Hadoop 上で動作する Sawzall サブセットの実装

中田秀基[†1] 井上辰彦 [†2,†1] 工藤知宏 [†1]

Sawzall は、Google が 2006 年に発表した大容量データの並列バッチ処理に適した言語である。Sawzall の計算モデルは MapReduce 型の分散演算であるが、リダクション操作を組み込みの Aggregator に限定することで、エンドユーザによる容易な記述を可能にしている。われわれは現在開発中の並列データ処理機構上の言語処理系を開発するための 1 ステップとして、Hadoop 上で動作する Scala 言語による Sawzall 言語のサブセット処理系を実装した。文法やセマンティクスに関しては明確な定義がなかったため、2006 年の論文をベースに推測した。Scala は Java VM 上で動作するため、Java で記述される Hadoop 上での実行は容易である。構文解析に Scala 言語の Parser Combinator を用いることで、処理系の記述量を削減した。言語インタプリタは、Hadoop の Mapper 上で動作する。Reducer 上では処理系の提供する Aggregater が動作する。Hadoop で直接記述した場合と、プログラム量および実行速度の点で比較を行った。比較の結果、プログラム量は大幅に小さくなる一方、実行速度の面でも一定のオーバヘッドがあることが確認された。

An implementation of Sawzall subset interpreter working on Hadoop

SawzallCloneHidemoto Nakada ,[†1] Tatsuhiko Inoue [†2,†1] and Tomohiro Kudoh [†1]

Sawzall is a script language designed for batch processing of large amount of data, which is introduced by Google in 2006. The processing model of Sawzall is the MapReduce. Sawzall allows programmers only to program 'mappers' to ease the burden. Sawzall provides a set of 'built-in aggregators', from which programmer choose reducing function. We are developing distributed data processing system for large scaled data. As a part of the project, we have iplemented an interpreter for Sawzall subset in Scala language. We employed parser combinator for syntax parsing. Currently, the system is running on Hadoop. In the paper, we provide detailed implementation of the system. We also compared the system with the native Hadoop in terms of program size and execution speed. We confirmed that with Sawzall program size is much smaller, while there are certain overhead in terms of execution speed.

1. はじめに

容易に並列計算を記述できる実行パラダイムとして MapReduce[1] が注目されている。MapReduce は Hadoop を始めとする処理系の普及によって、科学技術計算のみならず業務データの解析などに広く用いられつつある。われわれは、MapReduce の高速な実装[2] を進めるとともに、MapReduce を汎化した計算機構の検討を行い、さらに汎化計算機構上の計算記述を補助するための言語処理系を検討しているこの言語処理系検討の 1 ステップとして、Google による MapReduce 向け言語である Sawzall のサブセット (以下 SawzallClone) を実装し Hadoop 上で実行できる環境を実現したので報告する。

実装開始時には Sawzall は論文で公表されていたのみで、文法やセマンティクスに関しては明確な定義がなかったため、2006 年の論文をベースに推測したものを実装した。2010 年に Google の Sawzall 処理系 szl がオープンソースで公開されたが、この処理系と SawzallClone とでは文法上いくつかの相違点がある。

SawzallClone の実装は Scala 言語によっておこなった。これは、Java で記述された Hadoop 処理系との相性を考えてのことである。構文解析に Scala 言語の Parser Combinator を用いることで、処理系の記述量が削減できた。

Hadoop 上での実行は、mapper として SawzallClone インタプリタを、reducer としてアグリゲータを稼働させることで実現した。Hadoop の Java コードで直接記述した場合と比較し、プログラム量は大幅に小さくなる一方、実行速度の面でも一定のオーバヘッドがあることを確認した。

次節以降の構成は次の通りである．2 節で研究の背景について述べる．3 節で Sawzall の言語仕様を紹介する。4 節で SawzallClone の設計と実装について述べる．5 節で評価を行う。6 節で関連研究に言及し，7 節にまとめを示す．

†1 独立行政法人 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
†2 株式会社創夢
SOUM Corporation

```
>>> map(lambda x: x + 1, [1, 2, 3, 4, 5])
[2, 3, 4, 5, 6]

>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

図 1 Python における map と reduce

2. 背　　景

2.1 MapReduce

MapReduce[1] は、並列演算のパラダイムのひとつであり、Google が内部的にログ解析等に利用していることで知られている。

MapReduce の名は、Lisp などの関数を第一級オブジェクトとして扱うことのできる言語で一般的な 2 つの高階関数 map と reduce に由来する。map 関数はリストと関数を引数とし、リストの各要素に対して同一の関数を適用した結果のリストを返す関数である。reduce 関数は、リストと関数を引数とし、リストの各要素に対して関数を用いた縮約操作を行う。Python 言語による map と reduce の例を図 1 に示す。

これらの関数の特徴は、実行順序を規定しておらず、依存関係がないため潜在的に並列実行が可能なことである。特に、map 操作ではすべての関数適用を並列に実行することができる。これに対して、reduce 操作では、縮約関数が可換であることを前提とすれば、例えば 2 分木を構成して、並列に実行することが可能ではあるが、その並列度は限定的である。

MapReduce は、この map 関数と reduce 関数に想を得た並列実行パラダイムである。MapReduce では、map 操作を並列実行し、その結果に対して複数の reduce 操作を同時に行う。reduce 操作内部が並列化されているわけではなく、複数の reduce を同時に行うのである。

MapReduce による map 処理は出力として複数のキーとバリューのペアを出力する。複数の map 処理から出力されたキーバリューペア群はキーでグループ分けされ、それぞれのキーに対応するグループの内部で reduce 処理を行う。

2.1.1 Combiner

reduce 処理が順序に非依存である場合、reduce 処理を一括して行う必要はない。reduce 処理によってデータ量を削減できることが期待できる場合には、Map の直後に reduce を部分的に行うことができる。この関数を Combiner と呼ぶ。Combiner は Reducer と同じ関数になる場合が多いが、そうでない場合もある。

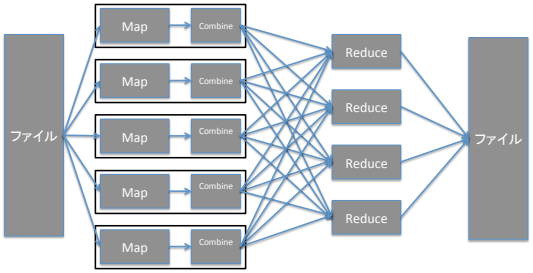

図 2 MapReduce の概要

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

図 3 proto ファイルの例

2.2 Hadoop

Hadoop[3] は、Yahoo が開発した MapReduce 処理系である。現在は Apache のトッププロジェクトの一つとなっている。Hadoop には大きく分けて 2 つの側面がある。ひとつは複数のノードから構成される大規模な分散ストレージ、もう一つが MapReduce に基づく分散並列処理系である。これらは関連してはいるが独立しており、Hadoop 分散ストレージに依存しない分散並列実行を行うことも可能である。分散ストレージ部分は HDFS と呼ばれる。Hadoop は大規模なデータのバッチ処理に適しており、さまざまな目的に広く使われている。

2.3 Protocol Buffers

Protocol Buffers[4] は、Google が内部で広範に用いている、言語非依存の構造データのバイナリ表現形式である。XDR[5] と目的は類似しているが、データ量の削減に主眼を置いている点に特徴がある。

Protocol Buffers では、データ構造の定義を言語非依存の IDL で記述する。このファイルを proto ファイルと呼ぶ。proto ファイルの例を図 3 に示す。

このような proto ファイルをコンパイラでコンパイルすることで、言語依存の構造体定

義と、シリアライズ、デシリアライズコードが生成される。Google が提供している処理系 protoc では、C++、Java、Python がサポートされており、他の言語に関してもオープンソース実装が入手可能である。

3. Sawzall

Sawzall[6] は、Google が内部的に利用しているデータ処理用の言語で、Google の MapReduce 実装上で稼働している。Sawzall は、MapReduce モデルの map 部と reduce 部のうち、reduce 部を Sawzall が提供する固定的な Aggregator で提供し、ユーザには map 部分のみを記述させる。ユーザが記述する範囲を限定することによって、非技術者のユーザでもさらに容易に並列プログラミングを行うことを可能にしている。

Sawzall は型付きの言語である。Sawzall コードはマップの入力データ 1 レコードに対して処理を行う。複数のレコードにまたがった処理を記述することはできない。レコード単位で処理するという点は、テキストファイルを 1 行ずつ処理する Awk 言語と類似している。ただし、Awk では行単位の処理を行いながら内部状態を更新することで、文書全体に対する処理を行うことができる。これに対して、Sawzall では言語処理系内部の状態はレコードごとにリセットされる。文書全体に対する総計処理は Aggregator 部分で行われる。

3.1 Protocol Buffers との連携

Sawzall は Protocol Buffers でマーシャリングされたデータ構造の処理を前提としている。このため、Protocol Buffers 用に定義された proto ファイルを取り込む構文として `proto` 文が用意されている。

```
proto "p4stat.proto"
```

proto ファイルを取り込むと、proto ファイル内の定義された構造体に相応する構造体が Sawzall 内で定義されると同時に、構造体とバイト列の間の自動変換（マーシャリングとアンマーシャリング）が定義される。

3.2 変数 input

Sawzall プログラムは入力を特殊な変数 `input` から受け取る。これは awk プログラムが変数`$0` に処理の対象となる行を受け取るのと類似している。

`input` の型は byte 列であるが、これを適当なデータ型に型変換して利用する。一般には Protocol Buffers で定義された型を用いる場合が多い。ただし、Sawzall では型変換は暗黙裡に行われるので、単なる代入の形をとる。

表 1 Sawzall の代表的なテーブル

テーブル名	意味
collection	emit されたすべての要素を含む集合を作る
maximum	値の大きい順に指定された要素数を保持
sample	指定された要素数を統計的にサンプリング
sum	emit された要素をすべて積算
top	頻度の高いものを指定された要素数保持。統計的処理
quantile	出力された値を、指定した数で分位する数を統計的に求める
unique	重複を排除した要素数を推定

```
WhenStatement = 'when' '(' {QuantifierDecl} BoolExpression ')' Statement.
QuantifierDecl = var_name ':' quantifier Type ';'.
quantifier = 'all' | 'each' | 'some'.
```

図 4 when 文の定義

```
log: P4ChangelistStats = input;
```

3.3 テーブルと emit

Aggregator は、Sawzall の言語要素としてはテーブルという特殊な型として表現され、キーワード `table` を用いて定義される。

```
submitsthroughweek: table sum[minute: int] of count: int;
```

ここで、`sum` は予め定義されたテーブルの型であり、出力された結果を積算していくタイプのテーブルを示している。ここでは、`int` 型をおさめる `sum` 型テーブルの配列を定義している。

このテーブルに対して値を出力するのが `emit` 文である。出力された値は Aggregator に送出され積算される。

```
emit submitsthroughweek[minute] <- 1;
```

表 3.3 に、Sawzall の代表的なテーブルを示す。

3.4 when 文

Sawzall の構文で特徴的なのが When 文である。数量限定子（quantifier）を付記した複数の変数に対して、条件式が真となるときに後続の文を実行する。図 4 に when 文の BNF を示す。

数量限定子としては `all`、`some`、`each` の 3 種類が選択できる。`all` は、その変数の候補

```
when (
  somei: some int;
  somej: some int;
  lowercase(valuewords[somei]) == values[somej]
)
  emit valuecount[values[somej]] <- 1;
```

図 5 when 文の例

すべてが条件をみたすときにのみ、後続式が 1 度だけ実行される事を意味する。each は、その変数の候補のうち条件をみたす場合にのみ後続式が実行される事を意味する。すなわち、何度か実行される可能性がある。some は、その変数の候補のうち条件をみたすもののうち 1 つに対して、一度だけ実行される事を意味する。

図 5 に示すプログラムは、when の使用例である。valuewords[somei]) と values[somej] が一致する場合に、一度だけ後続の emit 文が実行される。

3.5 Sawzall のプログラム例

図 6 に、文献[6] より抜粋した Sawzall によるログ解析プログラムの例を示す。このプログラムは、Protocol Buffers 形式で格納されたログデータの頻度を分単位で集計するものである。

1 行目の proto 文は、このプログラムが入力として期待するデータの型を定義している。データは Protocol Buffers の proto ファイルで与える。2 行目は、Aggregator の定義である。Aggregator はキーワード table を用いて指定する。ここでは、sum 型の Aggregator が用いられている。この Aggregator は与えられたデータを積算する Aggregator である。Aggregator に関しては後述する。

4 行目は入力データを型変換して変数 log に代入している。P4ChangelistStats は、p4stat.proto 内で定義されたデータ型である。変数 input は特殊な変数で、入力レコードがバインドされる。5,6 行目では、ログの 1 行を表す変数 log から記録された時刻をとりだし、その時刻を分単位に変換している。

9 行目の emit 文は、Aggregator に対する出力である。2 行目で定義された sum 型のテーブルに 1 を出力している。sum は積算を行うので、結果として出力された回数が記録されることになる。

```
1  proto "p4stat.proto"
2  submitsthroughweek: table sum[minute: int] of count: int;
3
4  log: P4ChangelistStats = input;
5
6  t:      time = log.time; # microseconds
7  minute: int  = minuteof(t)+60*(hourof(t)+24*(dayofweek(t)-1))
8
9  emit submitsthroughweek[minute] <- 1;
```

図 6 Sawzall プログラムの例

4. SawzallClone の設計と実装

4.1 処理系の構造

SawzallClone 処理系は、大別して SawzallClone インタプリタと、Aggregator 実装、および全体を処理するドライバ部分から構成される。

ドライバは Hadoop のエントリポイントとなる。ドライバは、Sawzall スクリプトを読み込みパースツリーを作成する。さらに後述するようにスクリプトから参照されている proto ファイルから、処理対象となる構造体データのメタ情報を取得する。このパースツリーとメタ情報は、Hadoop の設定情報の一部として Mapper に渡される。

Mapper 内部ではインタプリタ本体と Combiner となる Aggregator 実装が稼働する。インタプリタはパースツリーを解釈して実行する。その際 Hadoop からデータの供給を受ける。インタプリタから emit されたデータは Combiner に渡され、処理される。さらにシャッフルを通して Reducer に渡される。

Reducer 上では、Aggregator 実装が動作する。処理したデータは HDFS に Protocol Buffers 形式ではき出される。

4.2 Protocol Buffers 情報の取り込み

3.1 で述べたように、proto 文で指定された proto ファイルの情報を読み込み、処理する必要がある。

この目的のために、われわれは Google による Protocol Buffers 処理系 protoc と、Java ランタイムを用いた。protoc は proto ファイルを解析し、proto ファイルに定義された構造体のメタ情報を Protocol Buffers 形式で出力する機能と、各言語に対応したスタブコー

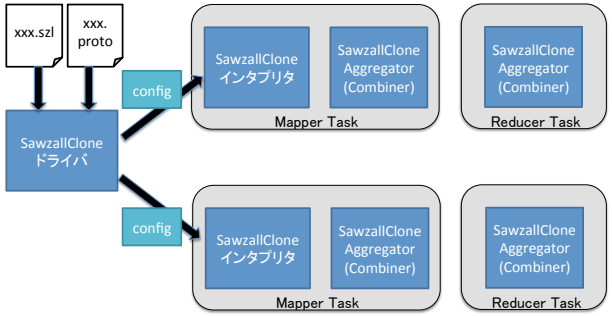

図 7 SawzallClone の実装

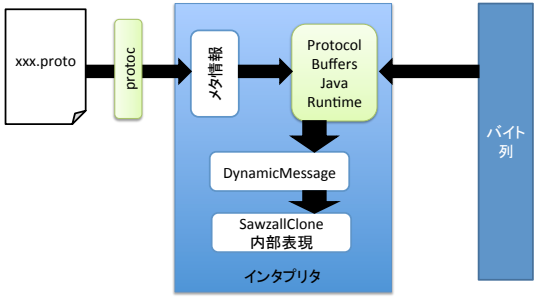

図 8 Protocol Buffers 情報の読み込み

ドを生成する機能を持つ。Java ランタイムは、メタ情報とバイト列を入力として構造データを構築する機能を提供する。Java スタブコードはこのランタイムを利用する。

SawzallClone では、スタブコードを用いず、メタ情報のみを用いた。`proto` 文を読み込んだ時点で、protoc を外部プロセスとして起動し、メタ情報を取得する。このメタ情報と入力バイト列を Java ランタイムで処理して、構造データを作成、さらにこれを SawzallClone 内部表現に変換して内部処理に用いている。

4.3 パーサコンビネータ

実装言語としては、Scala を選択した。これは、Hadoop を当初の実行環境と想定したため、Hadoop のネイティブ API を直接利用できることが必須だからである。

Scala を選択したもうひとつの理由はパーサ部分をパーサコンビネータで記述できるからである。パーサコンビネータは、下向き構文解析を行う手法で、部分構文を解析する関数を高階関数で組み合わせることによってより大きい部分に対する構文解析関数を構築する。

```
def whenStmt     = ("when" ~> "(" ~> whenVarDefs ~ expr ~< ")" ~ statement ) ^^@ WhenStmt
def whenVarDefs  = rep1(whenVarDef)
def whenVarDef   = (varNames ~< ":" ~ whenMod ~ dataType ~< ";") ^^@ WhenDefVar
def whenMod: Parser[WhenMod] = "some" ^^^@ SomeMod() |
                               "each" ^^^@ EachMod() |
                               "all"  ^^^@ AllMod()
```

図 9 パーサコンビネータによる when 文の記述

BNF 表現と解析関数が対応するため、記述が容易である。また、プログラム本体に解析関数の記述を埋め込むことができるため、yacc などのパーサジェネレータを利用する手法と比較して、開発が容易であるという特徴がある。

この手法は、関数が一級オブジェクトとなっている言語であれば利用できるが、Scala では演算子のオーバーロードができるためより BNF に近い記述でパーサを記述することができる。また、ライブラリが標準で提供されているため利用が容易である。

図 9 に前出の when 文を定義したパーサコンビネータを示す。yacc などのように独立したファイルとして記述されているわけではなく、プログラムの内部に Scala プログラムの一部として記述されているにも関わらず、直感的に表現できていることがわかる

5. 評　　価

SawzallClone インタプリタのオーバヘッドを測定するために、同一のプログラムを Hadoop 上の Java API と Sawzall で実装し、比較した。

5.1 評価プログラム

評価プログラムとしては、ワードカウントを用いた。ワードカウントは、文章中の単語の発生回数をカウントするプログラムである。図 10,11 に Sawzall と Hadoop による実装をそれぞれ示す。Sawzall は 6 行、Hadoop API では 60 行程度となっている。

5.2 評 価 環 境

評価には 16 ノードのクラスタを用いた。各ノードは 4 コアの xeon 5580 を 2 基搭載しているが、各ノードでは 1 スレッドのみ動作するように調整して実行した。ノード間ネットワークは 10G Ether である。ストレージは SATA のディスクである。

入力データは、HDFS 上にレプリケーション数 16 で配置した。つまりすべてのノード上にすべてのデータがある状態である。このためすべてのマップスレッドはローカルのディス

```
document: string = input;
words: array of string  = split(document);

t: table sum[string] of int;

for (i: int = 0;i < len(words);i = i + 1) {
    emit t[words[i]] <- 1;
}
```

図 10 ワードカウントの Sawzall による実装

クから読みだすことになり、マップ時にはネットワーク通信は生じない。

対象となるテキストデータは、約 42MB で、これをワーカ数個用意して用いた。つまりワーカひとつあたりのデータ量は常に同じである。SawzallClone の入力は Protocol Buffers でエンコードした。このとき 1 行をひとつのレコードとした。レコード数が多いため Protocol Buffers のオーバヘッドも大きく、サイズは 50MB 程度となっている。

5.3 結　　果

測定は、ワーカ数を 1-16 と変化させて行った。このときリデューサ数は常にワーカ数と同じとした。また、Hadoop Java API、SawzallClone ともに Combiner ありの場合となしの場合を計測した。図 12 に示すのは 5 回の測定の平均値である。

前述のとおり、Map ワーカ数とデータ量が比例関係にあり、Map ワーカあたりのデータ量は定数であるため、Map の実行時間は常に同じになっている。グラフが右肩上りになっているのは Map ワーカからの出力のデータ量が増えているため、シャフルと Reduce により多くの時間がかかるようになるためである。

Sawzall は Hadoop に対して倍近くの実行時間がかかっている。この原因は、インタプリタのオーバヘッドと、Protocol Buffers を用いたマーシャリング、アンマーシャリングのコストによるものと考えられる。

また、Combiner の効果はいずれの場合にも明らかで、同程度の効果があることがわかった。

6. 関 連 研 究

6.1 Google による Sawzall 実装

2010 年 8 月に、Google の Sawzall 実装 szl がオープンソースとして公開された[7]。szl は C++で記述されており、バイトコードへのインタプリタとバイトコード処理系で構成されている。公開された部分はシングルプロセスで実行される。Hadoop 等を利用して容易に並列実行することはいまのところできない。

szl と SawzallClone はいくつかの点で異なっている。これは、SawzallClone が文献[6] を

```
public class WordCount {
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        while (values.hasNext())
            sum += values.next().get();
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }
}
```

図 11 ワードカウントの Hadoop API による実装

ベースに推測した仕様に基づいて実装されているためである。主要な相違点を表 6.1 に示す。

6.2 Hive

Hive[8][9] は、 Hadoop 上に構築されるウェアハウススケールデータベースで、SQL に類似した言語 QL によるデータの検索が可能にする。Hive は Facebook で利用されており、700TB ものデータ（複製により実データサイズは 2.1PB）を管理しているという。

Hive は HDFS 上に構造データをシリアライズした形で保持する。データベースのスキーマや、テーブルの HDFS 上での位置などのメタ情報は、Hadoop 外部の RDB で管理する。これを Metastore と呼ぶ。

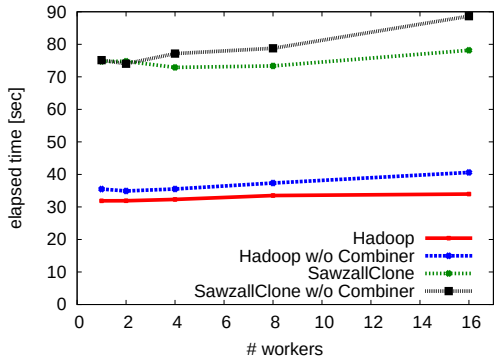

図 12 ワードカウントの実行結果

表 2 szl と SawzallClone の相違点

	szl	SawzallClone
関数定義	`function(): int {...};` のように定義。	定義できない
型定義	`type my_type = type;` のように定義	定義できない
switch 文	あり	なし
do while 文	あり	なし

```
INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;
```

図 13 Hive QL による検索の例

QL 内で利用できる関数は組み込みで用意されているが、ユーザ定義関数（UDF）で拡張することも可能である。UDF は特定のインターフェイスを実装する形で Java で記述する。

Hive は次に述べる Pig と比較すると、遙かに伝統的な RDB に近い。テーブルを CREATE 文で明示的に作成するし、検索言語 QL の語彙、セマンティクスともに SQL に近い。このため、RDB からの乗り換えが比較的容易である点が利点である。

6.3 Pig

Apache Pig[10)11)] は、Hadoop 上のデータをデータフロー的に処理する言語処理系である。Pig の言語 (Pig Latin) で記述されたプログラムは、一連の MapReduce にコンパイル

```
records = LOAD 'input/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records =  FILTER records BY temperature !=  9999 AND
  (quality == 0 OR quality == 1);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

図 14 Pig Latin プログラムの例

され、実行される。Pig Latin ではデータに対する一連の操作を手続き的に記述する。手続き型の処理に慣れ親しんだ一般のプログラマにとっては、SQL 同様に宣言的に記述する Hive と比較して、より直感的なプログラミングが可能である。

Pig Latin で利用できる操作は、データのロードとストア、フィルタリング、グループ化と結合、ソート、集約と分割の 5 つに分類できる。これらの操作を繰り返し利用してデータに対する操作を記述する。Pig Latin によるコードの例を図 14 に示す。年毎の最高気温を求めるプログラムの例である。

Pig の記述力は限定されているため、定型的な処理しか記述できない。しかし、データ入出力、フィルタリング、グルーピングなどの各操作に対してユーザ定義関数 (UDF) を定義することによって、拡張することができる。UDF は特定のインターフェイスを持つ Java クラスとして記述する。

6.4 Jaql

Jaql[12)13)] は、Hive や Pig 同様に Hadoop 上に蓄積したデータに対する検索を行う、問い合わせ言語である。Jaql は、JSON(JavaScript Object Notation) 形式でデータが HDFS 上に格納されていることを前提とする。いわば、データの側に明示的な構造があるため、外部に明示的にスキーマを持つ必要がない。

図 15 に、Jaql による MapReduce プログラムの例を示す。このプログラムは、ファイル `sample.dat` 内に格納された構造データを、属性 `x` の値でグループ分けし、その個数をカウントするプログラムである。

6.5 議　　論

Hive QL、Pig Latin、Jaql はいずれも Sawzall と同様に、MapReduce による大規模データ処理を容易に記述させることを目的としている。

この 4 つのなかで Sawzall が特徴的なのは、他の言語がデータフロー部分を記述させることに特化しているのに対して、Sawzall はマップ処理自体の記述に特化していることであ

```
mapReduce(
  { input:  {type: "hdfs", location: "sample.dat"},
    output: {type: "hdfs", location: "results.dat"},
    map:    fn($v) ( $v -> transform [$.x, 1] ),
    reduce: fn($x, $v) ( $v -> aggregate into {x: $x, num: count($)} )
  });
```

図 15 Jaql プログラムの例

る。他の 3 つの言語では、マップでの処理は限定的な組み込み関数、もしくは Java で記述した UDF で記述することを前提としている。したがって、Sawzall のカバーしている領域は、他の 3 つの言語と相補的であると言える。

7. おわりに

現在開発中の並列データ処理機構上の言語処理系を開発するための 1 ステップとして、Hadoop 上で並列動作する Scala 言語による Sawzall 言語のサブセット処理系を実装した。Hadoop で直接記述した場合と、プログラム量および実行速度の点で比較を行ったところ、プログラム量は大幅に小さくなる一方、実行速度の面でも一定のオーバヘッドがあることが確認された。

今後の課題としては下記が挙げられる.

- **コンパイラの実装** 現在の実装はインタプリタとなっており、実行時のオーバヘッドが大きい。このオーバヘッドを削減すべくコンパイラを実装中である。このコンパイラは Java 言語を中間言語として用いる。出力されたコードは自動的にコンパイルされ、Hadoop の Mapper 上で稼働する。
- **関数定義、型定義の実装** 本研究の目的は Sawzall のクローンを作ることではなく、szl との言語仕様の相違を埋めることには意味はない。とはいえ、関数定義と型定義は言語仕様として妥当だと思われるので今後実装を進める予定である。
- **SSS への適用** 本言語処理系の本来のターゲットは Hadoop ではなく、現在我々が開発中である MapReduce 処理系 SSS である。今回は SSS の処理系が未成熟であったため Hadoop を用いた。SSS のインターフェイスは Hadoop と若干異なるがセマンティクスは類似しているので、容易に適用が可能だとおもわれる。
- **言語の拡張** Sawzall は Map と Reduce のうち、Map しか記述できない。これは言語をシンプルにし、理解を容易にすることに貢献しているが、記述できる問題のクラスを狭める事にもなっている。Sawzall の簡素さを損なわずに、テーブルを拡張し Reduce フェイズを記述する言語機能の追加を検討する。さらに MapReduce を汎化した枠組みを検討し、その枠組に対する Sawzall 言語の拡張を検討していく。

謝　辞

本研究の一部は，独立行政法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務「グリーンネットワーク・システム技術研究開発プロジェクト（グリーン IT プロジェクト）」の成果を活用している.

参 考 文 献

1) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (2004).
2) Ogawa, H., Nakada, H., Takano, R. and Kudoh, T.: SSS: An Implementation of Key-value Store based MapReduce Framework, *Proc. of CloudCom 2010 (Accepted as a paper for MAPRED'2010)*, pp.754–761 (2010).
3) Hadoop, http://hadoop.apache.org/.
4) Protocol Buffers, http://code.google.com/p/protobuf/.
5) SunMicrosystems, I.: XDR: External Data Representation Standard, RFC 1014 (1987).
6) Pike, R., Dorward, S., Griesemer, R. and Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall, *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, Vol.13, No.4, pp.227–298 (2005).
7) Szl, http://code.google.com/p/szl/.
8) Hive, http://hive.apache.org/.
9) Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H. and Murthy, R.: Hive - A Petabyte Scale Data Warehouse Using Hadoop, *ICDE 2010: 26th IEEE International Conference on Data Engineering* (2010).
10) Pig, http://pig.apache.org/.
11) Olston, C., Chopra, S. and Srivastava, U.: Generating Example Data for Dataflow Programs, *SIGMOD 2009* (2009).
12) jaql: Query Language for JavaScript(r) Object Notation, http://code.google.com/p/jaql/.
13) Das, S., Sismanis, Y., Beyer, K.S., Gemulla, R., Haas, P.J. and McPherson, J.: Ricardo: Integrating R and Hadoop (2010).