TIAGO RODRIGO KEPE

SELF-TUNING BASED ON DATA SAMPLING

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

CURITIBA

2013

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

BA - Bacteriological Algorithm

DW - Data Warehouse

ELT - Extract Load Transform

ETL - Extract Transform Load

GA - Genetic Algorithm

MR - MapReduce

RESUMO

Atualemente, com a popularidade da internet e o fenômeno das redes sociais uma grande quantidade de dados é gerada dia à dia. MapReduce aparece como um poderoso paradigma para analisar e processar tal quantidade de dados. O arcabouço Hadoop implementa o MapReduce paradigma, no qual uma simples interface está disponível para implementar programas MapReduce(MR). Entretando, em programas MR desenvolvedores podem configurar vários paramêtros para otimizar a performance dos recursos disponíveis, mas encontrar boas configurações consome tempo e uma configuração encontrada em uma execução pode ser impraticável na próxima vez.

A fim de facilitar e automatizar o ajuste de programas do hadoop, nós propomos um auto-ajuste baseado em amostragem de dados. Nossa abordagem permite uma boa configuração considerando os dados armazenados e o programa em questão. Usuários até podem fornecer suas usuais configurações do programa e então obter uma nova configuração que será mais apropriada com o estado atual dos dados armazenados e com o cluster do hadoop. Então os usuários tem uma ferramenta de ponta-a-ponta para automatizar a escolha de paramêtros para cada programa.

ABSTRACT

Currently with the popularity of Internet and phenomenon of the social networks a large amount of data is generated daily. MapReduce appears as a powerful paradigm to analyse and process such amount of data. The Hadoop framework implements the MapReduce paradigm, in which a simple interface is available to implement MapReduce jobs. However, in MR jobs developers are allowed to setup several parameters to draw optimal performance from the available resources, but finding a configuration which best suits to the current state of the cluster and the data stored is time consuming and a configuration found in an execution may be impracticable for the next time.

Hadoop cluster administration involve, beyond other tasks, choosing configurations for each job. In Big Data environments this task is impracticable to be executed manually because of the huge set of jobs. In order to facilitate and automate tuning Hadoop jobs, we propose a self-tuning based on data sampling. Our approach allows to find a job configuration according to the cluster state and the data stored. Users can provide their usual job configurations then get the new job configuration that will be more appropriate with the Hadoop current state and the data stored. So the users have an end-to-end tool to automate the choice of knobs for each job.

CHAPTER 1

INTRODUCTION

In this chapter we present our motivation and objectives for this work, and we present the organization of the document.

1.1 Motivation

Nowadays big companies are processing and daily generating a vast amount of data. These companies are growing investing in distributed and parallel computing to process such data. To perform the distributed computing efficiently the data storage must be simple in order to allow parallel processing. The key-value model is a potential solution to enable the applications enjoy the data parallelism, e.g. the MapReduce (MR) programing paradigm which is based on key-value model [11].

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 MR paradigm are implemented on the framework Hadoop, an open-source implementation of MapReduce, and these organizations rely on Hadoop [32] to process their information. Besides Hadoop, several other implementations are available: Greenplum MapReduce [27, 16], Aster Data [3], Nokia Disco [26] and Microsoft Dryad [21].

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 and fault tolerance 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 the two higher-order functions. Even if some decisions may result in a functionally correct

application, bad design choices might also lead to poor performance.

Implement jobs on Hadoop is simple, but there are many knobs to adjust depending on the available resources (e.g. input data, online machines, network bandwidth, etc.) that improve the job performance. One relevant aspect is that the MR jobs are expected to work with large amounts of data, which can be the main barrier to find a configuration [6] that adapts to the current cluster state, such configuration we call of adaptive configuration. Therefore, data sampling can be useful to improve the testing time of new configuration parameters, instead of processing all data set how is done in [19]. 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 and intermediate data.

1.2 Objectives

Our objective is to propose a self-tuning based on data sampling over Hadoop, so we intend to use an evolutionary algorithm [4] to select adaptive configurations for MR jobs. Based on our knowledge the best way to test the job with such configurations is to run it and analyse the response time, normally this process is done manually. But a crucial trouble is the large amount of data stored that can exponentially increase the time needed to test the job. One way to solve this trouble is to create a data sample. We propose one method to implement data sampling distributed using key-value model and MR paradigm.

1.3 Contribution

We present an original approach to automate Hadoop job configuration. Our approach is based on an bacteriological algorithm [4] in order not to run it on all data storage we develop a distributed method to obtain data samples in Big Data environment. For data sampling we considered a lot of aspects related the paradigm MR, key-value model and others hadoop particularities. Our approach consists of a user interface to facilitate the

user iteraction with data sampling and drive tuning procedures through a domain-specific language (DSL).

Our proposal intents to establish a framework to automate Hadoop job configuration, through the following proposals:

- an algorithm to automate a self-tuning;
- a method for sampling data on Hadoop clusters;
- an interface for users based on domain specific language;

As measure of performance, we used the job response time. Furthermore, we intend to use other measures of performance such amount of intermidiate data, network usage and CPU usage.

1.4 Outline

The Chapter 2 introduces the fundamental concepts of the key-value model, MR paradigm and Hadoop framework. Chapter 3 presents the bacteriological algorithm to choose job configurations. Chapter 4 presents the distributed method to generate data sampling. Chapter 5 introduces the concepts of the domain-specific language and our proposal DSL. Chapter 6 presents our initial implementation with all components. Chapter 7 discuss a case study performed with our solution. Chapter 8 conclude our results.

CHAPTER 2

KEY-VALUE MODEL, MAPREDUCE AND HADOOP

This chapter introduces some concepts that are used in the subsequent sections: the key-value model, the MapReduce paradigm and the Hadoop framework.

2.1 Key-value model

The key-value model is a simple model for data storage, such simplicity is bound by the lack of data schema previously known. It is based on one linked pair: the key and the value. Generally, the pair is stored without any aggregation or creation of data schema, so the key-value model stores data of unstructured way. Thus all detailing of data is done at runtime. Unlike other models, such as the relational model [8], in which the simplistic notion of tables, attributes and relations define the data structure for the storage. Another similar example is the hierarchical data model [29], in which the links that connect the records define a data structure. Hence, those models and among other store structured data intrinsically.

Associated with the relational model was created the Data Warehouse (DW) technology, which aims to solve some issues involving the data structure of such model. It is a repository that aggregates data from several sources [29], through the Extract Transform Load (ETL) technique: data is extracted from sources, transformed and load in the DW.

Due to the simple storage of the key-value model, data *transformation* is done in the last phase, so the transformation occurs after the data has been loaded into the target database. Thus there is a inversion of ETL to ELT(Extract Load Transform) . This inversion causes one issue to process large amounts of data, requiring much computing power while querying the data. One programming paradigm that support ELT and handles the key-value model is the MR that is presented in the next section.

2.2 MapReduce

MapReduce is a programming system that allows many processes of one database to be written in simple way, [22]. Vast amount of data is splitted and assigned to a set of computers, called computers cluster to improve performance through parallelism. The goal is to reduce the complexity, so the users can focus on the main problem that is the data processing.

The paradigm is inspired on the high-level **Map** and **Reduce** primitives from functional programming languages. Hence the programmers can focus only on the creation of the two higher-order functions to solve a specific problem and to generate the necessary data.

According to [10]: [*"the computation takes a set of input key/value pairs, and produces a set of output key/value pairs."*]. A user writes the map function that receives a set of key/value pairs and produces an *intermidiate* set of key/value pairs. The reduce function receives the intermidiate pairs as input and produces the *resultant* set of key/value pairs. This process is shown in Figure 2.1:

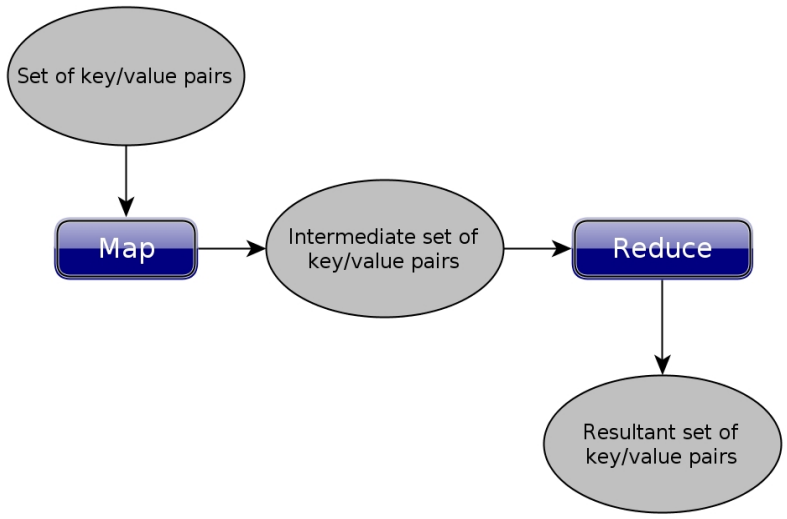

Figure 2.1: Map and Reduce process.

2.3 Hadoop

Map and reduce functions are present in Lisp and others functional languages. Recently the MapReduce paradigm have been implemented by several frameworks such as Greenplum MapReduce [27, 16], Aster Data [3], Nokia Disco [26], Microsoft Dryad [21], and the one open-source implemantation from Apache: Hadoop [20].

The Hadoop is a framework for reliable, scalable and distributed computing. It provides an interface to implement the map and reduce functions in high-level which are called map and reduce tasks. It was designed to allow users to focus on the implementation those functions, without worrying about the issues involving the distributed computing. All aspects involving the distributed computing and storage are left to the framework such as split files, replication, fault tolerance and distribution of tasks.

There are two main components on Hadoop:

- Hadoop Distributed File System (HDFS);
- Engine of MapReduce.

The HDFS stores all files in blocks, the block size is configurable per file, all blocks of one file have the same block size except the last block. It is divided into two components the ***NameNode*** and the ***DataNode***. The NameNode is placed in one master machine, it stores all the metedata and manages all the DataNodes. The DataNode stores data, when one DataNode starts it connects to the NameNode, then responds to requests from the NameNode for file system operations.

The engine of MapReduce is responsible for parallel processing. It is constituted by one master machine and slave machines, also called workers. The master designates which slaves will receive map and reduce tasks with its respective input blocks. The worker that receives map task is called mapper and the slave that receives reduce task is called reducer.

2.3.1 Job processing

A job is a program in a high-level language such as Java, Ruby or Python that implements the map and reduce functions. Initially the master machine receives jobs with the

relative input directory in the HDFS which contains all the files to be processed (inserted previously in the HDFS). Then the master requests to the NameNode infomation about the blocks and file locations, after that it deploys copies of the job across several workers.

With the blocks information the map task is scheduled to a set of workers with its respective input blocks. So the mappers process each input blocks, generate key/value intermediate pairs and append its in intermediate files. When the mapper instance terminate it notifies the master. The master split the intermediate files in blocks and shuffled them to the reducers to process. When all reducers intances terminate processing, they append their result to the final output file. The data flow between mappers and reducers are shown in Figure 2.2.

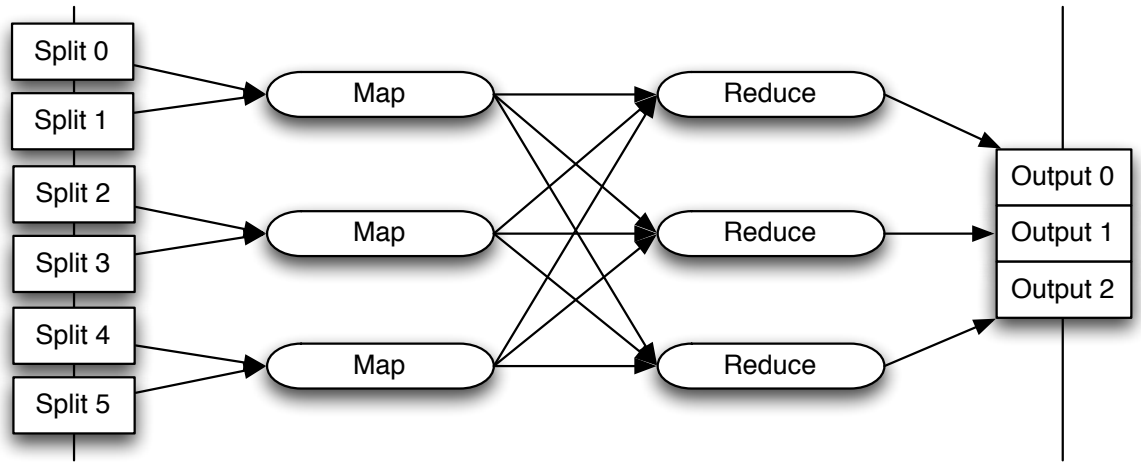

Figure 2.2: Execution of Map and Reduce operations.

2.3.2 MapReduce programing

The whole processing is based on $\langle key, value \rangle$ pairs. The mappers receive the file blocks, call the map function and pass the line number as key and the line as the value, so the pair "line number/line content" is the $\langle k1, v1 \rangle$. The map generates the intermediate result set of key and values $\langle set(k2, v2) \rangle$, when the mappers finished all values for *k2* are grouped in a list and the respective pair $\langle k2, list(v2) \rangle$ is generated. These pairs are sorted and passed as input for reducers that generate the result set:

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

Eventually, when the map results are already available in memory, a local reduce function *Combiner* is used for optimization reasons. Then all values for a given key are combined, resulting in a local set $\langle k2, list(v2)\rangle$. This function runs after the Map and before the Reduce functions on every node that run map functions. The Combiner may be seen as a *mini-reduce* function, which operates only on data generated by one machine.

A well known example of a MapReduce job is the Grep application listing 1, which receives as an input several textual documents and as an output a set of pairs $\langle Key, Value\rangle$, where each key is a different pattern found and the value is the number of occurrences of the pattern in the files. The responsibility of the Mapper is to find pattern in the files. The reducer is responsible for summing the number of occurrence for each patterns.

The **map()** method has four parameters: **key**; **value**, one line that contains the text to be processed; the **output**, which will receive the output pairs and **reporter** for debug. The body of the method uses the class **Pattern** to describe a desired pattern, the class **Matcher** to find this pattern, when pattern are found the pair $\langle matching, 1\rangle$ is emited to output.

```
public class RegexMapper<K> extends MapReduceBase
                        implements Mapper<K, Text, Text, LongWritable> {

    private Pattern pattern;
    private int group;

    public void configure(JobConf job)
    {
        pattern = Pattern.compile(job.get("mapred.mapper.regex"));
    }

    public void map(K key, Text value, OutputCollector<Text, LongWritable> output,
                                        Reporter reporter) throws IOException {
        String text = value.toString();
        Matcher matcher = pattern.matcher(text);
        while (matcher.find())
        {
            output.collect(new Text(matcher.group()), new LongWritable(1));
        }
    }
}
```

Listing 1: Class RegexMapper packed in Hadoop [20].

The implementation of the reduce function is presented in Listing 2. The **reduce()** method has also four parameters: **key**, which contains a single matching string; **values**, a set containing all values associated to the key (i.e. the matching); **output pair**, the resultant pair $\langle matching, total \rangle$ and **reporter** for debug. The behavior of the method is straightforward, it sums all values associated to the key and then writes a pair containing the same key and the total of matching found.

```
public class LongSumReducer<K> extends MapReduceBase
                        implements Reducer<K, LongWritable, K, LongWritable> {

    public void reduce(K key, Iterator<LongWritable> values,
                       OutputCollector<K, LongWritable> output, Reporter reporter)
                throws IOException {

    // sum all values for this key
    long sum = 0;
    while (values.hasNext())
    {
        sum += values.next().get();
    }

    // output sum
    output.collect(key, new LongWritable(sum));
  }

}
```

Listing 2: Class LongSumReducer packed in Hadoop [20].

An example of the inputs and the outputs of both functions when applied to a simple sentence is presented in Table 2.1. We applied the following regular expression: **"[a-z]∗o[a-z]∗"**, this expression finds the words that contains the vowel **o**.

map	"Test for hadoop regular expression inside hadoop"		→	$\langle for, 1\rangle$,$\langle hadoop, 1\rangle$, $\langle expression, 1\rangle$, $\langle hadoop, 1\rangle$
reduce	$\langle for, \{1\}\rangle$, $\langle expression, \{1\}\rangle$	$\langle hadoop, \{1, 1\}\rangle$,	→	$\langle for, 1\rangle$,$\langle hadoop, 2\rangle$, $\langle expression, 1\rangle$

Table 2.1: Regular expression example.

CHAPTER 3

ALGORITHM FOR TEST

In this chapter we present the bacteriological algorithm used to generate and select the job configurations for Hadoop.

3.1 Genetic Algorithm

Evolutionary Algorithms are inspired of biological evolution process to select the best inviduals that adapt themselves in the environment. For this adptation are used some biological mechanisms such as reproduction, mutation, recombination or crossover and **selection**. One of the most known evolutionary algorithms is the Genetic Algorithm (GA).

On the context of GA there are three important components:

- **Gene** that is the smallest particle.
- **Individual** that is composed of genes.
- **Population** that is composed of individuals.

The GA works at the gene level, so all changes are done at this level. At first glance, changes done on gene seems tiny and irrelevant, but these chnages can be crucial for adaptation of the individual in the environment. Genetic changes can be crucial for the survival of one entire population or even mean survival of a specie.

The GA process describes in 1 is based on tree biological mechanisms: reproduction, crossover and mutation which are further detailed below:

- **Reproduction**: copies the individuals to participe of the next stage (the crossover). They are chosen based on their ability to adapt the environment. Those abilities can be calculated according with a function *F(x)* called the fitness of the individual.

The choice of one individual is based in its fitness value. The choice process is similar to spin a roulette wheel where each individual receive slots according with its fitness, e.g. if an individual has F(X) = 10, then it has 10 slots in the roulette and suppose it has 100 slots, so the chance to choose this individual to participate in crossover is *100/F(X) = 1/10*. Thus the individual fitness is greater, then its number of copies tends to be greater.

- **Crossover**: the crossover is similar to the natural process called chromosomal crossover. This process is based on genetic recombination of chromosomes to produce new genetic combinations. Basically the genes of two individuals are genetically combined to generate another resultant individual, so the new individual has some characteristics of both parents. More precisely, in the genetic algorithm two individuals are chosen randomly (A, B), an integer k, between 0 and the size n of an individual minus one, is chosen randomly. The new individual A' is composed by the first k genes of A and the last k - n genes of B. The individual B' consists of the first k genes of B and the last k - n genes of A.

- **Mutation**: is occurs after the crossover. One mutation occurs in the genes of new individuals. The natural process consists basically in change enzymes or proteins of genes in order to create new individuals. The mutation process of GA is simple, one or more genes are randomly selected and then are changed (e.g. change one or more nucleotides of the DNA of one chromosome).

The algorithm starts with an initial population, for each individual is calculated its fitness that is the base for reproduction mechanism, so the three biological mechanisms are called in the specific order already detailed and so the resultant population is evaluated as one or more criteria, if necessary the three mechanisms are run again and the process continues until the criteria is achieved.

Algorithm 1: Genetic Algorithm

Input : *Pop* Initial population
Output: *BestIndiv* The best individual reached
repeat
  for *each* *indiv* ∈ *Pop* **do**
    CalcFitness(*indiv*)
  Reproduction(*Pop*)
  Crossover(*Pop*)
  Mutation(*Pop*)
until *X number of generations* ∨ *a give fitness value reached* ∨ ... ;
BestIndiv ← *getBestIndividual*(Pop)
return *BestIndiv*

3.2 Bacteriological Algorithm

The Bacteriological Algorithm (BA) is part of the family of genetic algorithms that works on genetic context. A minor particle of this algorithm is a gene that influences all phases of the algorithn. A group of genes forms an individual that have more representativeness than an gene and a group of individuals forms one population.

Compared to GA the BA, the individual is a bacteria and the focus of the algorithm is to adapt in a given environment. The BA has some peculiarities to improve some issues involving the GA and change it behavior.

The BAs introduces a new mechanism called memorization that is responsible for memorizing the best individuals created along the generations. As described in [4], it was proposed to improve the convergence of the GA, the introduction of the new mechanism might appear a small modification, but actually reflects a crucial change on GAs.

Besides, of the introduction of the new mechanism, the crossover mechanism was removed because of the peculiar biological behavior of the bacterium. This mechanism cannot be used anymore, in terms of natural bacteriologic process the remotion of the crossover makes sense, the bacterium reproduce itself asexually, consequently there is not crossover between two individuals, because the reproduction process consists in duplicating the DNA of a bacterium and after a division to form two new bacteria.

The algorithm in high-level of abstraction is described in 2. The BA is started and has four main mechanisms: Fitness computation, Memorization, Reproduction and Mutation

which are detailed below:

- **Fitness computation**: the fitness as in GA is one way to differentiate the abilities of each individual to adapt to the environment. Calculation depends on several criteria defined by the programmer and is used to select the best individuals for the next generation.

- **Memorization**: is the main mechanism introduced by the BAs. Its is responsible for memorizing the best individuals generated by the process of adaptation, as the process continues, the population improve more quickly its capacity of adaptation. The process consists in memorizing the best individuals through the generations, if one generation generates bad individuals, i.e. generate low fitness values, then the memorization operator ignores this generation and uses the best individuals from past generations to the next generation in order to avoid regressions in the process.

- **Reproduction**: is similar to GA, the best individuals are sorted randomly and selected to the mutation process. One drawback in this stage is the population size may grow up exponentially, so thresholds must be established.

- **Mutation**: is responsible for generating new individuals, one or several genes are changed in order to improve the adaptation of the bacteria population to the environment. These new individuals are evaluated by their fitness and they may be inserted in the set of best individuals.

Algorithm 2: Bacteriological Algotithm

Input : Pop Initial population
Output: $BestIndiv$ The best individual reached
$SetBestIndiv \leftarrow \{\}$
repeat
  for *each* $indiv \in Pop$ **do**
    CalcFitness($indiv$)
  SetBestIndiv.pop(Memorization(Pop))
  Reproduction(Pop)
  Mutation(Pop)
until *X number of generations* $\vee$ *a give fitness value reached* $\vee$ *...* ;
$BestIndiv \leftarrow getBestIndividual$(SetBestIndiv)
return $BestIndiv$

CHAPTER 4

SAMPLING ON HADOOP

In this chapter we present a method for data sampling on Hadoop that is basad in a random algorithm.

4.1 Motivation for sampling

One important aspect in Big Data environments is the vast amount of data, that is the main barrier to find a job configuration which best suits to the current state of the cluster [6]. BAs allow to create new configurations, but these configurations must be tested in order to select which is the best for the job at a given moment. Moreover, the volatility of the computing nodes (joining and leaving), big cluster setups at any time and data volatility, may spoil performance depending on the configurations setup.

Therefore, caused by the volatility of the cluster and the data, the job configuration may become deprecated and needs to be reviewed constantly. The review process consists in choosing a new configuration and testing it in order to analyse the performance. But the testing cannot be done on the entire data because may be time consuming.

The main question is: *How to test job configurations generated by BAs?* One possible answer is to run the job on data sampling, because the sampling in a database is a essential step to improve the response time, moreover run the job on all data stored would spend a lot of time and also would be impracticable due to larger number of intermediate configurations generated by the algorithm.

4.2 Challenge for data sampling in Big Data environments

On Big Data environments there are several aspects involving distributed computing and storage. For data sampling the both aspects are relevants and need to be considered.

In this context, the volume of data is the main issue because the data sampling must be distributed too, otherwise one machine couldn't bear all data storage in the cluster then make the data sampling. For instance, suppose that one cluster stores 100 terabytes and we want data sampling of 20%, then each node must sample and store locally. The resulting sample will be 20 terabytes: which one single machine may not have such storage capacity.

So the resulting data sample must be distributed in the cluster, because it might be too big to be store on a single machine. After the sample is obtained, jobs can be run on the sampling. So, as the example would avoid to run the job on 100 terabytes and run just 20 terabytes.

The Hadoop has the structure for distributed computing and storage, so a way to produce a sample is to utilize the benefits of Hadoop, i.e. taking into consideration that the data are already distributed. As a consequence, we can build one MapReduce program to sample data and then benefit from its advantages.

One of the most used data sample techniques is Random Sampling that consists in selecting a pre-determined amount of data randomly [28]. In the literature there are several others techniques such as Stratified Random Sampling, which splits data in strata where each element has the same chance of being selected [9]. Another thechnique is Systematic Sampling that consists in select randomly one element of the population, from this element the next k elements are select in sequence, the number k may be choosen randomly or based in some criteria. The systematic process continues till the sample is completed [15].

In the context of big data there are some implementations of data sample. One example is MonetDB which is a column-oriented database management system designed to hold data in main-memory and processing distributed of large-scale data [25]. This database supports data sample and uses the *Algorithm A* which is based on a random sample method [24].

The *Algorithm A* select n records from a file containing N records where $0 \leq n \leq N$. For each record that will be inserted in the sample, it chooses randomly one number V

that is uniformly distributed between 0 and 1. Based on *V, n* and *N*, the number *s* is calculated. The set of records *S* is then created from *s*. This set contains the *(s + 1)* first records of the file. Then one record is chosen from *S* and added to the sample. The records present in *S* are skipped in the next interaction [31].

The *Algorithm A* behaves as the stratified random sampling technique. The creation of the set S can translate to one stratum that contains neighbors records which one record is chosen.

Hive is another database management system that performs data sampling based on random methods. Hive was created to manage the data stored on Hadoop, allowing ad-hoc queries (with are casts to Hadoop MR jobs), data summarization and analysis of large datasets. Thus, Hive is considered a DW for Hadoop [2]. It samples at row or block size level. The row level consists in choosing randomly the rows according with the colunm name. If the column name is not defined, then the entire row is selected. If it is defined the choice can be done using the Bucketized Table in which the sample is done only on the buckets that contains the specified column [1]. The block size sample is also done ramdomly and consists in selecting the blocks that match with the specified block size.

Those sample methods on Hive are based on random sample and handle structured data. Hive consists storing the Hadoop data as a data warehouse and facilitate queries submitted by users. Moreover, the clustering by bucket and block size requires a prior structuring of data, so in the Hive several information about the data are previously known.

In Hadoop, data are stored in a unstructured manner and this characteristic is the biggest challenge to develop data sampling methods. According to [30, 7, 18, 33] the challenge with unstructured data stream can be addressed with Reservoir Sampling: "*Say you have a stream of items of large and unknown length that we can only iterate over once. Create an algorithm that randomly chooses an item from this stream such that each item is equally likely to be selected.*"

The Reservoir Sampling algorithm is also a random algorithm. It consists in randomly choosing *k* elements from a list *L* containing *N* items. The length N is either unknown or

too large to fit in memory.

To understand the algorithm one example can clarify the idea. Suppose we have one reservoir sampling of size equal 1 and we have to get one item such that all items have the same probability to be choosen.

In the first round when the first item cames the probability is $P(1)^1 = 1$ because the stream length is 1 at the moment and we unknow if the stream finished. When the next element cames (second element), the first element has been holding and we need to choose continuing hold the first or choosing the second element. So, the probability to choose the second element is $P(2) = \frac{1}{2}$ because the stream length is 2 at the moment. On the other hand, the probability $P(1)^2$ continuing hold the first element is the probability to choose it in the last round multiple by the probability doesn't choosing the second element, which is $P(1)^2 = P(1)^1 \times \overline{P(2)} = 1 \times (1 - P(2)) = 1 \times \left(1 - \frac{1}{2}\right) = 1 \times \frac{1}{2} = \frac{1}{2}$. So, the probability of the first and the second element in the second round is $\frac{1}{2}$.

In the next round when the third element cames, we need to decide if continue holding the element chosen in the last round or if we choose the third element. The probability to choose the third element is $P(3) = \frac{1}{3}$ because the stream length at the moment is 3. Now, we need to calculate the probability to continue holding the element chosen in the last round, considering which is the first element, its probatility in the third round is the probability of it in the second round multiple by the probability doesn't choosing the third element: $P(1)^3 = P(1)^2 \times \overline{P(3)} = \frac{1}{2} \times (1 - P(3)) = \frac{1}{2} \times \left(1 - \frac{1}{3}\right) = \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$. So, the probability of the first, second and third element in the third round is the same, i.e. $\frac{1}{3}$.

In sequence, for the Nth round the probability of all elements is $1/N$. We can prove such idea by induction:

- *Base Case*: $N = 1$.

  The probability for the first element is $P(1) = \frac{1}{N} = 1$.

- *Induction Step*: $N > 1$.

  By the induction hypothesis, the probability for all elements in the round N: $P(1)\ ...\ P(N) = \frac{1}{N}$.

  In the round $N+1$, when the $N+1$ element cames its probability to be chosen is $P(N+1) = \frac{1}{N+1}$. Because the stream length is $N+1$ at the moment.

  We need to prove that the probability of the all other elements is the same. Let's prove the probability of the elements 1 to N in the round $N+1$:

  $$P(1\ to\ N) = \text{(the probability of the all elements in the round } N\text{)} \times \text{(the probability doesn't choose the element } N+1\text{)}$$
  $$P(1\ to\ N) = P(1\ to\ N)^n \times \overline{P(N+1)}$$
  $$P(1\ to\ N) = \frac{1}{N} \times \left(1 - \frac{1}{N+1}\right)$$
  $$P(1\ to\ N) = \frac{1}{N} \times \frac{N}{N+1}$$
  $$P(1\ to\ N) = \frac{1}{N+1}$$

  So, the probability of all elements in the round $N+1$ is $\frac{1}{N+1}$.

This prove can be generalized to the reservoir of the length K. So, an implementation of the reservoir algorithm is presented in algorithm 3. The goal is to build a reservoir which is smaller than the memory. So it receives as parameter the number K that is the resulting sampling size and *stream* of data that constantly receives new data. Initially the resultant sampling is assigned with the first $K elements$, so the algorithm aims to

calculate the probability of the next element, (e.g. $K+1$), to be inserted to the sample, which is $P(K+1) = \frac{K}{K+1}$, after one random number ($randNumber$) between 0 and 1 is chosen, if $randNumber < P(K+1)$, then the next element is added to random position in the resultant sample.

Algorithm 3: Algorithm for Reservoir Sampling

```
Input  : k size of sample
Input  : stream data stream with indefided length
Output: arraySample[k]
for i = 1 → k do
    arraySample[i] ← stream[i]
nextElement ← k
while stream != EOF do
    nextElement ← nextElement + 1
    probability ← k/nextElement
    randNumber ← Random(0, 1)
    if randNumber < probability then
        pos ← Random(1, k)
        arraySample[pos] ← stream[nextElement]
return arraySample
```

However, choosing the number K is hard task, because the resultant sample must be representative and fit in memory. To solve this problem Vitter [30] suggests to store the reservoir on secundary storage (hard disk). This approach may be inapplicable for Hadoop, because the secundary storage is HDFS and the time needed to retrieve the reservoir and update is long, caused by the communication overload in contact the namenode, look up the reservoir in the cluster and make the reservoir available for the algorithm in the current round.

4.3 One method for data sampling on Hadoop

We propose the follow algorithm based on the MR paradigm to generate data samples on Hadoop. We intend to enjoy the distributed architecture of Hadoop, so we can classify each line by the mappers and perform the sample distributed using the reducers. Our approach follows the random algorithm family which has been used widely in database and big data environments, and it performs the sampling in row level.

First of all, we present the behavior of the algorithm on map and reduce process using the Figure 4.1. Initially the program receives tree files: *file1, file2 and file3*. There are two mappers to classify each line of the files and generate the output $\langle lineNumer, content \rangle$. Then, the shuffle step aggregates each content of the same key: $\langle lineNumber, \{content1, content2, ..., contentN\} \rangle$.

The only reducer receives the shuffle output and, for each content of the same key, chooses one random number. If it is less than sample threshold then chooses this content, otherwise discards it.

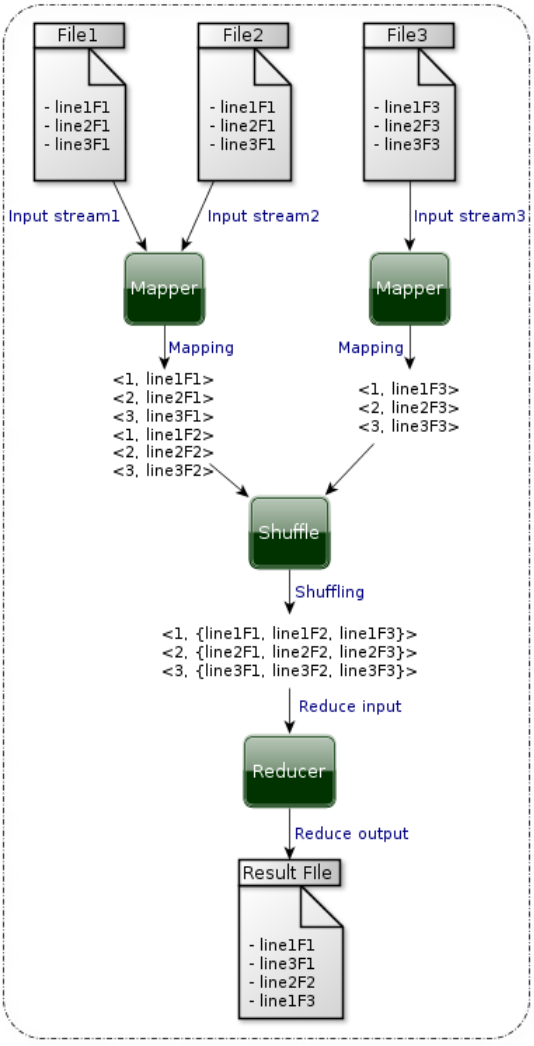

Figure 4.1: Map and Reduce random sample process

The map function is described by algorithm 4. Before presenting the algorithm, some previous definitions are necessary. Let us denote F the set of files stored on the cluster, f

a file belonging to F, l current line of f and o the order of the L in F.

Algorithm 4: Map function for data sample

```
Input  : F file set in the cluster
Output: map < Integer, String > resultant list of key-value
map ← {}
for each f ∈ F do
    l ← f.getNextLine()
    o ← l.getOrder()
    map.put(o, l)
return map
```

The map function consists in classifying each L in the F with it respective order o. Then intermediate key-value pair $\langle o, l \rangle$ are emitted. Next, each pair is added to a map structure that is the output of the map function. After the shuffle phase aggregates values sharing the same key. These aggregated values are the input for reduce function.

Algorithm 5: Reduce function for data sample

```
Input  : mapList < key, values > list of key-values aggregated by shuffle phase
Output: list of selected values
for each key ∈ mapList do
    for each v ∈ values do
        rand ← random(1, n)
        if rand ≤ n then
            list.add(v)
return list
```

The reduce function is in charge of the sampling and is described by algorithm 5. Before presenting the algorithm, some previous definitions are necessary. Let us denote *mapList* the intermediate set generated by map phase and aggregated for the shuffle phase, it contains tuples $\langle key, list < values > \rangle$. The *key* is a key belonging to the *mapList*, and *values* is a set of values sharing the same key. The v is a value belonging to *values* and n is the amount of the values sharing the same key.

First the reduce algorithm iterates in each *key* and get the *values* list. Then for each value v that share the same key, one random number between the 1 and n is chosen. If the random number is lower or equal toz n then v is added to resultant list.

CHAPTER 5

DOMAIN-SPECIFIC LANGUAGE

In this chapter we present some fundamentals about Domain-Specific Language (DSL) and a language used as interface with the users and self-tuning.

A *DSL* is a way to approach some specific context through appropriate notations and abstractions [13]. 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. The developers must be expert in the domain in order to develop a DSL that cover the features required for the users. There are a lot of examples of DSLs in differents domains: (LEX, YACC, Make, SQL, HTML, CSS, LATEX, etc.) [5].

DLSs normally focus on specific domains, containing notations and specific abstractions. Also it is a *small* and *declarative* language. A DSL can be extended to different domains. Such DSL is called general-purpose language (GPL), because its expressiveness power is not restrict to an exclusive domain, examples of such DSLs are Cobol and Fortran, which could be viewed as languages focused on the domain of business and scientific programming [13].

DSL are used in several areas, such as Software Engineering, Artificial Intelligence, Computers Architecture (in this area a good exemple is VHSIC Hardware Description Language (VHDL), where VHSIC stand for **V**ery **H**igh **S**peed **I**ntegrated **C**ircuits), Database Systems (SQL, Datalog, QBE, Bloom), 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.

5.1 DSL Design Methodology

The first step to create a new DSL consists in identifing the problem domain. Depending on the context, it is not trivial to abstract the complete knowledge of the domain, because the developers must have a deep prior knowledge of the context, so considering all variables and intrinsics aspects belonging to the domain. Furthermore, sometimes the context can cover more than one domain (for example the GPLs). In other cases the correct abstraction of the domain is fast and there is not room for doubts and equivocation. In both cases the foreknowledge of the developers is the factor that influences the most the quality of the resulting DSL.

After identifying the problem domain developers must abstract all relevant aspects from it. For example *VHDL*, group semantic notations and operations on logical circuit that allows to express logical components: gates circuits, bus, datapath and control signals. With these four components we can describe any logical circuit since a ALU (Arithmetic Logic Unit), register bank till one complex microprocessor.

The next step consists in designing a DSL that expresses applications in the domain. DSL will have limited concepts which are all focused on the specific domain. To design the DLS, it is necessary to analyse the relationship between it and the existing languages. According to [23], there are some design patterns to develop a DSL based on existing languages that is represented by figure 5.1.

Pattern	Description
Language exploitation	DSL uses (part of) existing GPL or DSL. Important subpatterns: • Piggyback: Existing language is partially used • Specialization: Existing language is restricted • Extension: Existing language is extended
Language invention	A DSL is designed from scratch with no commonality with existing languages
Informal	DSL is described informally
Formal	DSL is described formally using an existing semantics definition method such as attribute grammars, rewrite rules, or abstract state machines

Figure 5.1: Design patterns - Figure extracted from [23].

In the implementation, a library with the semantic notations are built together with

a compiler that perfoms the lexical, syntactic and semantic analysis, after converting the DSL programs to sequence of library calls. Generally the library and the compiler are built with support of the tools or framework developed for this purpose. Xtext [14] and Groovy [17, 12] are examples of tools to develop DSLs.

5.2 Context Transformation

Our context is focused on Hadoop environment that have its own particularities. Thus, a context transformation is mandatory to implement the bacterionlogical algorithm on such environment.

On Hadoop there is huge set of configuration parameters, we called one specific parameter as *knob*. A job use several knobs which we called as set of knobs. When sets of knobs are agglomerate we have a population of set of knobs.

In the context tranformation, each component of genetic context was translated to one component of Hadoop environment. Figure 5.2 shows that a gene is transformed into a knob, an individual (which is a set of genes) is transformed into a set of knobs and, an individuals population is transformed into a population of set of knobs.

An interesting characteristic of the tranformation is its bijection that one component in the genetic domain is translated to one component in Hadoop domain. Beyond that the transformation has inversion property, i.e, all components in Hadoop domain can be translated to respective components in genetic domain. That properties represent compatibility between both domains and somewhat a good representativeness.

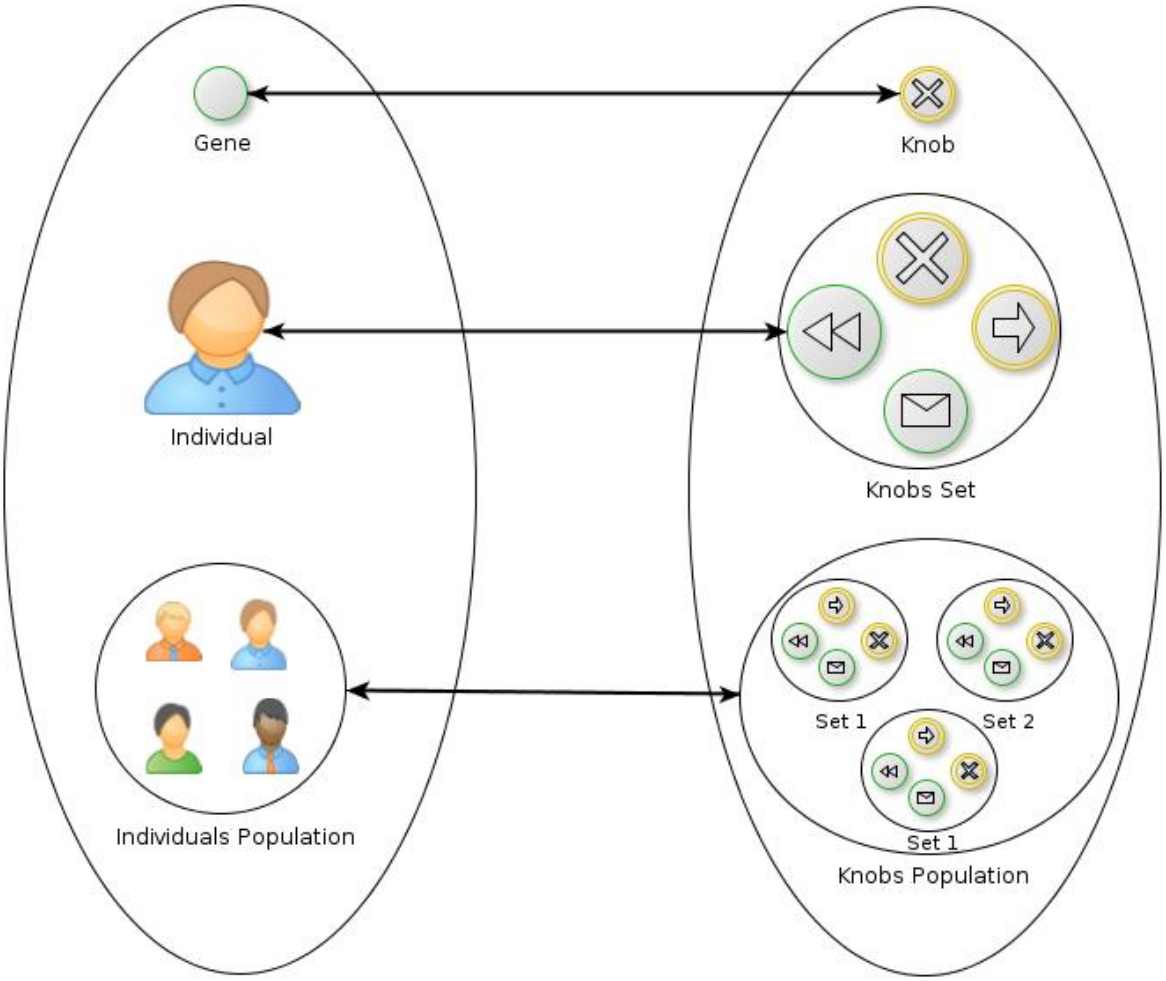

Figure 5.2: Context transformation.

5.3 DSL Proposal

Our DSL proposal is based on the **Xtext** framework [14]. It requires to define a grammar and rules for the specific domain. The base of the DLS is the self-tuning using BA. So our effort aims to describe rules to represent all aspects and components required for self-tunning.

Our domain has the following components:

- The job with its properties;
- The knobs;
- The knob with own type, minimum and maximum thresholds and its initial value.

Based on these components we present a preliminary version of our DSL:

```
DomainModel:
        job=Job;

Job:
        'Job' name=ID '{'
        setProperties+=Properties*
                setKnobs+=Knobs*
        '}'
;

Properties:
    'Properties' '{'
        properties+=Property
    '}'

Property:
    name=ID Value
;

Value: String;


Knobs:
        'knobs' '{'
                knobs+=Knob*
        '}'
;

Knob:
         name=ID Type
;

Type:
        IntType | FloatType | BoolType
;

IntType:
        'int' MinInt MaxInt '=' INT
;
MaxInt: INT;
MinInt: INT;

FloatType:
        'float' MinFloat MaxFloat '=' Float
;
MaxFloat: Float;
MinFloat: Float;

Float:
        INT*'.'INT*
;

BoolType:
        'boolean' '=' Boolean
;
Boolean:
        'true' | 'false'
;
```

Listing 3: Initial DSL proposal

Oll the rules forming our grammar are presented below:

1.

```
DomainModel:
        job=Job;
```

The first rule in a grammar is always used as the entry or starting rule. It espresses that the **DomainModel** contains one element **Job** assigned to a feature called *job*.

2.
```
Job:
    'Job' name=ID '{'
        setProperties+=Properties*
        setKnobs+=Knobs*
    '}'
;
```

The rule **Job** starts with the definition of a keyword (*Job*) followed by a name. Between brackets the job contains one indefinite number (*) of **Properties** and **Knobs** which will be added (+=) to a feature called setProperties and setKnobs, respectively.

3.
```
Properties:
    'Properties' '{'
        properties+=Property
    '}'
;
```

The rule **Properties** starts with the definition of a keyword **Properties** and between brackets contains one indefinite number (*) of **Property** which will be added (+=) to a feature called properties.

4.
```
Property:
    name=ID Value
;

Value: String;
```

These two rules are used to describe job properties, each property has an ID followed by its value. The value is an String.

5.
```
Knobs:
        'knobs' '{'
                knobs+=Knob*
        '}'
;
```

The rule **Knobs** starts with the definition of a keyword **knobs** and between brackets contains one indefinite number (*) of **Knob** which will be added (+=) to a feature called knobs.

6.
```
Knob:
        name=ID Type
;
```

The rule **Knob** contain one name followed by a **Type** with your peculiarities explained below.

7.
```
Type:
        IntType | FloatType | BoolType
;
```

The rule **Type** can accept three type: `integer`, `float` or `boolean`, this three are all possibles types on hadoop parameters configuration.

8.
```
IntType:
        'int' MinInt MaxInt '=' INT
;
MaxInt: INT;
MinInt: INT;
```

These three rules are used for integer types, the rule **IntType** starts with the keyword **int** followed by your respective minimum and maximum possibles values. In sequence there is the keyword '=' and the initial value for the knob.

9.
```
FloatType:
        'float' MinFloat MaxFloat '=' Float
;
MaxFloat: Float;
MinFloat: Float;

Float:
        INT*'.'INT*
;
```

These four rules are used for float types, the rule **FloatType** is similar the IntType rule, it starts with the keyword **float** followed by your respective minimum and maximum possibles values. In sequence there is the keyword '=' and the initial float value for the knob. The rule **FloatType** expresses the float format.

10.
```
BoolType:
        'boolean' '=' Boolean
;
Boolean:
        'true' | 'false'
;
```

The last rule **BoolType** expresses the boolean type, it starts with the keyword **boolean** followed by signal of '=' and the initial boolean value that can be **true** or **false**.

CHAPTER 6

PRELIMINARY IMPLEMENTATION

In this chapter we present our preliminary implementation composed of three modules: one front-end for the users to iteract with the system, one engine to choose good job configurations called tuning-by-testing and one back-end to report the new job configuration.

6.1 Overview

In figure 6.1 we show all components together. Fist of all, the user creates one file containing initial knobs, this file is submitted to component front-end which performs lexical, syntatic and semantic analysis on the file, after it is parsed and sent to engine component. This component, as described previously, actives the BA to choose one good job configuration until it reaches the criteria. The result is passed to back-end component that saved in a file.

One interesting feature is the result file can be used as input for the next round of the self-tuning, just only the user submit it as input. So the framework can work as incremental software to improve its last result.

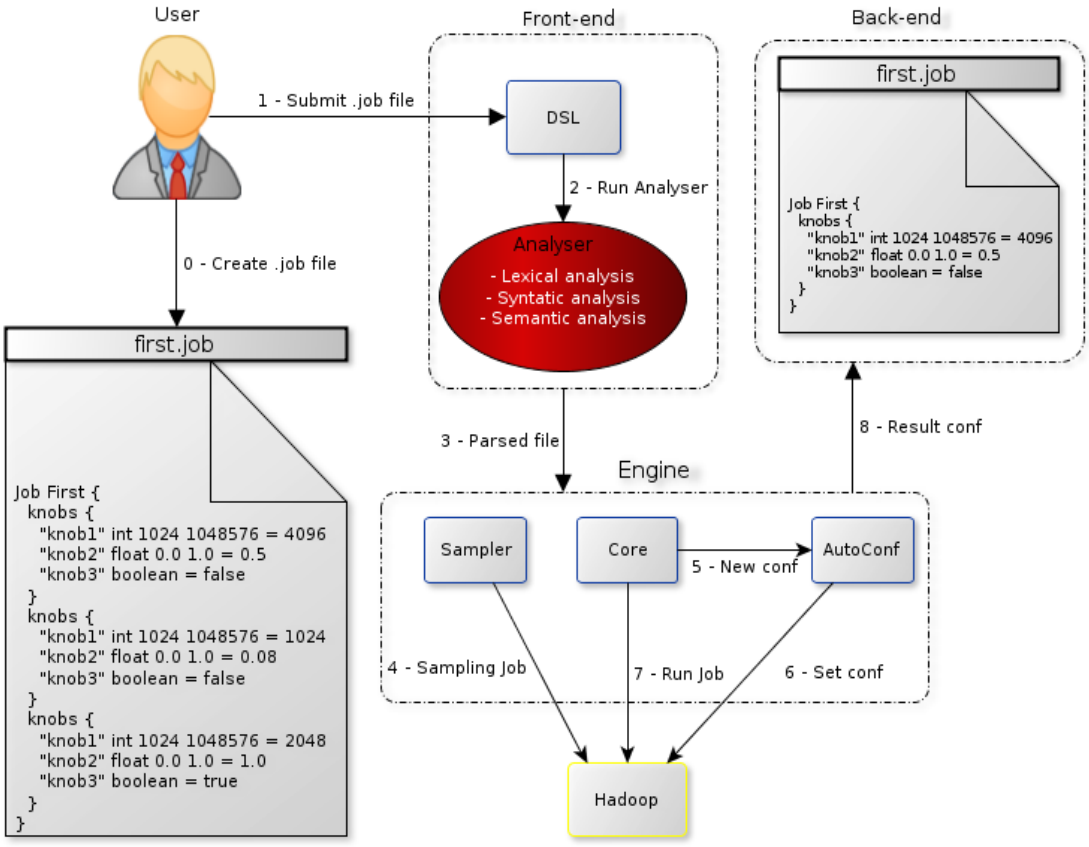

Figure 6.1: Implementation overview.

6.2 Front-end

The front-end is the DSL shown in the chapter 5, it consists in saying what is the job name to choose a good configuration and one or several set of knobs with their initial values, which can be assigned with any values since the last configuration or rule of thumbs until a random assignment.

One use case of the DSL is shown below, it contains the job name **WordCount**, its properties and several sets of knobs. The sets of knobs form the initial population for the BA, each set of knob (**knobs**) represents one individual and one single knob corresponds one gene, any change is done on knob level.

```
Job WordCount {
    Properties {
        jarPath /tmp/test/wc/wordcount.jar
        inputHDFSDir /tmp/test/wc/input
        outputHDFSDir /tmp/test/wc/output
        samplePercent 0.2
    }
    knobs {
            "dfs.block.size" int 1024 1048576 = 4096
            "io.sort.spill.percent" float 0.0 1.0 = 0.5
            "mapred.map.tasks.speculative.execution" boolean = false
    }
    knobs {
            "dfs.block.size" int 1024 1048576 = 1024
            "io.sort.spill.percent" float 0.0 1.0 = 0.08
            "mapred.map.tasks.speculative.execution" boolean = false
    }
    knobs {
            "dfs.block.size" int 1024 1048576 = 1048576
            "io.sort.spill.percent" float 0.0 1.0 = 1.0
            "mapred.map.tasks.speculative.execution" boolean = true
    }
}
```

Listing 4: Example of a configuration written in our proposal DSL.

As shown in 4 there are four properties for the job: the jar path, the HDFS input directory where are stored the input files, the HDFS output directory and the sample percent. Also there are three initial set of knobs to test, in this example the first set could be the last job configuration, the second the rule of thumbs suggested by the comunity and the last one random assignment. However, the users can be interested in testing new set of knobs that would be easy, just put a new set of knobs to test, so this front-end covers quite use cases as the users wish.

6.3 Engine

The engine is divided in three components: the component to generate data sample, the component to auto configure Hadoop and the core component that choosing configurations based on the BA.

with all configurations generated by that is responsible for choosing job configurations using BA.

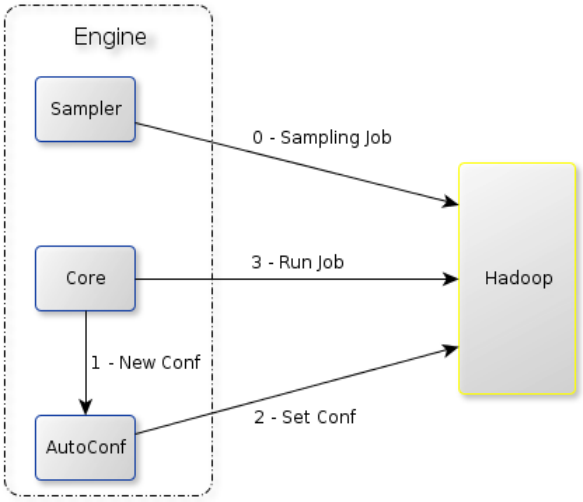

Figure 6.2: Engine processing.

6.3.1 Sampler component

engine

The sampler is responsable to generate data sample, so it sends a command to hadoop in order to run the sampling job and its output will be used by the core component, this step is represented by the action **0 - Sampling Job**.

6.3.2 AutoConf component

The AutoConf is one component developed by Ramiro, E [1].It is responsable for communicate with the Hadoop and inject new job configuration, as seen in the action **2 - Set Conf**. Despite the configurations assigned for the user in the Hadoop configuration files, the Hadoop will use job configurations sent for AutoConf.

6.3.3 Core component

The core component generates job configurations using the BA in order to test its performance and evaluate the response time, it sends the action **1 - New Conf** to component AutoConf. After assigned the configuration, the job is submited to Hadoop through the action **Run Job**, then its response time is evaluated and the job configuration maybe added or not to list of good configurations. The actions sequence *1, 2 and 3* occur until the BA finishes, i.e., until one criteria is reached.

6.4 Back-end

The back-end at the moment is is still being developed, currently the result is being saved in one file with the same format of the input file, but it contains just the best job configuration found by the core component.

[1]https://github.com/erlfilho/AutoConf

CHAPTER 7

INITIAL EXPERIMENTS

In this chapter we present initial experiments to show advantages of our proposal. First we present case study to show the high convergence of BA.

7.1 Bacteriological algorithm convergence

The first case study aims to show the high convergence of BA. We ran our solution in Hadoop standalone mode. The test consisted in running the BA with the wordcount job that calculated the frequency that occur in the input files.

The test was configured with a population of size 3, i.e. with three set of knobs per generation, each set of knobs had 10 knobs. The input files were generated with Hadoop job called *randomtextwriter* generating 10GB of random text file. The sample percent was of 10 percent of the input, i.e. 1GB of data.

We ran the BA in three rounds, up to three generations, up to six generations and last one up to ten generations. The input set of knobs for the first round was generated randomly, after for the next round the set of knobs were the best three of the previous round. So, ocurring feedback of the process.

In Fig. 7.1 the first round started with three random set of knobs, we can see the progress in 3 generations resulting an improvement almost of 3% better than the first generation.

In Fig. 7.2 the second round started with the best three set of knobs of the first round. We can see the two first generations were worse than the best result of the last round, this occured because greater usage of the cluster. However, from the third generation the performance improved and the resulting was better in 16%.

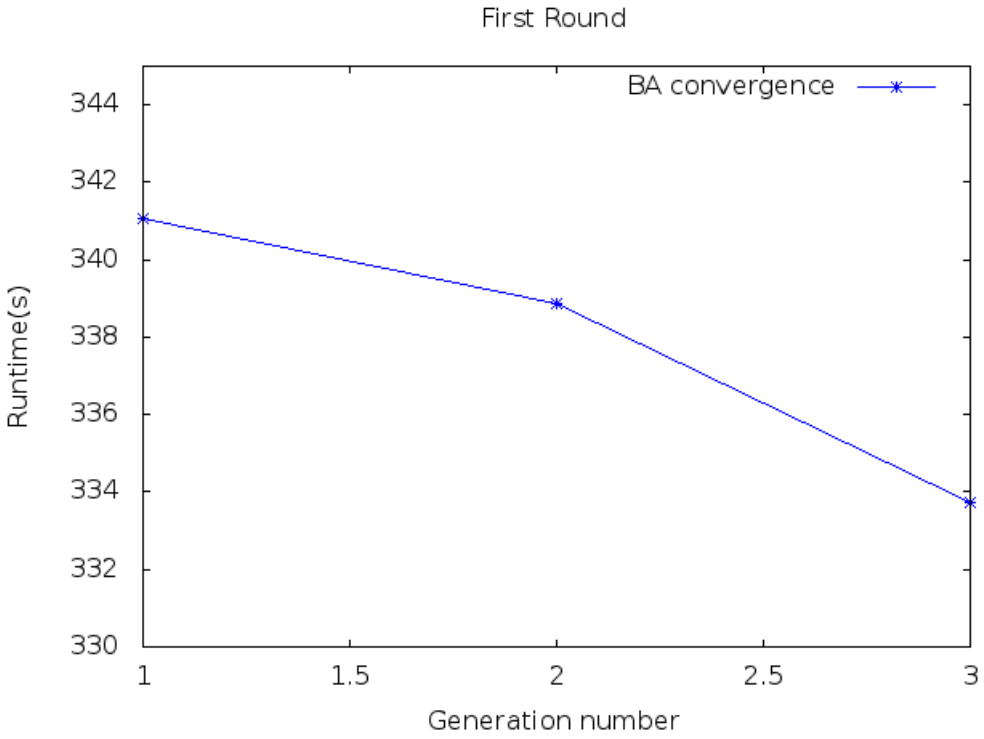

Figure 7.1: First round up to 3 generations

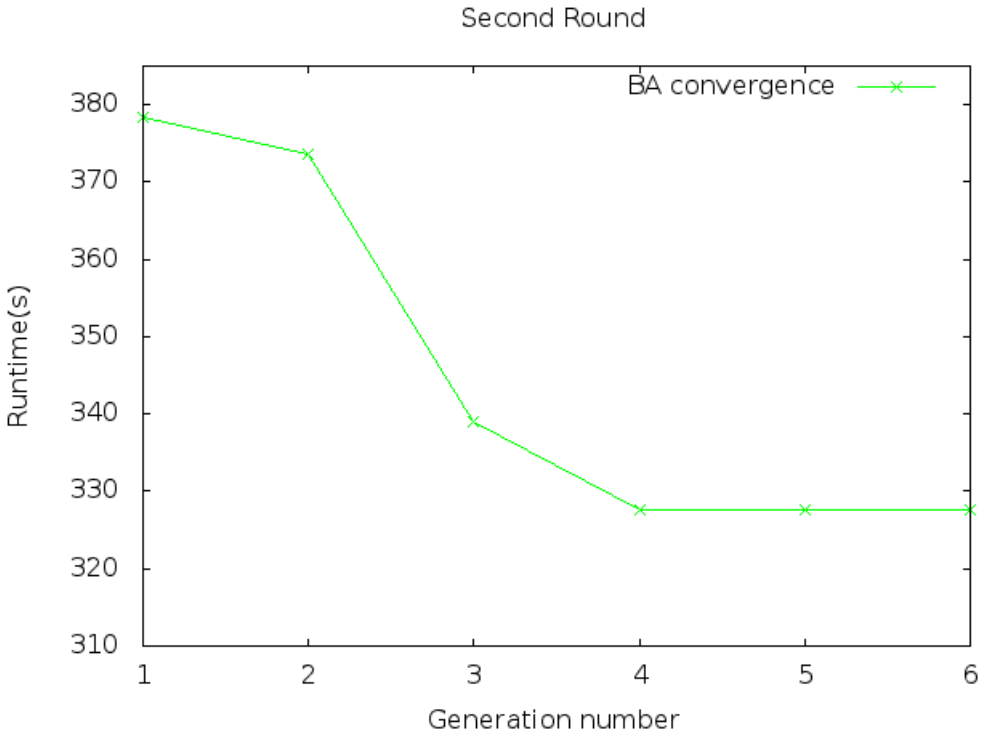

Figure 7.2: Second round up to 6 generations

In Fig.7.3 the third round started with the best three set of knobs of the second round. In this round we can realize the biggest drop of the runtime. It started with the same time of the best set reached in the second round, as the BA runs the runtime improves until stabilization in the final generations. This improvement occurred because some knobs values representing input, merge and sort buffers were changed, occurring further adjustment to the cluster.

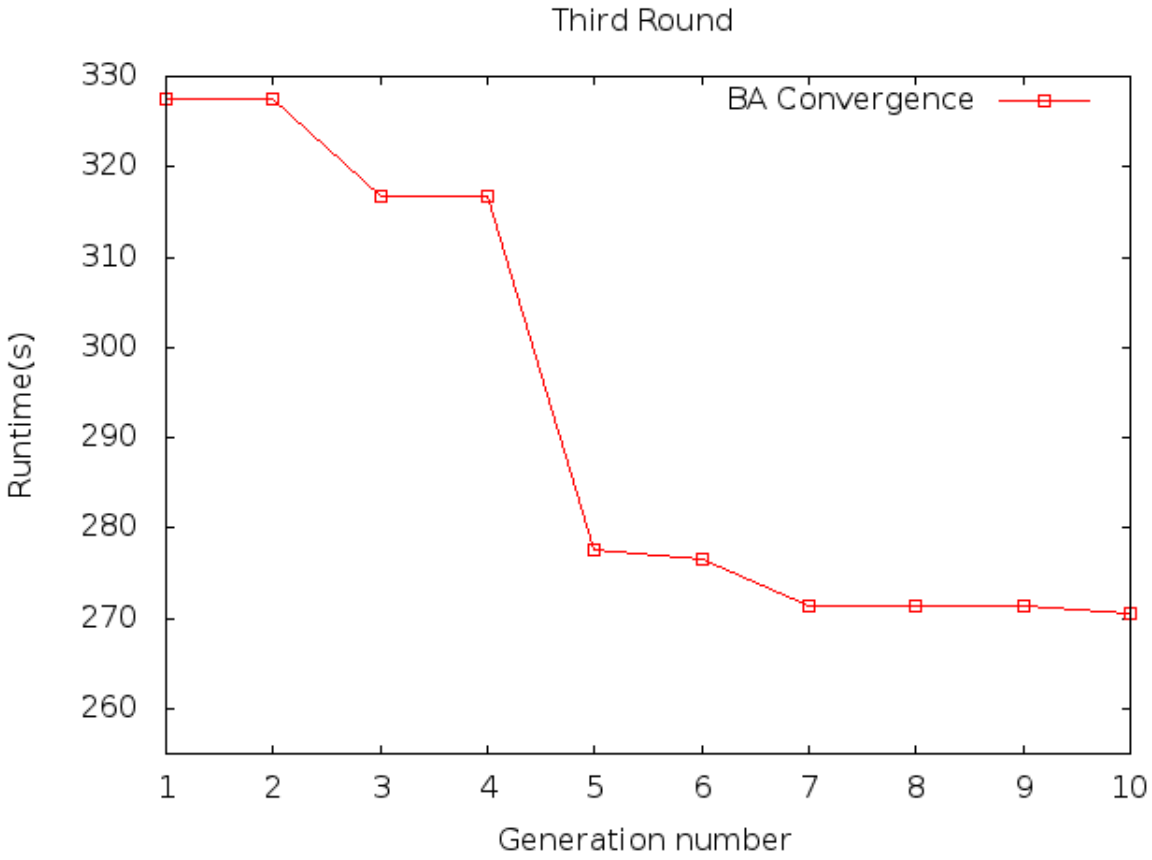

Figure 7.3: Third round up to 10 generations

With these three figures 7.1, 7.2 and 7.3 we can realize as the generations progress the runtime never recede, i.e. the runtime of one generation is lower or equal than the previous generation. This occurs because of the memorization operator avoids regressions in the BA performance.

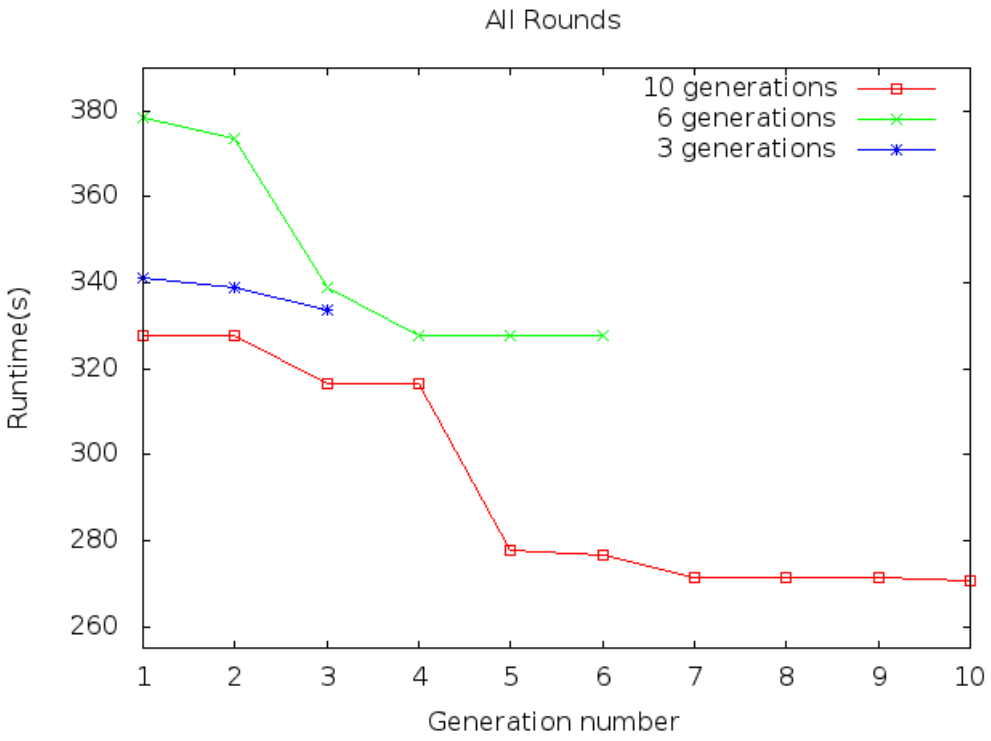

Figure 7.4: All Rounds

Figure 7.4 is an aggregation of all the rounds. It aims to show the overall improvement of the runtime. The first set of knobs reached the runtime of 380 seconds in the first round, and the last set of knobs reached the runtime of 270 seconds in the last round. The overall improvement was about 30 % which is a considerable improvement.

Moreover, the result were influenced by the feedback provided by the software between the rounds. So, as the rounds progress the improvement continue occurring influenced by the feedback operator.

CHAPTER 8

CONCLUSION

8.1 Contribution

The initial experiments show the feedback feature of our solution. This feature improve more quickly the BA performance that directly impacts the variation of runtime. Using BA the performance never receds showing up attractive for self-tuning on Hadoop through its convergence.

The sample method on Hadoop is an original approach to avoid executing BA on all data storage. It benefits of the MapReduce paradigm and the key-value model, thus taking greater advantages of the Hadoop framework.

The context transformation described in 5.2 shown the genetic domain and hadoop domain are consistent for the bijection property. So, the DLS developed has an intuitive use.

8.2 Future work

The front-end(DSL) is not integrated with the others components and some rules need to be added to the grammar.

Improve the intregration between AutoConf component because it has been calling as system calls. The task consist in create a library in order to use the AutoConf classes.

Yet is missing to test our solution on distributed Hadoop. We want to run tests increasing machines on the cluster and analyze the performance. So, we can analyze what the BA performance on dynamic clusters.

Maybe, improve the BA implementation using heuristics to orient the assignment values to the knobs.

BIBLIOGRAPHY

[1] apache.org. Language manual sampling. Site: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Sampling, 2013. Accessed on 11th July 2013.

[2] apache.org. Welcome to hive! Site: http://hive.apache.org/, 2013. Accessed on 23th September 2013.

[3] Inc. Aster Data Systems. In-database mapreduce for rich analytics. Site: http://www.asterdata.com/resources/mapreduce.php, 2013. Accessed on 3rd October 2013.

[4] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Software, Testing, Verification & Reliability (STVR)*, 15:73–96, June 2005.

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

[6] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, August 2012.

[7] Cloudera, Inc. Algorithms every data scientist should know: Reservoir sampling. Site: http://blog.cloudera.com/blog/2013/04/hadoop-stratified-randosampling-algorithm, 2013. Accessed on 14th July 2013.

[8] Edgar Frank Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[9] P.G. de Vries. *Sampling theory for forest inventory: a teach-yourself course*. Springer-Verlag, 1986.

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

[11] Jeffrey Dean, Sanjay Ghemawat, and 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.

[12] Fergal Dearle. *Groovy for Domain-Specific Languages*. june 2010.

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

[14] eclipse.org. Xtext. Site: http://www.eclipse.org/Xtext, 2013. Accessed on 1st April 2013.

[15] en.wikipedia.org. Systematic sampling. Site: http://en.wikipedia.org/wiki/Systematic-sampling, 2013. Accessed on 2nd July 2013.

[16] Greenplum. A unified engine for rdbms and mapreduce. Site: http://docs.huihoo.com/greenplum/Greenplum-MapReduce-Whitepaper.pdf, 2013. Accessed on 3rd October 2013.

[17] groovy.codehaus.org. Domain-specific languages with groovy. Site: http://groovy.codehaus.org/Writing+Domain-Specific+Languages, 2013. Accessed on 1st April 2013.

[18] Greg Grothaus. Reservoir sampling - sampling from a stream of elements. Site: http://gregable.com/2007/10/reservoir-sampling.html, 2013. Accessed on 6th August 2013.

[19] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin and S. Babu. Starfish: A self-tuning system for big data analytics. *In Proc. of the 5th Conference on Innovative Data Systems Research (CIDR '11)*, January 2011.

[20] hadoop.apache.org. Apache hadoop. Site: http://hadoop.apache.org/, 2013. Accessed on 24th March 2013.

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

[22] Jennifer Widom Jeffrey D. Ullman, Hector Garcia-Molina. *Database systems - the complete book*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2009.

[23] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[24] monetdb.org. Database sampling. Site: http://www.monetdb.org/Documentation/Cookbooks/SQ
2013. Accessed on 6th August 2013.

[25] monetdb.org. Monetdb. Site: http://www.monetdb.org, 2013. Accessed on 2nd July 2013.

[26] Prashanth Mundkur, Ville Tuulos, and 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.

[27] Pivotal. Pivotal greenplum databasee. Site: http://gopivotal.com/pivotal-products/data/pivotal-greenplum-database, 2013. Accessed on 3rd October 2013.

[28] Randomsampling.org. Random sampling. Site: http://www.randomsampling.org, 2013. Accessed on 2nd July 2013.

[29] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

[30] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[31] Jeffrey Scott Vitter. Faster methods for random sampling. *Commun. ACM*, 27(7):703–718, 1984.

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

[33] wikipedia.org. Reservoir sampling. Site: http://en.wikipedia.org/wiki/Reservoir-sampling, 2013. Accessed on 14th July 2013.

TIAGO RODRIGO KEPE

SELF-TUNING BASED ON DATA SAMPLING

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