TIAGO RODRIGO KEPE

KONFIGJOB: A FRAMEWORK BASED IN BACTERIOLOGICAL ALGORITHM FOR HADOOP JOB CONFIGURATION

Master's dissertation presented to the Informatics Graduation Program, Federal University of Paraná.
Advisor: Prof. Dr. Eduardo C. de Almeida

CURITIBA

2013

CONTENTS

RESUMO

Texto do resumo....

ABSTRACT

Currently with the popularity of internet and phenomenon of the social networks a large amount of data is generated day-to-day. To analyse and process such quantity of data the big companies are using MapReduce paradigm. The Hadoop framework implements MapReduce paradigm, it is robust tool that provides a simple interface to implement MapReduce jobs, however for each job there are many knobs to adjust that depents of the data stored and job running. Find a good configuration spends too much time and a configuration found right now may be impracticable of the next time.

In order to facilitate and automate the hadoop job configuration, we propose a framework based on a revolutionary algorithm. Our framework allows to find a good job configuration considering the data stored and the job in question. The users can provide your usuals job configurations and get the new job configuration that will be more appropriate with the current state of data stored and the hadoop cluster. So the users have a tool end-to-end to automate the choice of knobs for each job.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Currently with the popularity of internet and phenomenon of the social networks a large amount of data is generated day-to-day. To analyse and process such quantity of data is needed a big computing power that one single machine could not analyse such data. To solve it the big companies, researchers and governments are using distributed computation. To perform the distributed computation efficiently the data storage must be simple and so to allow parallel processing. A model that has such features is the key-value model and the interface with this model is MapReduce paradigm [5].

MapReduce became the industry de facto standard for parallel processing. Attractive features such as scalability and reliability motivate many large companies such as Facebook, Google, Yahoo and research institutes to adopt this new programming paradigm. Key-value model and MapReduce paradigm are implemented on the framework Hadoop, an open-source implementation of MapReduce, and these organizations rely on Hadoop [10] to process their information. Besides Hadoop, several other implementations are available: Greenplum MapReduce [7], Aster Data [1], Nokia Disco [9], Microsoft Dryad [8], among others.

MapReduce has a simplified programming model, where data processing algorithms are implemented as instances of two higher-order functions: Map and Reduce. All complex issues related to distributed processing, such as scalability, data distribution and reconciliation, concurrence, fault tolerance, etc., are managed by the framework. The main complexity that is left to the developer of a MapReduce-based application (also called a job) lies in the design decisions made to split the application specific algorithm into two higher-order functions. Even if some decisions may result in a functionally correct application, bad design choices might also lead to poor resource usage.

Implement jobs on Hadoop is simple, but there are many of knobs to adjust that depends of the data stored and job running. A good configuration can improve the job performance and one relevant aspect is that the MapReduce jobs work with large amounts of data, such fact is the main barrier to find a good configuration. Therefore a data sample is essencial, but generate a representative and relevant data sampling is hard and a bad sampling may not represent several aspects related to the computation in large-scale: efficient resource usage, correct merge of data, intermediate data, etc.

Hence is very important to adjust the configuration knobs for each job and this configutarion must be specific for own job. However, according with the cluster variation, eg. to add or to remove machines, the data insertion or remotion, may be need to adjust again the job configuration.

Find a good configuration is not so easy and may spend much time. So one way to automate the job configuration is very useful for users.

1.2 Objectives

Our objective is to propose autoconfiguration of Hadoop, for this we intend to use an evolutionary algorithm [2] to select good configurations of the jobs. Based in our knowledge the best way to find such configurations is to run the jobs with its and analyse the performance, but a crucial trouble is the large amount of data stored that can increase exponentially the test time of the job. One way to solve this trouble is to create a data sample. We propose one methodology to implement a data sample using key-value model and MapReduce paradigm.

1.3 Contribution

We present an original approach to automate Hadoop job configuration, our implementation is basead in an bacteriological algorithm [2] and in order to use this algorithm we develop a method to obtain data sample on hadoop cluster. To develop this method we needed to consider a lot of aspects related the paradigm MapReduce, key-value model

and others hadoop particularities. Our framework has an user interface which have been implementing with domain specific language (**DSL**), it's a front end for the users and facilitates the use of the our framework, after ran it the user can obtain the job configuration resultant, so the users have a tool end-to-end.

The work presented here contributes to the establishment a framework to automate Hadoop job configuration, through the following proposals:

- a interface for users basead on domain specific language;
- an algorithm to automate a good choice of jobs configuration;
- a method for sampling data on Hadoop clusters.

As measure of performance we used the latency time that the job led to conclude. Furthermore, we intend to use other measures of performance such as amount of intermidiate data generate, network usage and among others.

1.4 Outline

- Chapter 2 introduces the fundamental concepts of the MapReduce framework.
- Chapter **??** introduces the concepts of the domain specific language.
- Chapter **??** presents the bacteriological algorithm.
- Chapter **??** presents the method to generate sampling data.
- In chapter **??** we presents our framework with all components.
- In chapter **??** we discussed a case study performed with our solution.
- In chapter **??** we conclude our results.

CHAPTER 2

MAPREDUCE

MapReduce [4] is a programming model and a paradigm to build large-scale parallel data processing applications. The programming model is inspired on the *Map* and *Reduce* primitives from functional programming languages. The framework proposes two extensions points (or hooks), whose interface is based on this same two higher-order functions. Users create MapReduce programs (or jobs) by defining the precise behavior of these functions.

During execution, the framework deploys copies of the program across several worker nodes, partitions the input data and schedules the execution across a set of nodes. The framework also handles node failures and the required communication between nodes. After the deployment, the master selects idle workers to assign a map or a reduce instance and orchestrates their execution. The data flow between the map and the reduce functions is shown in Figure 2.1.

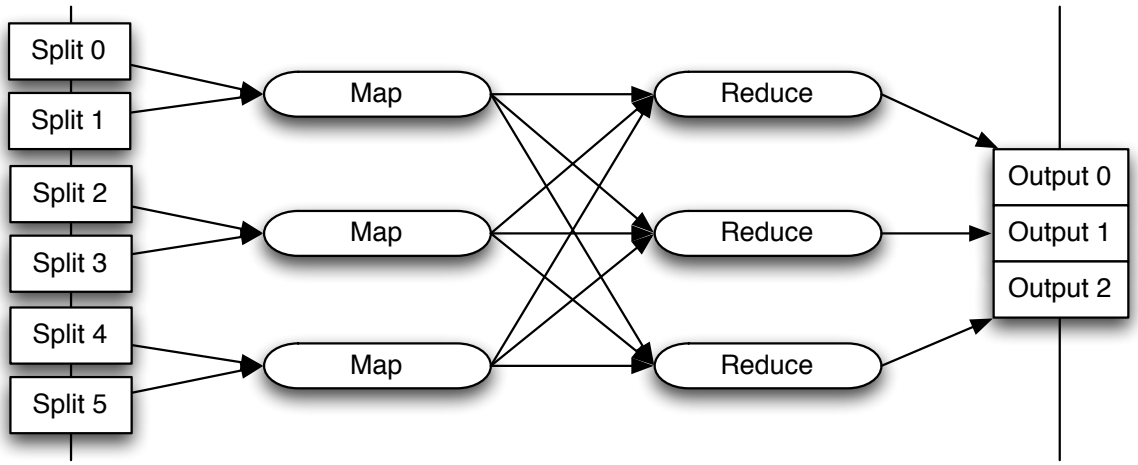

Figure 2.1: Execution of Map and Reduce operations

The input data set is divided into several *splits*. Each split is assigned to a map task and executed on a node. The result of this processing is a set of intermediate keys and

associated values. Reduce tasks take the results from map tasks to produce the final result. When all the reduce instances terminate, they append their result to the final output file.

The whole processing is based on $\langle key, value \rangle$ pairs. The Map function groups together all input values $v1$ associated to the same key $k1$ into an intermediate result set of keys and values $\langle k2, v2 \rangle$. These values are passed to the Reduce function that combines them into a reduced set:

map	$k1, v1$	$\rightarrow$	$set(k2, v2)$
reduce	$k2, set(v2)$	$\rightarrow$	$set(v2)$

Eventually, the data flow may be completed with a *Combiner*, a local reduce function used for bandwidth optimization. This function runs after the Mapper and before the Reducer and is run on every node that has run map functions. The Combiner may be seen as a *mini-reduce* function, which operates only on data generated by one machine.

A canonical example of a MapReduce job is the Word Count application, which has as an input several textual documents and as an output a set of pairs $\langle Key, Value \rangle$, where each key is a different word and the value is the number of occurrences of the word on all input documents. The responsibility of of the Mapper is to separate the text into a set of words and that of the Reducer is to aggregate matchings words and count the number of occurrences. In this example, the function of the Combiner is almost identical to that of the Reducer. The main difference is that combiners are executed locally to each node, immediately after each mapper, and use local data, while reducers execute on different nodes and use data that comes from different mappers.

The Java implementation of the map function is presented in Listing 1. The **map()** method has three parameters: **key**, which is never used; **value**, which contains the text to be processed; and **context**, which will receive the output pairs. The body of the method uses the class **StringTokenizer** to break the input text into tokens and then, for each token, writes a pair containing the token and the number 1. The map does not count words, if there are several occurrences of a word in the input, there will also be several occurrences of it in the output.

```java
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

Listing 1: Class TokenizerMapper

The implementation of the reduce function is presented in Listing 2. The **reduce()** method has also three parameters: **key**, which contains a single word; **values**, a set containing all values associated to the key (i.e. the word); and **context**, the output. The behavior of the method is quite simple, it sums all values associated to the key and then writes a pair containing the same key and the calculated amount.

```java
public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    private IntWritable result = new IntWritable();

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

        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Listing 2: Class IntSumReducer

Eventually, the reduce function could also be used as a combiner, to locally aggregate different occurrences of the same word. The choice of using or not a combiner, as well as the number of reducer instances, is not automatic, it is left to developer. Still, this choice may have an important impact in both, the correctness and the efficiency of the job. An example of the inputs and the outputs of both functions when applied to a simple sentence is presented in Table 2.1.

map	"Never Say Never Again"		$\rightarrow$	$\langle Never, 1\rangle$,$\langle Say, 1\rangle$, $\langle Never, 1\rangle$, $\langle Again, 1\rangle$
reduce	$\langle Never, \{1, 1\}\rangle$, $\langle Again, \{1\}\rangle$	$\langle Say, \{1\}\rangle$,	$\rightarrow$	$\langle Never, 2\rangle$,$\langle Say, 1\rangle$, $\langle Again, 1\rangle$

Table 2.1: Word count example

CHAPTER 3

DOMAIN-SPCIFIC LANGUAGE

A *domain-specific language* (DSL) is way to approach of some specific context through appropriate notations and abstractions [6]. DSL transforms a particular problem domain into a context intelligible for expert users that can work in a familiar environment.

Problem domain is a crucial term of DSL that requires prior background of the developers in the specific context, so the developers must be expert in the domain in order to develop DSLs that cover all features required for the users. There are a lot of examples of DSLs in differents domains, **(LEX, YACC, Make, SQL, HTML, CSS, LATEX, etc.)** are classical examples of DSLs [3].

DSLs are usually focused in its domains containg notations and specific abstractions, normally DSLs are *small* and *declarative* languages. However, a DSL can be extended to others domains, in this case such DSL is general-purpose language (GPL), because its expressive power is not restricts an exclusive domain, examples of such DSLs are **Cobol and Fortran**, which could be viewed as languages focused towards the domain of business and scientific programming [6], respectively, but they are not restricts just in this domains.

DSL are used in several big areas, such **Software Engineering**, **Artificial Intelligence**, **Computers Architecture**(in this area a good exemple is VHSIC Hardware Description Language (VHDL), where VHSIC mean **V**ery **H**igh **S**peed **I**ntegrated **C**ircuits), **Database Systems**(SQL is a classical example already cited), **Network**(where its protocols are examples of DSLs), **Distribuited Systems**, **Multi-Media** and among others. A current area that have been emerged recently is **Big Data** , this area may be considered as a sub area of Database, but is has many particularities that involve a mix features of Database and Distributed Systems.

BIBLIOGRAPHY

[1] Inc. Aster Data Systems. In-database mapreduce for rich analytics.

[2] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, e Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Softw. Test. Verif. Reliab.*, 15:73–96, June de 2005.

[3] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(1):711–721, August de 1986.

[4] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[5] Jeffrey Dean, Sanjay Ghemawat, e Google Inc. Mapreduce: simplified data processing on large clusters. *In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. USENIX Association, 2004.

[6] Arie Van Deursen, Paul Klint, e Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN NOTICES*, 35:26–36, 2000.

[7] Greenplum. A unified engine for rdbms and mapreduce, 2008.

[8] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, e Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, 2007.

[9] Prashanth Mundkur, Ville Tuulos, e Jared Flatow. Disco: a computing platform for large-scale data analytics. *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, New York, NY, USA, 2011. ACM.

[10] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1 edition, july de 2009.

TIAGO RODRIGO KEPE

KONFIGJOB: A FRAMEWORK BASED IN BACTERIOLOGICAL ALGORITHM FOR HADOOP JOB CONFIGURATION

Dissertation presented as partial requisite to obtain the Master's degree. M.Sc. program in Informatics, Federal University of Paraná. Advisor: Prof. Dr. Eduardo C. de Almeida

CURITIBA

2013