、最初の例のコマンドは、次のようになります。

```
$hadoop jar HadoopJMSLExamples.jar <class_directory>/SummaryDriver /
-libjars /lib/jmsl.jar /user/hdfs/SummaryInput /user/hdfs/SummaryOutput
```

クラスタ上でジョブを実行する際のその他の考慮事項については付録を参照してください。

例 2：JMSL 線形回帰

多重線形回帰は、最も広く使用される予測モデルの 1 つです。このモデルでは、1つの変数、つまり従属変数または応答変数の変動が、独立変数、つまり説明変数または予測変数のセットでの変動によって説明可能であり、係数同士の関係が線形であると仮定します[4]。

製品を販売する企業が新店舗の場所を計画する場合を例に考えると、売り上げを従属変数とし、人口や1人あたりの国民所得、その他の人口統計学的変数などを予測変数として回帰モデルを設定します。
数多くの製品や顧客を扱い、1 日のトランザクション量が数百万になるような大企業では、Hadoop MapReduce などの分散ソリューションを使って、データを適切な詳細レベルで蓄積して回帰モデルに適合させる必要があるでしょう。この例では、新店舗でのビジネス計画で使用されるようなケースを、Hadoop アプリケーションで JMSL 線形回帰を使用して示します。

観測値が回帰モデルで利用可能となるためには、通常、入力データを組織化するための多くのステップが必要です。これらのステップがすでに行われたとして（Hadoop MapReduce はこの目的にも使用できます）、回帰モデルのための観測値は次の形式をとります。

$$
\begin{aligned}
&Y_1,\ x_{11},\ x_{12},\ x_{13},\ \cdots,\ x_{1m}\\
&Y_2,\ x_{21},\ x_{22},\ x_{23},\ \cdots,\ x_{2m}\\
&Y_3,\ x_{31},\ x_{32},\ x_{33},\ \cdots,\ x_{3m}\\
&\cdots\\
&Y_n,\ x_{n1},\ x_{n2},\ x_{n3},\ \cdots,\ x_{nm}
\end{aligned}
$$

つまり、従属変数 Y は、予測変数の観測値とペアになります。線形回帰アルゴリズムは、各インデックス i に対して、完全な（m+1）個の組（Y_i, x_{i1}, xi_2, x_{i3}, …, x_{im}）を知っている必要があります。

例 1 では、プロトタイプの MapReduce ジョブ のデータ形式である <key, Item> ペア上で直接動作しましたが、この例の Mapper クラスがプロセス <key, (Item1,…, ItemM)> タイプのデータを読み込むためには工夫が必要です。ここで、キーの値が異なれば回帰データセットも異なります。 1 つのやり方として、行全体を単一のテキストオブジェクトとして取り扱ってから、要素をパースして doubleValuesに割り当てることができます。わかりやすくするために、ここでは応答変数と予測変数を 1 セットとしてグループ化します。これによって観測値は Hadoop によってひとかたまりのデータとして処理されます。

[4] モデルの技術的な前提条件は、診断テストを使用して特定の問題に対して考慮され検証される必要があります。詳しくは、たとえばNeter, et al.を参照してください。

まず、新しい型 DoubleArrayWritable を作成します。これは、2 つの Hadoop データ型 DoubleWritable および ArrayWritable を拡張して作成されます。

```
public class DoubleArrayWritable extends ArrayWritable {

     public DoubleArrayWritable() {
          super(DoubleWritable.class);
     }

     public DoubleArrayWritable(DoubleWritable[] values) {
          super(DoubleWritable.class, values);
     }

     public void set(DoubleWritable[] values) {
          super.set(values);
     }

     public DoubleWritable get(int idx) {
          return (DoubleWritable) get()[idx];
     }

     public double[] getArray(int from, int to) {
          int sz = to - from + 1;
          double[] vector = new double[sz];
          for (int i = from; i <= to; i++) {
               vector[i - from] = get(i).get();
          }
          return vector;
     }
}
```

新しいクラスに加えて、今回も Driver、Mapper、Reducer が必要です。

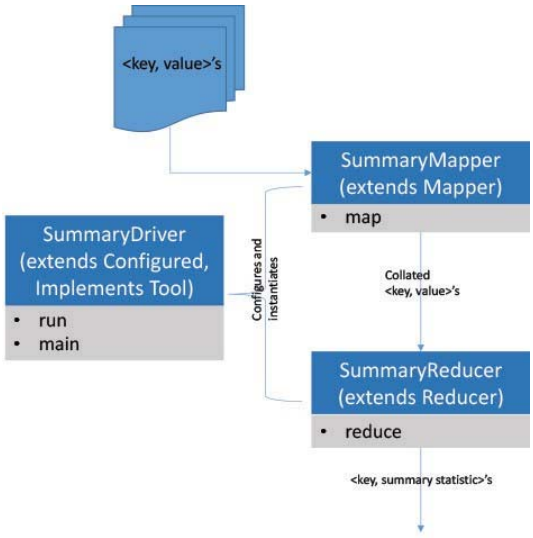

図3:線形回帰の構成要素

Mapper クラスは新しい入力を処理し、DoubleArrayWritable オブジェクトを作成します。map() メソッドは、context.write(word, val)行でReducer 用のキーと配列を書き出します。

```
public static class LinearRegressionMapper extends
          Mapper<Object, Text, Text, DoubleArrayWritable> {

     @Override
     public void map(Object key, Text value, Context context)
               throws InterruptedException, IOException {

          String origColumns[] = value.toString().split( “,” );
          Text word = new Text( “key” );
          word.set(origColumns[0]);
          DoubleWritable[] x = new DoubleWritable[origColumns.length - 1];

          for (int i = 0; i < x.length; i++) {
              x[i] = new DoubleWritable();
              x[i].set(Double.parseDouble(origColumns[i + 1]));
          }
          DoubleArrayWritable val = new DoubleArrayWritable(x);
          // Emit (word, val) for the reducer where word
          // is the key and val is a DoubleArrayWritable.
          context.write(word, val);
     }
}
```

LinearRegressionReducer では、DoubleArrayWritable オブジェクトから観測値を抽出し、応答値を割り当ててから、予測変数を割り当てます(予測変数の数をチェックすることで、キーによってサイズが異なる問題にも対応できます)。観測値から情報を抽出した後、LinearRegression オブジェクト(ここでは lr)に update が適用されます。

```
public static class LinearRegressionReducer extends Reducer<Text, D
          oubleArrayWritable, Text, Text> {

     @Override
     public void reduce(Text key, Iterable<DoubleArrayWritable> values,
               Context context) throws IOException, InterruptedException {

          int numPred = 0;
          double y = 0.0;
          double[] x = null;
          DoubleWritable[] dwa = null;
          LinearRegression lr = null;

          for (DoubleArrayWritable val : values) {
               dwa = (DoubleWritable[]) val.toArray();
               if (numPred == 0) {
                    numPred = dwa.length - 1;
                    //JMSL
                    lr = new LinearRegression(numPred, false);
```

```
                }
                y = dwa[0].get();
                x = new double[numPred];
                for (int i = 0; i < numPred; i++) {
                     x[i] = dwa[i + 1].get();
                }
                //JMSL
                lr.update(x, y);
          }

          // Now we emit the regression model object
              double[] coefs = lr.getCoefficients();
              StringBuilder val = new StringBuilder();
              String out = “Problem “ + key.toString() + “ Coefficients “;
              val.append(“¥n”);

               for (int i = 0; i < coefs.length; i++) {
                    if (i < coefs.length - 1) {
                         val.append(String.format(“%5.4f,”, coefs[i]));
                    } else {
                         val.append(String.format(“%5.4f”, coefs[i]));
                    }
               }
               val.append(String.format(“%n”));
               context.write(new Text(out), new Text(val.toString()));
     }
}
```

出力のフォーマットとコンテンツには複数のオプションがあります。ここでは、推定された係数を抽出し、Text オブジェクトに追加します。なお、Reducer からの出力形式は、最後の２つの引数と一致する必要があります。

```
Reducer<Text, DoubleArrayWritable, Text, Text>
```

LinearRegressionDriver の run() メソッドには、SummaryDriver.run と同様の文が含まれ、前と同じようにジョブ構成を設定してLinearRegressionDriver.main がrun() メソッドを呼び出します。

```
@Override
public int run(String[] args) throws Exception {
     // Set up a new job.
     Job job = new Job(super.getConf(), “Linear Regression”);
     // Sets the Jar file to this specific class.
     job.setJarByClass(getClass());
     // Sets the Mapper and Reducer to LinearRegressionMapper and
     // LinearRegressionReducer.
     job.setMapperClass(LinearRegressionMapper.class);
     job.setReducerClass(LinearRegressionReducer.class);
     // Set the Map output classes and the Reducer Output classes.
     job.setMapOutputKeyClass(Text.class);
     job.setMapOutputValueClass(DoubleArrayWritable.class);
     job.setOutputKeyClass(Text.class);
```

```
        job.setOutputValueClass(Text.class);
        //Set the input and output paths.
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ?0 :1);
        return 0;
    }

    public static void main(String[] args) throws IOException,
            InterruptedException, ClassNotFoundException, Exception {

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
                                                    args).getRemainingArgs();

        if (otherArgs.length != 2) {
            System.err.println("Usage:LinearRegressionDriver <in> <out>");
            System.exit(2);
        }

        int res = ToolRunner.run(conf, new LinearRegressionDriver(), args);
    }
```

同様に、クラスタでジョブを実行するコマンドは次のとおりです。

```
$hadoop jar HadoopJMSLExamples.jar / <class_directory>/LinearRegressionDriver -libjars
/lib/jmsl.jar / /user/hdfs/LinearRegressionInput /user/hdfs/LinearRegressionOutput
```

この Hadoop MapReduce アプリケーションは分散データで線形回帰を実行し、大企業が人口統計学上のデータにもとづいた予測売上を使って新しい店舗の場所を計画する際などに役立ちます。入力データが前述した観測値の形式を取っていれば、このコードはこのままでキー別線形回帰モデルに適合します。出力結果は、キーごとの各モデルの係数推定値です。私たちは、売上と人口統計学データを、３つの地域 R1、R2、R3 でシミュレーションし、３つの回帰問題を、従属変数として売上を、独立変数として人口と一人あたりの可処分所得を使用して実行しました。それぞれの問題ごとに評価係数を含む ファイルが出力されます。

```
$ hadoop fs -cat LinearRegressionOutput/part-r-00000

Problem R1 Coefficients
4.5092, 0.0592, 0.0089

Problem R2 Coefficients
2.0779, 0.0640, 0.0095

Problem R3 Coefficients
14.7010, 0.1002, 0.0057
```

地域 1 に対して、適合された回帰直線は次のとおりです。
Y = 4.5092 + 0.0592*X1 + 0.0089*X2

この地域内の新しい市場エリアについて、人口 X1 = 302（1000 人単位）および平均可処分所得 X2 = 4500（1 人あたりドル)の場合、予測売上（1000 ドル単位）は次になります。

Y = 4.5092 + 0.0592*302 + 0.0089*4500 = 62.4376.

同様に、他の ２つの地域に対しても予測売上を取得できます。線形回帰に関するその他のビッグデータアプリケーションの例を数例あげるとすると、公共の調査データ、ソーシャルネットワーク解析、ゲノム相関解析などがあります。

例 3：アソシエーションルール発見のための Apriori

アソシエーションルールの発見は、離散したアイテムの間の相関を発見する問題と言えます。たとえば、マーケットバスケット解析では、離散したアイテムは、個々のトランザクションで一緒に購入されたそれぞれの製品です。多種多様な商品を販売する企業は、プロモーションの改善に役立てるためにマーケットバスケット解析を使用することがあります。たとえば、２つの商品に高い相関があると知っていれば、企業は商品の両方ではなくどちらか一方だけのプロモーションを実行するだけで済みます。テキストマイニングやバイオインフォマティクスの分野で、アソシエーションルール発見のためのアプリケーションが使われています。

Apriori アルゴリズム（Agrawal and Srikant, 1994）は、トランザクションデータセットにおけるアソシエーションルール発見に関する最も有名なアルゴリズムの 1 つです。Apriori ではまず、頻出するアイテムセットに対してトランザクションを調べます。アイテムセットが頻出である、とは最小トランザクション数よりも多く表れるということを指します。アイテムセットを含むトランザクション数は、そのアイテムセットのサポート（支持度）として知られ、最小サポート（トランザクションの割合）は、アルゴリズムの制御パラメータの1つです。頻出アイテムセットの集合から、アルゴリズムはそれらのアイテム間の重要な相関をフィルタします。

１つのトランザクションデータセットでアイテムセットが頻出するからといって、データの集合全体で頻出であるとは限りません。そのためApriori は、データが分散された場合、データを2回走査することを必要とします。この手順の主要な参考資料は、Savasere, Omiecinski, and Navathe（1995）です。この手順は、Rajaraman and Ullman（2011）の代替アプローチと比較されます。

分散データに対する手順は次のとおりです。

1. 各トランザクション区分で頻出アイテムセットを見つけます。データの最初の走査で、MapReduce ペアを設定し使用します。
2. すべての頻出アイテムセットの集合を見つけます。このステップでは、ステップ 1 での別々のタスクの出力を読み込みますが、トランザクションの再読み込みは必要ありません。このステップは、2 番目の MapReduce ペアで完了します。
3. 集合のすべてのアイテムセットに対して、各トランザクションセットで発生する回数をカウントします。これは 2 番目のトランザクションデータセットの走査で必要です。ステップ 3 の出力は、全データに対して頻出なアイテムセットのすべての集合であり、つまり候補となる頻出アイテムセットである可能性があります。このステップは、ステップ 2 からの中間出力を読み込む Mapper によって実行されます。このステップで Mapper は、アイテムセットとトータルカウントを書き出します。
4. 候補アイテムセットの集合に対して、最小サポート のしきい値を満たすアイテムセットをフィルタします。このステップは、ステップ 1 と同じ MapReduce ペアで実行されます。

さまざまなデザインパターンによって効率の良し悪しがある可能性がありますが、デモの目的のために私たちは次のデザインを選択しました。Driver コードは 3 つの MapReduce ジョブ構成を管理し、Mapper2 は Reducer1 出力を検出し、Reducer3 は Reducer2 出力を検出する場所を知っている等を保証します。これらの関係を以下の図に示します。

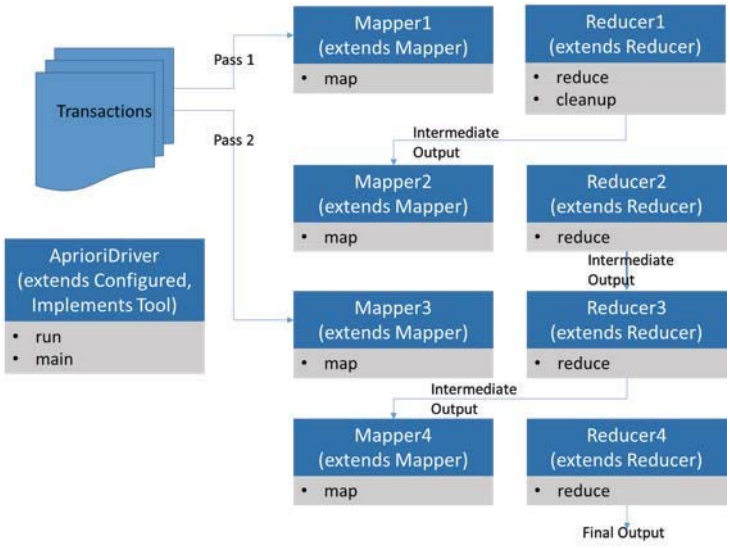

図4:Apriori の構成要素

```
public class AprioriDriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws IOException, InterruptedException,
            ClassNotFoundException {

        // Job 1
        Configuration conf1 = super.getConf();
        conf1.set( “minSupportPercentage”, “0.05” );
        conf1.set( “maxNumProducts”, “100” );
        conf1.set( “maxSetSize”, “3” );

        Job job1 = new Job(conf1, “Apriori Job 1” );

        job1.setJarByClass(getClass());

        job1.setMapperClass(AprioriMapper1.class);
        job1.setReducerClass(AprioriReducer1.class);

        job1.setNumReduceTasks(4);

        job1.setMapOutputKeyClass(Text.class);
        job1.setMapOutputValueClass(IntArrayWritable.class);

        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(Text.class);
```

```
job1.setInputFormatClass(TextInputFormat.class);
job1.setOutputFormatClass(TextOutputFormat.class);

TextInputFormat.addInputPath(job1, new Path(args[0]));
TextOutputFormat.setOutputPath(job1, new Path(args[1]));

// Job 2
Configuration conf2 = super.getConf();
Job job2 = new Job(conf2, " Apriori Map Task 2");
job2.setJarByClass(getClass());
job2.setMapperClass(AprioriMapper2.class);
job2.setReducerClass(AprioriReducer2.class);

job2.setMapOutputKeyClass(Text.class);
job2.setMapOutputValueClass(DoubleWritable.class);

job2.setOutputKeyClass(Text.class);
job2.setOutputValueClass(Text.class);

job2.setInputFormatClass(TextInputFormat.class);
job2.setOutputFormatClass(TextOutputFormat.class);

TextInputFormat.addInputPath(job2, new Path(args[1]));
TextOutputFormat.setOutputPath(job2, new Path(args[2]));

// Job 3
Configuration conf3 = super.getConf();
conf3.set("intermediateOutputPath", args[2]);
conf3.set("maxNumProducts", "100");
conf3.set("maxSetSize", "3");
conf3.set("minSupportPercentage", "0.05");

Job job3 = new Job(conf3, " Apriori Map Task 3");
job3.setJarByClass(getClass());
job3.setMapperClass(AprioriMapper3.class);
job3.setReducerClass(AprioriReducer3.class);

job3.setMapOutputKeyClass(Text.class);
job3.setMapOutputValueClass(IntArrayWritable.class);

job3.setOutputKeyClass(Text.class);
job3.setOutputValueClass(Text.class);

job3.setInputFormatClass(TextInputFormat.class);
job3.setOutputFormatClass(TextOutputFormat.class);

// Mapper3 reads the transactions again.
TextInputFormat.addInputPath(job3, new Path(args[0]));
TextOutputFormat.setOutputPath(job3, new Path(args[3]));
```

```
        // Job 4
        Configuration conf4 = super.getConf();

        Job job4 = new Job(conf4, " Apriori Map Task 4");
        job4.setJarByClass(getClass());
        job4.setMapperClass(AprioriMapper4.class);
        job4.setReducerClass(AprioriReducer4.class);

        job4.setMapOutputKeyClass(Text.class);
        job4.setMapOutputValueClass(Text.class);
        job4.setOutputKeyClass(Text.class);
        job4.setOutputValueClass(Text.class);

        job4.setInputFormatClass(TextInputFormat.class);
        job4.setOutputFormatClass(TextOutputFormat.class);

        TextInputFormat.addInputPath(job4, new Path(args[3]));
        TextOutputFormat.setOutputPath(job4, new Path(args[4]));

        // Run job1.
        job1.waitForCompletion(true);
        // Run job2.
        job2.waitForCompletion(true);
        // Run job3.
        job3.waitForCompletion(true);
        // Run job4.
        job4.waitForCompletion(true);
        return 0;
    }

public static void main(String[] args) throws IOException,
          InterruptedException, ClassNotFoundException,
          Exception {

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
              args).getRemainingArgs();
        if (otherArgs.length != 5) {
              System.err.println("Usage:AprioriDriver <input_dir> "
                    + "<intermediate_dir_1>"
                    + "<intermediate_dir_2"
                    + "<intermediate_dir_3"
                    + "<output_dir>");
              System.exit(2);
       }
       int res = ToolRunner.run(conf, new AprioriDriver(), args);
    }
}
```

入力フォーマットを以下に示します。各行には、日付、時間、トランザクション ID、当該トランザクションで購入したアイテムのリストが含まれます(アイテムは、ここでは一意の 整数ID でエンコードされますが、別の英数フォーマットである場合もあります)。

```
5/27/2015 11:22:13,0,67,56,70,84,4
5/27/2015 11:22:23,1,6,99,53,62,11,51,43
5/27/2015 11:22:33,2,93,56
5/27/2015 11:22:43,3,1
5/27/2015 11:22:53,4,81,60,31,16,19,47,76,2
5/27/2015 11:23:03,5,86,36,99
5/27/2015 11:23:13,6,82,53,57,47,0,46
5/27/2015 11:23:23,7,78,81,6,20,39,32,87
5/27/2015 11:23:33,8,17
5/27/2015 11:23:43,9,87,57,68,73,40,55,46
5/27/2015 11:23:53,10,50,3,42,11,13,1,66
5/27/2015 11:24:03,11,27,14,59
5/27/2015 11:24:13,12,52,74,7,15,57,88
…
```

図5:入力フォーマット

AprioriMapper1 は、入力ディレクトリからトランザクションを読み込み、<key, item> ペアを書き出します。ここでキーは Text に変換され、アイテムは単一トランザクションで購入された商品のリストです。商品のリストは、1つのIntArrayWritable に変換されます。メソッド context.write() を含む行をご覧ください。

```
public class AprioriMapper1
          extends Mapper<Object, Text, Text, IntArrayWritable> {

     @Override
     public void map(Object key, Text value, Context context)
               throws IOException, InterruptedException {

          SimpleDateFormat dateFormat = new SimpleDateFormat( “M/yy” );
          String origColumns[] = value.toString().split( “,” );

          try {
               Date trxDate = dateFormat.parse(origColumns[0]);
               String trxId = origColumns[1];

               if (origColumns.length > 1) {
                    IntWritable products[] =
                                   new IntWritable[origColumns.length - 2];

                    for (int i = 0; i < products.length; i++) {
                         Integer prodId = Integer.parseInt(origColumns[i + 2]);
                         products[i] = new IntWritable(prodId);
                    }

                 context.write(new Text(dateFormat.format(trxDate)),
                                                 new IntArrayWritable(products));
               }
          } catch (ParseException ex) {
              Logger.getLogger(AprioriMapper1.class.getName()).log(Level.SEVERE,
                                                                   null, ex);
          }
     }
}
```

キーを定義するために、月-年を使用しています。

```
SimpleDateFormat dateFormat = new SimpleDateFormat( “M/yy” );
…
    context.write(new Text(dateFormat.format(trxDate)),
                                        new IntArrayWritable(products));
```

Hadoop の区分には、複数キーのレコードが含まれる場合がありますが、1つのキーに対するすべてのレコードは単一の区分にある必要があり、さらに1つの区分はメモリに収まる必要があります。今回の例では、一ヶ月のすべてのトランザクションは同一区分内に存在する必要があります。一ヶ月にあまりにも多くのトランザクションがある場合は、より小さい単位（週、日など）を選択できます。
特定の Hadoop アプリケーション構成の一部分として適切なキーレベルが決定されます。AprioriMapper1 が、すべての取引を月-年で照合し、その後 AprioriReducer1 が、それらのチャンクを処理し、JMSL Apriori を呼び出して中間出力ディレクトリに頻出アイテムセットのリストを書き出します。

```
public class AprioriReducer1 extends
        Reducer<Text, IntArrayWritable, Text, Text> {

    int totalNumberOfTrx =  0;

    @Override
    public void reduce(Text key, Iterable<IntArrayWritable> values,
            Context context) throws IOException, InterruptedException {

        ArrayList trxList = new ArrayList();

        int numberOfTrx = 0;
        int numRows = 0;
        IntWritable[] iwa;

        for (IntArrayWritable val : values) {
            iwa = (IntWritable[]) val.toArray();
            trxList.add(iwa);
            numRows += iwa.length;
            numberOfTrx++;
        }
        totalNumberOfTrx += numberOfTrx;

        String sMinSupportPct
                =  context.getConfiguration().get( “minSupportPercentage” );
        double minSupportPct = Double.parseDouble(sMinSupportPct);
        String sMaxNumProducts
                =  context.getConfiguration().get( “maxNumProducts” );
        int maxNumProducts = Integer.parseInt(sMaxNumProducts);
        String sMaxSetSize = context.getConfiguration().get( “maxSetSize” );
        int maxSetSize = Integer.parseInt(sMaxSetSize);

        int xint[][] = new int[numRows][2];
        int idx = 0;
```

```
        for (int i = 0; i < trxList.size(); i++) {
            IntWritable[] A = (IntWritable[]) trxList.get(i);
            for (IntWritable A1 :A) {
                  xint[idx][0] = i + 1;
                  xint[idx][1] = A1.get() + 1;
                  idx++;
             }
        }


        Itemsetsfis = Apriori.getFrequentItemsets(xint, maxNumProducts,
                  maxSetSize, minSupportPct);
        int numberOfItemSets = fis.getNumberOfItemsets();

        DoubleWritable support;
        for (int i = 0; i < numberOfItemSets; i++) {
             String outString = “” ;
             int[] itemSet = fis.getItemset(i);
             support = new DoubleWritable(fis.getSupport(i));
             for (int j = 0; j < itemSet.length; j++) {
                  outString += Integer.toString(itemSet[j]) + “,” ;
             }
             String outStringF = String.format( “%s%n”, support.toString());
             context.write(new Text(outString), new Text(outStringF));
        }
    }

     @Override
    public void cleanup(Context context) throws IOException,
            InterruptedException{
        context.write(new Text( “TotalTrx,” ), new
                                  Text(Integer.toString(totalNumberOfTrx)));
    }
}
```

次のような方法でApriori パラメータが設定から取得されます。

```
String sMinSupportPct =
    context.getConfiguration().get( “minSupportPercentage” );
```

AprioriReducer1 は、<itemset, support> を 2 番目の中間出力ディレクトリに書き出します。トランザクションの全数を後続の MapReduce ジョブに渡すために、cleanup() メソッドを上書きしました。2 番目の MapReduce ジョブは、アイテムセットの集合を検索します。AprioriMapper2 はIdentity Mapperであり、AprioriReducer1 の出力を読み取って、そのまま AprioriReducer2 に送信します。ここで、reduce() メソッドはアイテムセットの集合を生成し、出力を 3 番目の中間出力ディレクトリに送信します。
集合のアイテムセットの一つ一つは、トランザクションのチャンクの少なくとも 1 つで頻出である一方、すべてのトランザクションでは頻出とは限らないため、全トランザクション上でもう一度数えられる必要があります。3 番目の MapReduce ジョブがこのステップを管理します。集合や候補となる頻出アイテムセットは、メモリに収まる必要があることに注意してください。通常これは妥当な要請です。もしあまりに多くの候補アイテムセットがある場合、最小サポートの割合があまりにも小さくなりすぎてしまいます。私たちの目的は強固な相関を発見することであり、すべての組み合わせを列挙することではありません。

```
public class AprioriMapper2 extends Mapper<Object, Text, Text, DoubleWritable> {
    // This is an identity mapper.
     @Override
     public void map(Object key, Text value, Mapper.Context context)
             throws IOException, InterruptedException {

          String origColumns[] = value.toString().split( “,” );
          int len = origColumns.length;
          String outString=” ” ;

          if(len >= 2){
          for(int i = 0; i < origColumns.length-2; i++){
              outString += (origColumns[i] +  “,” );
          }

   outString += origColumns[origColumns.length-2];
          double outDouble =
               Double.parseDouble(origColumns[origColumns.length -1]);
          context.write(new Text(outString), new DoubleWritable(outDouble));
          }
    }
}
```

```
public class AprioriReducer2 extends Reducer<Text, DoubleWritable, Text, Text> {

     // Performs the union of the candidate itemsets.
     @Override
     public void reduce(Text key, Iterable<DoubleWritable> values,
             Context context) throws IOException, InterruptedException {

          String outString;
          int sumOfSupports = 0;
          Iterator<DoubleWritable> iterator = values.iterator();

          for (Iterator<DoubleWritable> it = values.iterator(); it.hasNext();) {
               double value = it.next().get();
               sumOfSupports += value;
          }
          outString = String.format( “,%s%n” , Integer.toString(sumOfSupports));
          context.write(key, new Text(outString));
     }
}
```

AprioriMapper3 は、AprioriReducer3 とペアであること以外はAprioriMapper1 での map タスクの繰り返しです。

```
public class AprioriMapper3 extends Mapper<Object, Text, Text, IntArrayWritable> {

     @Override
     public void map(Object key, Text value, Context context)
               throws IOException, InterruptedException {

          SimpleDateFormat dateFormat = new SimpleDateFormat( “M/yy” );
          String origColumns[] = value.toString().split( “,” );

          try {
               Date trxDate = dateFormat.parse(origColumns[0]);

               if (origColumns.length > 1) {
                    String trxId = origColumns[1];
                    IntWritable products[] =
                              new IntWritable[origColumns.length - 2];

                    for (int i = 0; i < products.length; i++) {
                         Integer prodId = Integer.parseInt(origColumns[i + 2]);
                         products[i] = new IntWritable(prodId);
                    }

                    context.write(new Text(dateFormat.format(trxDate)),
                              new IntArrayWritable(products));
               }
          } catch (ParseException ex) {
          Logger.getLogger(AprioriMapper1.class.getName()).log(Level.SEVERE, null, ex);
          }
     }
}
```

AprioriReducer3 は、中間出力パスから候補アイテムセットを読み取るために、 setup() を上書きします。その後、AprioriMapper3 からトランザクションを受けとります。

```
public class AprioriReducer3 extends
          Reducer<Text, IntArrayWritable, Text, Text> {

     int totalTrx = 0;
     ArrayList listOfCandidateItemSets = new ArrayList();
     Itemset[] candidateItemSets;
     int numberOfSets;

     @Override
     public void setup(Context context) throws IOException {

          // Retrieve the intermediate output path.Configuration
          conf = context.getConfiguration();
          String intermediateOutputPath = conf.get( “intermediateOutputPath” );
```

```
        Path opPath = new Path(intermediateOutputPath);

        try {
                FileSystem fs = FileSystem.get(conf);
                FileStatus[] status = fs.listStatus(opPath);

                for (FileStatus statu : status) {
                        BufferedReader d = new BufferedReader(new
                                        InputStreamReader(fs.open(statu.getPath())));
                        String line;
                        String origColumns[];
                        String regexp = "[¥¥s,;¥¥n¥¥t]+" ;

                        int aRow[];

                        // Read the candidate itemsets from the intermediate path.
                        while (d.ready()) {
                                line = d.readLine();
                                origColumns = line.trim().split(regexp);
                                if (origColumns[0].contains( "TotalTrx" )) {
                                        this.totalTrx = Integer.parseInt(origColumns[1]);
                                } else {
                                        aRow = new int[origColumns.length];
                                        for (int i = 0; i < origColumns.length; i++) {
                                                aRow[i] =
                                                        Integer.parseInt(origColumns[i].trim());
                                        }
                                        listOfCandidateItemSets.add(aRow);
                                }
                                d.readLine();
                        }
                        d.close();
                }
        } catch (IOException | NumberFormatException ex) {
                Logger.getLogger(AprioriReducer3.class.getName()).log(Level.SEVERE, null, ex);
        }
        numberOfSets = listOfCandidateItemSets.size();
        candidateItemSets = new Itemset[numberOfSets];

        for (int i = 0; i < numberOfSets; i++) {
                int[] arow = (int[]) listOfCandidateItemSets.get(i);
                int[] itemSet = new int[arow.length - 1];
                System.arraycopy(arow, 0, itemSet, 0, arow.length - 1);
                candidateItemSets[i]
                        = new Itemset(itemSet, arow[arow.length - 1]);
        }
}

@Override
public void reduce(Text key, Iterable<IntArrayWritable> values,
                Context context) throws IOException, InterruptedException {
```

```
if(numberOfSets > 0){
// Process
ArrayList trxList = new ArrayList();
int numberOfTrx = 0;
int numRows = 0;
IntWritable[] iwa;

for (IntArrayWritable val : values) {
        iwa = (IntWritable[]) val.toArray();
        trxList.add(iwa);
        numRows += iwa.length;
        numberOfTrx++;
}


String sMinSupportPct
          =  context.getConfiguration().get( “minSupportPercentage” );
double minSupportPct = Double.parseDouble(sMinSupportPct);
String sMaxNumProducts
          =  context.getConfiguration().get( “maxNumProducts” );
int maxNumProducts = Integer.parseInt(sMaxNumProducts);
String sMaxSetSize = context.getConfiguration().get( “maxSetSize” );
int maxSetSize = Integer.parseInt(sMaxSetSize);

int xint[][] = new int[numRows][2];
int idx = 0;

// Build input data from data in the partition.
for (int i = 0; i < trxList.size(); i++) {
        IntWritable[] A = (IntWritable[]) trxList.get(i);
        for (IntWritable A1 :A) {
                xint[idx][0] = i + 1;
                xint[idx][1] = A1.get() + 1;
                idx++;
        }
}

int[] freq;

Itemsets candSets = new Itemsets(candidateItemSets, numberOfTrx,
        maxNumProducts, minSupportPct, maxSetSize);

// Count frequencies of the itemset in the current
// segment of transactions and emit the result.
freq = Apriori.countFrequency(candSets, xint);

for (int i = 0; i < candSets.getNumberOfItemsets(); i++) {
        String keyString =  “” ;
        int[] itemset = candSets.getItemset(i);

        for (int j = 0; j < itemset.length - 1; j++) {
```

```
                    keyString += Integer.toString(itemset[j]) + “,” ;
                }
                // Include the total number of transactions so that
                // the final reducer will have the information.
                keyString += Integer.toString(itemset[itemset.length - 1]); String out
                String = String.format( “,%s,%s%n” ,
                          Integer.toString(freq[i]),
                          Integer.toString(this.totalTrx));

                context.write(new Text(keyString), new Text(outString));
            }
        }}
}
```

最後の reducer は頻出 Itemsets を最終出力ディレクトリに書き出します。この例では最終出力は頻出アイテムセットのリストだけです。オプションとして今回のサンプルは、JMSL のメソッド Apriori.getAssociationRules() を使用してアソシエーションルールを生成し、その結果を使用して、最終出力を形成することができます。

```
public class AprioriReducer4 extends Reducer<Text, Text, Text, Text> {

    // Performs the final filtering for frequent itemsets.
    @Override
    public void reduce(Text key, Iterable<Text> values,
              Context context) throws IOException, InterruptedException {

        String sMinSupportPct
                  =  context.getConfiguration().get( “minSupportPercentage” );
        double minSupportPct = Double.parseDouble(sMinSupportPct);

        String outString;
        int sumOfSupports = 0;
        double minimumSupport = 0;
        Iterator<Text> iterator = values.iterator();

        for (Iterator<Text> it = values.iterator(); it.hasNext();) {
            String[] valueSplit = iterator.next().toString().split( “,” );
            sumOfSupports += Integer.parseInt(valueSplit[0]);
            minimumSupport = minSupportPct * Integer.parseInt(valueSplit[1]);
        }

        if (sumOfSupports >= minimumSupport) {
            outString = String.format( “,%s%n” ,
                                       Integer.toString(sumOfSupports));
            context.write(key, new Text(outString));
        }
    }
}
```

この Hadoop アプリケーションは、Apriori アルゴリズムと分散データセット上で、Apriori の結果を集めるためのスキームを使用してマーケットバスケット分析を実行します。最終出力は、頻出セットとそれに対応するサポート、つまりそのセットが発生したトランザクション数を表示します。以下におよそ 100 万のトランザクションに対する最終出力での頻出アイテムセットをいくつか示しました。

```
0               ,718729
0,1             ,592565
0,1,10          ,522135
0,1,19          ,355607
0,1,2           ,521825
0,1,3           ,355753
0,1,4           ,521838
0,1,5           ,356492
0,1,6           ,356687
0,1,7           ,356325
0,1,8           ,356652
…
```

結論として、Id {0} の製品が高い頻出（トランザクションの 72%）であり、そのスーパーセットのうちの {0, 1}、{0, 1, 10} なども、トランザクションで頻出度が高いです。次のステップでは、アソシエーションルールを生成するために、アプリケーションにメソッドを追加することになるでしょう。さらに、レポートを生成するために、処理後のステップを利用して製品 Id の代わりに製品名を使用することもできます。

まとめ

1 番目の例では、JMSL クラス Summary は、単純な平均を計算しました。Summary.update が入力値を 1 つ取得し、その入力値を使って平均値やその他の統計値を更新します。MapReduce が想定している通りに<key, value> のペアの集合として一変量のデータを取り扱うことは素直な手順です。
Hadoop は3 ノードのクラスタ上で、map()、reduce()メソッドとその他のクラスを使用してネットワーク全体のデータを管理し、最終的に各データ要素に対して Summary.update() を呼び出し、ユニークなキーによる結果（平均）を収集します。

JMSL の LinearRegression は、入力レコードごとに、指定された従属変数1つと複数の独立変数といった複数の値を必要とします。このデータ型をMapReduce が適切に使用できるフォーマットに変換することが2 番目のサンプルにおける課題でした。最も優れた解決法は、Hadoop クラス ArrayWritable のカスタム拡張を DoubleArrayWritable クラスに書き込むことでした。この新しいクラスを使用することで、<key, value > ペアは <key, DoubleArrayValues> ペアになります。

3 番目の例では、JMSL の Apriori を分散データ セット上でテストしました。Apriori は集計を実行するためにデータ全体に対して2回の走査を必要とします。問題の性質を活かして、3 番目の例は、Hadoop を使用する上で次のような4つの新しい使い方を明らかにしました。

- 複数の MapReduce ジョブ
- 中間出力ディレクトリからの読み取り
- <key, value> ペアを配列へ格納
- 設定パラメータを mapper と reducer に渡す

ビッグデータにまつわる高揚感は、より的確な結果や新しい洞察が得られることへの期待です。というのも、私たちは今後より多く、さらに正確に世界を計測するようになるからです。これを実現するための数多くの検討事項の中で、特に大規模なクラスタ上で効率的に実行できる、高度にスケーラブルな数学的/統計的アルゴリズムが重要です。すべてのデータを 一ヶ所に集めるという従来の方法はもはや不合理で実現不可能であるためです。根本的にはこのことはアルゴリズムを並列および分散コンピューティングに適用することを意味します。

こういった基礎的事実にもとづいてローグウェーブソフトウェアはIMSL数値ライブラリの最新リリースを推し進め、その製品ロードマップをお知らせします。ここで述べられた JMSL アルゴリズムについて詳しくは、IMSL Numerical Libraries を参照してください。

参考資料

Agrawal, R. and Srikant, R. (1994), “Fast algorithms for mining association rules (アソシエーションルールのマイニングのための高速アルゴリズム),” *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29 - September 1, 1994.(1994 年 8 月 29 日～ 9 月 1 日 チリ サンティアゴにおける「大規模データベースに関する第20回国際カンファレンス会報」)

Neter, John; Wasserman, William; and Kutner, Michael H. (1990), *Applied Linear Statistical Models*, 3rd ed. (線形統計モデルの適用、第 3 版), Richard D. Irwin, Inc. Homewood, Ill.

Rajaraman Anand and Ullman, Jeff David (2011), *Mining of Massive Datasets (大規模データセットのマイニング)*, Cambridge University Press, Cambridge, UK.

Savasere, Ashok; Omiecinski, Edward; and Navathe, Shamkant (1995), *“An Efficient Algorithm for Association Rules in Large Databases (大規模データベースにおけるアソシエーションルールの効率的アルゴリズム)”*, Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995.

White, Tom (2012).*Hadoop:The Definitive Guide*O’Reilly Media, 3rd edition.(邦題：Hadoop 第 3 版)

付録

Maven プロジェクト

私たちはMaven を使用した NetBeans で 実際にHadoopJMSLExamples.jar ファイルを構築しました。次の依存関係を pom.xml に追加しました。

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>2.7.0</version>
    <type>jar</type>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>1.2.1</version>
    <type>jar</type>
</dependency>
```

```
    <dependency>
      <groupId>jmsl</groupId>
      <artifactId>jmsl</artifactId>
      <version>8.0</version>
      <scope>system</scope>
      <systemPath>${basedir}¥lib¥jmsl-8.0.jar</systemPath>
    </dependency>
```

ちなみに私たちはこれらの例を Windows で実行する機会がありませんでした。

クラスタ上でのジョブの実行

コマンドラインからHadoop アプリケーションを実行するためには、最初にユーザーを HDFS に切り替えることが必要です。これは、Hadoop のインストールがどのように設定されたかに大きく依存しますが、以下のようになります。

```
    <user1>@myHadoopServer <136> su hdfs
    Password:
    hdfs@myHadoopServer:<homedir>$          <****>
```

Hadoop ファイルシステムをリスト表示するには、次のようにコマンド hadoop fs -ls <dirname> を使用します。

```
    hdfs@UbuntuServer1:<homedir>$ hadoop fs -ls /user/hdfs/
```

たとえば、入力データディレクトリに入力ファイルが含まれていることを確認するには、ディレクトリを表示します。

```
    hdfs@UbuntuServer1:<homedir>$ hadoop fs -ls /user/hdfs/SummaryInput
```

データファイルがこのディレクトリに存在しない場合、Hadoop ファイルシステムコマンド “put” を使用して、入力ディレクトリにデータファイルを配置します（テスト目的）。

```
    hdfs@UbuntuServer1:<homedir>$ hadoop fs -put Number* /user/hdfs/SummaryInput
```

コマンドラインからジョブを実行するための一般的なフォーマットは次のとおりです。

```
    $ hadoop jar <nameOfJarFile>.jar <class_directory/nameOfMainClass> /
    <Hadoop generic options> <args expected by the main class>
```

<class_directory> はJava プロジェクトの構成によって異なり、一般的なオプションや main クラスで期待されるその他の引数はスペースで区切られます。私たちの和を求める例ではコマンドは次のようなフォーマットです。

```
    $ hadoop jar HadoopJMSLExamples.jar com.mycompany.hadoopjmsl/SummaryDriver /                -libjar /
    lib/jmsl.jar /user/hdfs/SummaryInput /user/hdfs/SummaryOutput
```

このコマンドが動作するためにはjmsl.jar を含むローカルディレクトリ /lib/ が存在し、さらにその jar ファイルが$HADOOP_CLASSPATH に追加済みである必要があります。

```
    hdfs@UbuntuServer1:<homedir> export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:jmsl.jar
```

Apriori の例にではいくつかのコマンドライン引数を指定します。

```
$ hadoop jar HadoopJMSLExamples.jar com/mycompany/hadoopjmsl/AprioriDriver / -libjar /
lib/jmsl.jar /user/hdfs/InputDirectory /user/hdfs/IntOutput1 / /user/hdfs/IntOutput2 /
user/hdfs/IntOutput3 /user/hdfs/FinalOutput
```

出力ディレクトリがジョブの実行時にすでに存在する場合はエラーになります。
ディレクトリを空にするには次のコマンドを実行します。

```
$ hadoop fs -rm /user/hdfs/Summary_output/*
```

ディレクトリを削除するには次のコマンドを実行します。

```
$ hadoop fs -rmdir /user/hdfs/Summary_output
```

ジョブが例外を発生せず終了した場合、Hadoop は画面に次のように出力します。

```
15/06/26 10:21:27 INFO mapreduce.Job:Counters:38
      File System Counters
            FILE:Number of bytes read=22885514
            FILE:Number of bytes written=26089332
            FILE:Number of read operations=0
            FILE:Number of large read operations=0
            FILE:Number of write operations=0
            HDFS:Number of bytes read=922622
            HDFS:Number of bytes written=4712
            HDFS:Number of read operations=177
            HDFS:Number of large read operations=0
            HDFS:Number of write operations=13
      Map-Reduce Framework
            Map input records=1396
            Map output records=1396
            Map output bytes=16618
            Map output materialized bytes=19470
            Input split bytes=1516
            Combine input records=0
            Combine output records=0
            Reduce input groups=453
            Reduce shuffle bytes=19470
            Reduce input records=1396
            Reduce output records=453
            Spilled Records=2792
            Shuffled Maps =10
            FailedShuffles=0
            Merged Map outputs=10
            GC time elapsed (ms)=521
            CPU time spent (ms)=0
            Physical memory (bytes) snapshot=0
            Virtual memory (bytes) snapshot=0
            Total committed heap usage (bytes)=2825912320
```

```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=111378
        File Output Format Counters
                Bytes Written=4712
```

出力を確認するには、ここでもHadoop の fs コマンド、-ls と -cat を使用できます。

```
$ hadoop fs -ls /user/hdfs/SummaryOutput

-rw-r--r--      3 hdfs supergroup                0 2015-06-26 10:21 SummaryOutput2/_SUCCE
SS
-rw-r--r--      3 hdfs supergroup             4712 2015-06-26 10:21 SummaryOutput2/part-r
-00000
$ Hadoop fs     -cat /user/hdfs/SummaryOutput/part-r-00000

                        BDK184.0
                        ABC275.0
                        EAF177.0
                        AGC71.0
                        IJK47.0
                        DAL187.0
                        MUZ125.0
                        TLK258.0
                         …
```

ローグウェーブは、ミッションクリティカルなアプリケーション向けのソフトウェア開発ツールとサービスを提供しています。当社が提供する信頼性の高いソリューションは、ますます複雑化、大規模化するソフトウェア開発をサポートし、コードから得られる価値を高めます。補完的なクロスプラットフォームツールを含むローグウェーブの製品とサービスは、アプリケーション迅速な開発を支援し、ソフトウェアの品質向上やコードの整合性を保証すると同時に、開発サイクルの時間を短縮することができます。