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

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 day-to-day. 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(MR) jobs. However, in MR jobs developers are allowed to setup several parameters to draw optimal performance from the available resources, but find a good configuration is time consuming and a configuration found in an execution may be impracticable for the next time.

In order to facilitate and automate tuning hadoop jobs, we propose a self-tuning based on data sampling. Our approach allows to find a good job configuration considering the data stored and the job in question. The users till can provide their usual job configurations then 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 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 the big companies are processing and generating a vast amount of data day-to-day. This 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 possible solution to build applications to data distributed processing, e.g. the MR programing paradigm which is based on key-value model[9].

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 [29] to process their information. Besides Hadoop, several other implementations are available: Greenplum MapReduce [14], Aster Data [2], Nokia Disco [24] and Microsoft Dryad [19].

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 resource usage.

Implement jobs on Hadoop is simple, but there are many of 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 good configuration [5]. Therefore, data sampling can be useful to improve on processing adjust of parameters instead of processing all data set how is done in [17]. 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 [3] to select good configurations for MR jobs. Based in our knowledge the best way to find such configurations is to run the jobs with its and analyse the performance, normally this process is done manually. 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 method to implement data sampling 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 [3] in order to avoid run it on all data storage we develop one method to obtain data samples from hadoop input data. For data sampling we considered a lot of aspects related the paradigm MR, key-value model and others hadoop particularities. Our approach presents an user interface which through a domain specific language (**DSL**), to facilitate the user iteraction with data sampling and drive tuning procedures.

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 latency time that the job led to conclude. Furthermore, we intend to use other measures of performance such as amount of intermidiate data, network usage and cpu usage.

1.4 Outline

- Chapter 2 introduces the fundamental concepts of the key-value model, MR paradigm and hadoop framework.
- Chapter 3 presents the bacteriological algorithm.
- Chapter 4 presents the method to generate sampling data.
- Chapter 5 introduces the concepts of the domain specific language and our proposal DLS.
- Chapter 6 we presents our initial implementation with all components.
- Chapter 7 we discussed a case study performed with our solution.
- Chapter 8 we 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, MapReduce paradigm and Hadoop framework.

2.1 Key-value model

Key-value model is a simplified model for data storage. 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, thus all detailing of data is done in runtime. Unlike other models, such as the relational model [7], in which the simplistic notion of relation already gives some sense for the data, similarly to the hierarchical data model [26] in which the links that connect the records give details for the data.

The Data Warehouse(DW) was create to solve some particularities involving relational model. It is a repository that aggregates data from several sources [26], 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 cause one issue to process large amounts of data, requiring much computing power while querying the data. One programming paradigm that 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, [20]. 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 omit all complexity for that users 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.

Acording [8]: [*"the computation takes a set of input key/value pairs, and produces a set of output key/value pairs."*]. An 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:

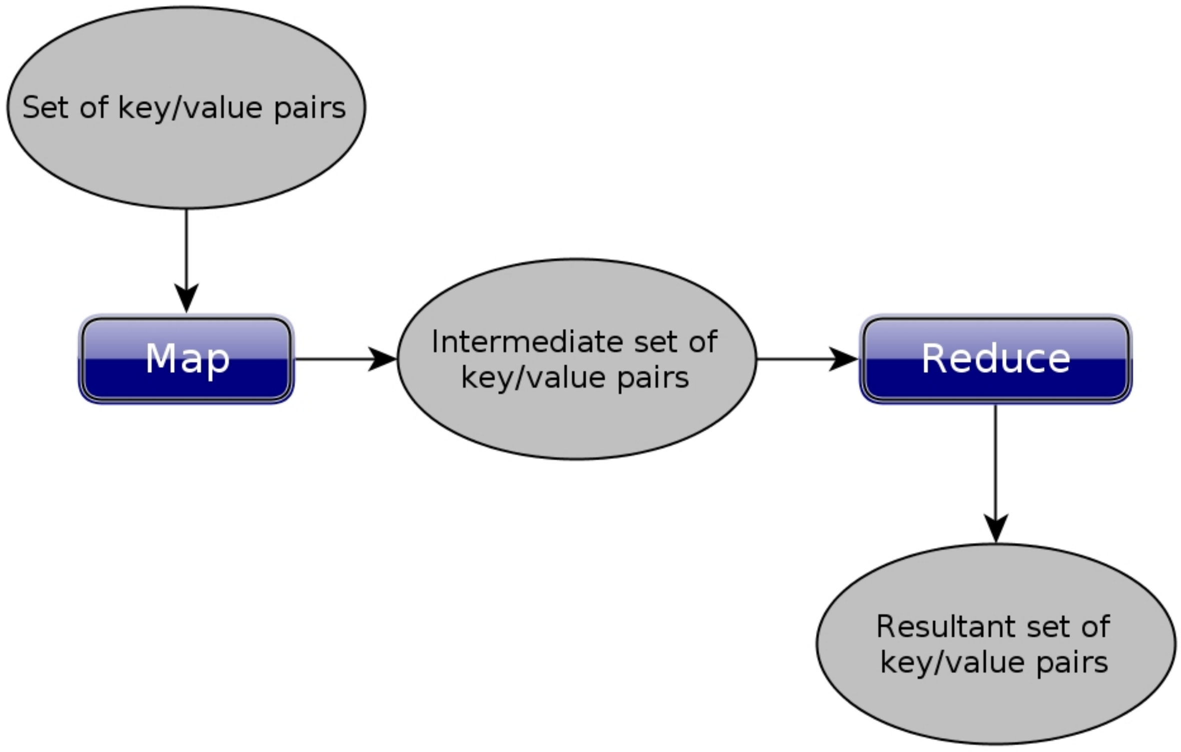

Set of key/value pairs
Map
Intermediate set of key/value pairs
Reduce
Resultant set of key/value pairs

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 [14], Aster Data [2], Nokia Disco [24], Microsoft Dryad [19], and the one open-source implemantation from Apache: Hadoop [18].

The Hadoop is a framework for reliable, scalable and distributed computing. It provide an interface to implement the map and reduce functions in high-level which are internally called as map and reduce tasks. It was designed for users 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 distribuition of the 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 in two components the *NameNode* and *DataNode*. The NameNode is placed in one master machine, it stores all metedata and manages all DataNodes. The DataNode stores data, when one DataNode starts it connects to theNameNode, then responds to requests from the NameNode for filesystem 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(java, ruby or python) that implements so the map and reduce functions. Initially the master machine receives jobs with the relative input directory in the HDFS where are all 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.

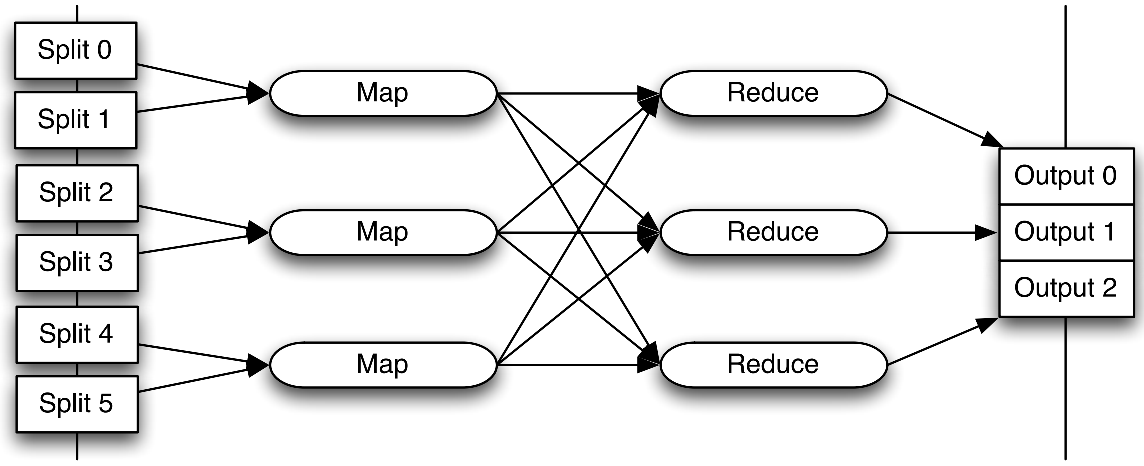


Figure 2.2: Execution of Map and Reduce operations

2.3.2 MapReduce programing

The whole processing is based on $\langle key, value\rangle$ pairs. The mappers receive the file blocks, the mappers 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 generate the intermediate result set of key and values $\langle set(k2, v2)\rangle$, when the mappers finished all values for *k2* are agrouped in a list and the respective pair $\langle k2, list(v2)\rangle$ is generated. This pairs are sorted and pass 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 result are already available in memory, a local reduce function *Combiner* is used for optimization reasons, then all values for determinated 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 good 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 and the reduce is to sum the amount found each patterns.

The **map()** method has four parameters: **key**, which is never used; **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.

```java
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 [18]

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.

```java
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 [18]

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 find the words that contains the vowel **o** in the midle of them.

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

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

On GA context there are three important components:

- **Gene** that is the smaller particle.
- **Individual** that is composed for genes.
- **Population** that is composed for individuals.

The GA works on the gene level, so all changes are done in this level. At first glance, changes done on gene seems tiny and without much relevance, but them can be crucial for adaptation of the individual in the environment, also genetic changes can be crucial to survival of one entire population or even mean survival of a species.

The GA process describe in Figure 3.1 has its main strategy based in 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 abilities in adapt themselves to the environment.

Those abilities can be calculated according with a function *F(x)* that is called as the fitness of the individual like described in Figure 3.1.

The choice of one individual is based in your fitness, it is similar to spin a roulette wheel where each individual receive slots according with your 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 slots / F(X) = 1/10*. Thus the individual fitness is greater, then your number of copies tends to be greater.

- **Crossover**: the crossover is similar 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 parent. More minutely in the genetic algorithm two individuals are chosen randomly **(A, B)**, an integer k, between 0 and the size n of an individual less 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**: after the crossover stage one mutation occur in the genes of new individuals. The natural process consist basically in change enzymes or proteins of genes in order to create new individual. The process on GA is simple in which one or more genes are selected randomly and then are changed (e.g. change one or more nucleotides of the DNA of one chromosome).

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

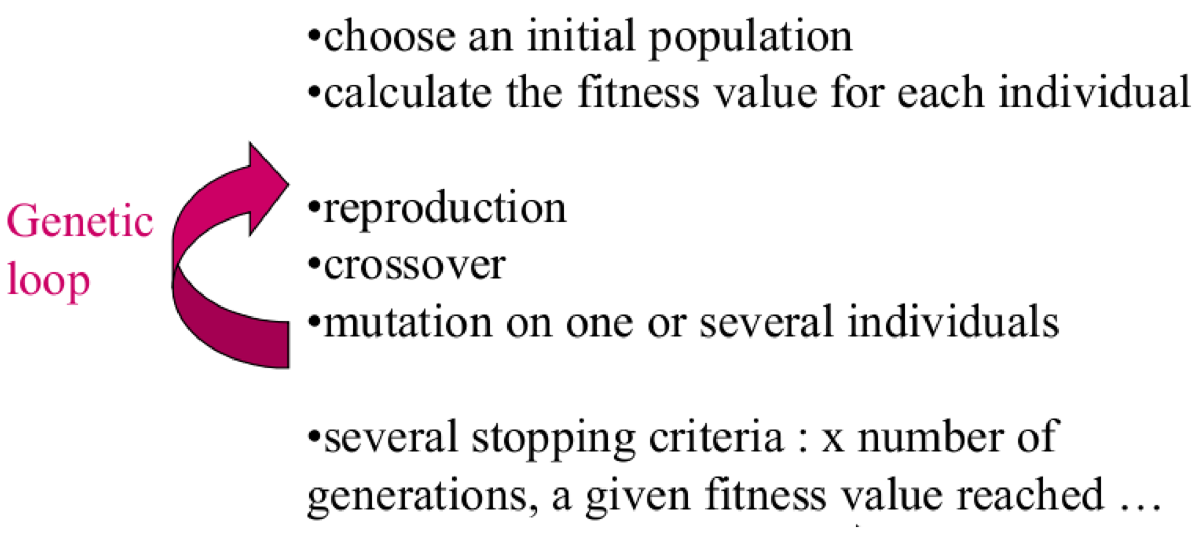

•choose an initial population
•calculate the fitness value for each individual
Genetic loop
•reproduction
•crossover
•mutation on one or several individuals
•several stopping criteria : x number of generations, a given fitness value reached …

Figure 3.1: Genetic Algotithm process - Figure extracted from [3].

3.2 Bacteriological Algorithm

The bacteriological algorithm is part of the family of genetic algorithms that works on genetic context. A minor particle this algorithm is a gene, but without minor importance, all changes are influenced by it. The genes when clustered form an individual that have more representativeness than an gene and the top is the population that is set of individuals.

Compared to GA the Bacteriological Algorithm(BA) the individual is one bacteria and the focus of the algorithm is to adapt itself in a given specific environment. The algotithm is more one adaptive approach of GA than one otimization, but it have some peculiarities that improve some issues involving the GA and change your behavior.

BA introduces a new mechanism called memorization that is responsable for memorize the best individuals created along the generations. As described in [3], it was proposed to improve the convergence of the GA, the introduction of the new mechanism might appear one small modification, but actually reflects one crucial change on GA's.

Besides of the introduction the new mechanism, the crossover mechanism was removed because of the bacteria behavior on its adaptation process in the environment. This mechanism cannot be used anymore, in terms of natural bacteriologic process the remotion of the crossover make sense, the bacteria reproduce themself asexually, consequently there is not crossover between two individuals, because the reproduction process consist in duplication of DNA of one bacterium and after a division to form two new bacteria.

The algorithm in high-level of abstraction is described in Figure 3.1. The BA is started and has four main mechanisms: **Fitness computation**, **Memorization**, **Reproduction** and **Mutation** which are detailed below:

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

- Memorization: is the main mechanism introduced by the BA. Its is responsable 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 consist in memorize 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 can grow up exponentially, so thresholds must be established.

- Mutation: this stage 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.

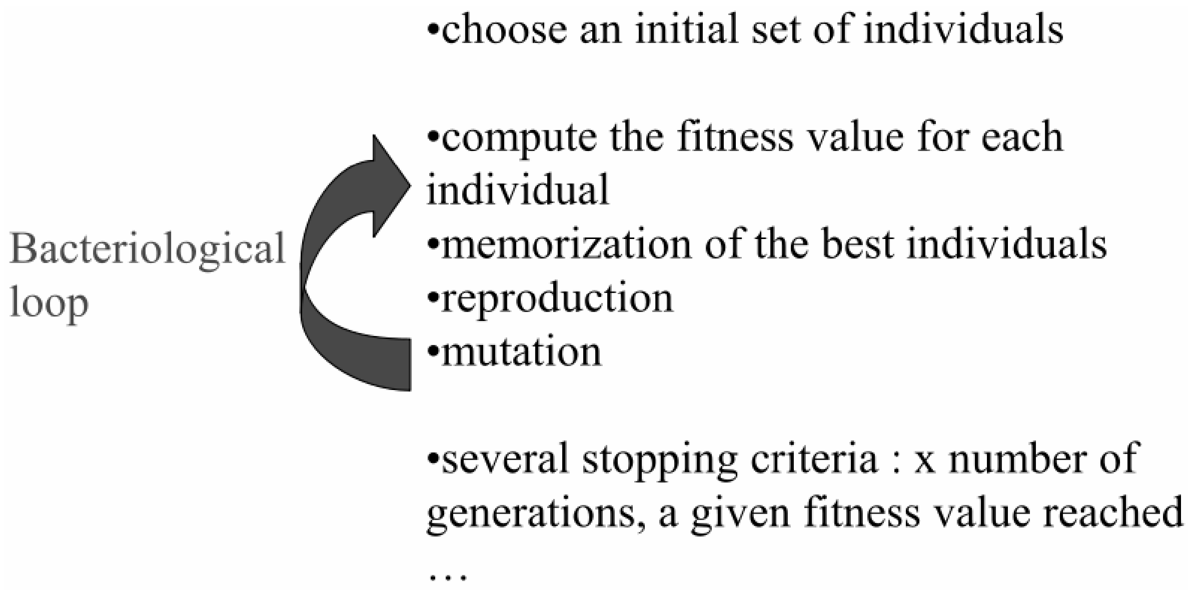

•choose an initial set of individuals
•compute the fitness value for each individual
•memorization of the best individuals
•reproduction
•mutation
•several stopping criteria : x number of generations, a given fitness value reached …
Bacteriological loop

Figure 3.2: Bacteriological Algotithm process - Figure extracted from [3].

CHAPTER 4

SAMPLING ON HADOOP

In this chapter we present one method for data sampling on hadoop that is basead in a random algorithm.

4.1 Motivation for sampling

One relevant aspect in Big Data environment is the vast amount of data, that is the main barrier to find a good job configuration job. The bacteriological algorithm is an option to create new configurations, but these configurations must be tested in order to select which is the best for the job at the a given moment. Moreover, the dynamism 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 for these dynamisms described the job configuration needs to be reviewed constantly. This review process consist in to choose a new configuration and to test it in order to analyse the performance. But the test step cannot be done on the entire data.

So, how must we test the job configurations generated by BA? 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 much time and also would be impracticable due larger number of intermediate configurations generated by the algorithm.

4.2 Challenge for data sampling in Big Data environments

On Big Data environment there are several aspects involving distributed computing and storage. For data sampling the aspects involving storage distributed are more relevant. In this context, without doubt, the data volume is the main issue because the data sampling must be done distributed too, otherwise one machine couldn't bear all data storage in the cluster then make the data sampling, e.g. suppose that one cluster storage 100 terabytes and we want data sampling of 20 percent, then each node must do local sampling and store it locally, so the resultant sample will be 20 terabytes which one conventional machines doesn't have this storage capacity.

So the resultant data sample must be storage distributed in the cluster, because even being a sample the result can be big and a single machine couldn't bear too. After the sampling obtained, jobs can run on the sampling, avoiding to run on 100 terabytes and run just 20 terabytes as the example.

The Hadoop has the structure for distributed storage, so one way to obtain data sample data is to utilize its benefits, i.e. taking into consideration that the data already are distributed storage, so we can build one MapReduce program to sample data and we will be benefiting of its advantages as framework to distributed computing.

One of the most used data sample techniques is **Random Sampling** that consists in selecting a pre-determined amount of data randomly [25]. 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 [25]. Another thechnique is **Systematic Sampling** that has one number k which is chosen randomly or chosen with some criteria, then randomly one element is chosen of the population and from this element till the next k-esimo elements in sequence are selected to sample [13].

In the context of big data there are some implementations of data sample. One example is the **MonetDB** which is column-oriented database management system and was designed to hold data in main-memory and processes large-scale data distributed [23], this database support data sample and use the *Algorithm A* that is based in random sample method [22].

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 in V, n and N is calculated the number s, from s is created the records set called S, this set will contain the *(s + 1)*st records in the file, then one record is chosen from S and put in the sample. The records present in S are skipped in the next interaction [28].

Another database management system that performs data sampling based on random methods the **Hive** a data warehouse system for Hadoop. It samples in 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. With the colunm name 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 consist in selecting the blocks that match with the specified block size.

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

In Hadoop, data are stored unstructured and this characteristic is the biggest challange to develop data sampling. According to [27, 6, 16, 30] the challange 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 is part of the randomized algorithm family and consists in choosing randomly k elements from a list L containing N items. The length N is either unknown or large enough to fit in memory. See algorithm 1.

The goal is to build a reservior smaller than the memory. So it receive as parameter the number k that is the resultant sampling length and *stream* of data that constantly receive new data. Initially the resultant sampling is assigned with the first k *elements*, so the algorithm aim to calculated the probability of the next element, e.g. *k+1*, to be

Algorithm 1: Algorithm for Reservoir Sampling

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

inserted to the sample, which is *P(k+1) = k/(k+1)*, after one random number(*chance*) between 0 and 1 is chosen, if *chance* < *P(k+1)*, then the next element is putted in random position in the resultant sample.

However, choosing the number k is a hard task, because the resultant sample must be representative and fit in memory. To solve that problem Vitter [27] suggests that the reservoir be stored on secundary storage(hard disk). But this approach may be implactible on the Hadoop context, because the secundary storage is HDFS and the time of retrieving the reservoir and update is very large, because of the override among contact namenode, it look up the reservoir in the cluster and make it available.

4.3 One method for data sampling on Hadoop

We propose the follow algorithm based on the MapReduce paradigm to generate data samples on Hadoop. It follows the random algorithm family which has been used largely in database and big data environment.

First of all, we will 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 $\langle lineNumer, \{content1, content2, ..., cont$

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.

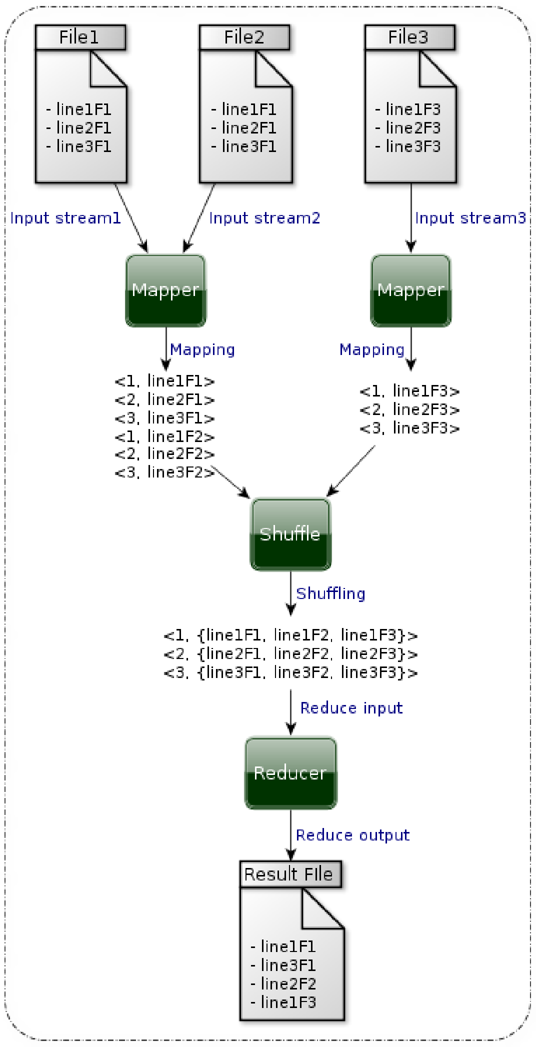


Figure 4.1: Map and Reduce random sample process

The map function is described by algorithm 2. Before presenting the algorithm, some previous definitions are necessary. Let us denote F the files set storage on the cluster, f

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

Algorithm 2: Map function for data sample

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

The map function consists in classify 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. This aggregated values are the input for reduce function.

Algorithm 3: Reduce function for data sample

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

The reduce function is in charge of the sampling and is described by algorithm 3. 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 one key belonging the *mapList* and *values* are a set of values sharing the same key. The v is one 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 sharing the same key, one random number between the 1 and n is chosen. If the random number is less than equal n then v is added to resultant list.

CHAPTER 5

DOMAIN-SPECIFIC LANGUAGE

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

A *DSL* is way to approach of some specific context through appropriate notations and abstractions [11]. 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.)** are classical examples of DSLs [4].

DLSs normally focus in specific domain, containing notations and specific abstractions. Also it is *small* and *declarative* languages. But, a DSL can be extended to differents 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 [11].

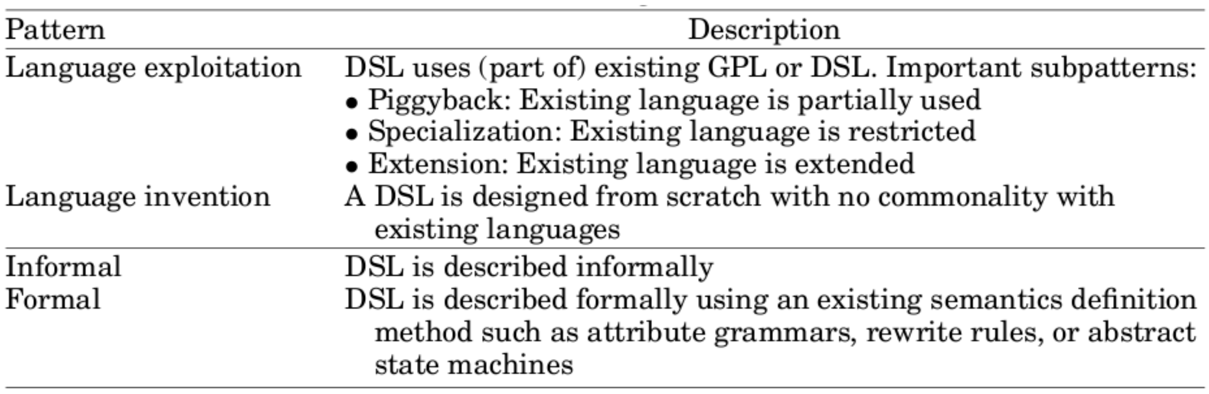
DSL are used in several big areas, such **Software Engineering**, **Artificial Intelligence**, **Computers Architecture**(in this area a good exemple is VHSIC Hardware Description Language (VHDL), where VHSIC mean **V**ery **H**igh **S**peed **I**ntegrated **C**ircuits), **Database Systems**(SQL, Datalog, 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 is identifing the problem domain. Depending on context is not so easy 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, some times 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 margin for doubts and equivocation. In both cases the foreknowledge of the developers is the factor that more influences in good or bad DSLs resultants.

After identifying the problem domain the developer must abstract all relevant aspects from it. For example *VHDL*, it agroup semantic notations and operations on logical circuit that allows to express logical components, 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 is designing a DSL that expresses applications in the domain. DSL will have limited concepts which are all focused on the specific domain. For design the DLS is necessary to analyse the relationship between it and the existing languages. According with [21] 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 [21].

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** [12] and **Groovy** [15, 10] 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 parameters as **knob**. A job use several knobs that are one set of knobs. When sets of knobs gathered we have a population of knobs.

In the context tranformation, each component of genetic context was translated to one component of hadoop environment. Like shown in the figure 5.2 we can realize that one gene was transformed to one knob, one individual(that is a set of genes) was transformed to one set of knobs and one individuals population was transformed to a population 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.

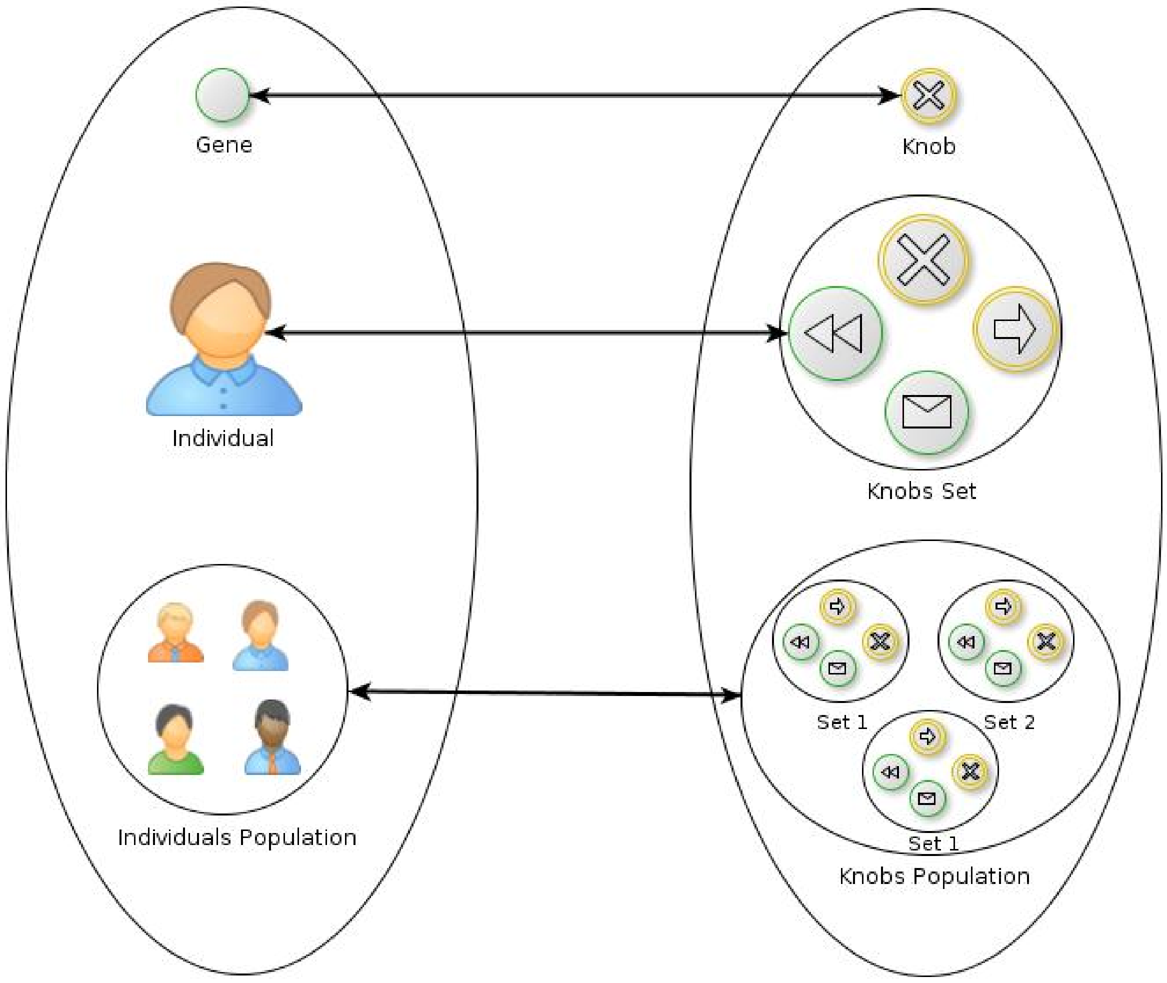


Figure 5.2: Context transformation.

5.3 DSL Proposal

Our DSL proposal is based on the **Xtext** [12] framework. 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 componens we present one draft of our DSL proposal:

```
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

Let's explain all rules involving our grammar:

1.
```
DomainModel:
        job=Job;
```

The first rule in a grammar is always used as the entry or start rule. It says 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 'braces' the job contains one arbitrary 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 'braces' contains one arbitrary 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 'braces' contains one arbitrary 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 key word **'='** and the initial float value for the knob. The rule **FloatType** expresses the float format.

10.

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

The last one 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

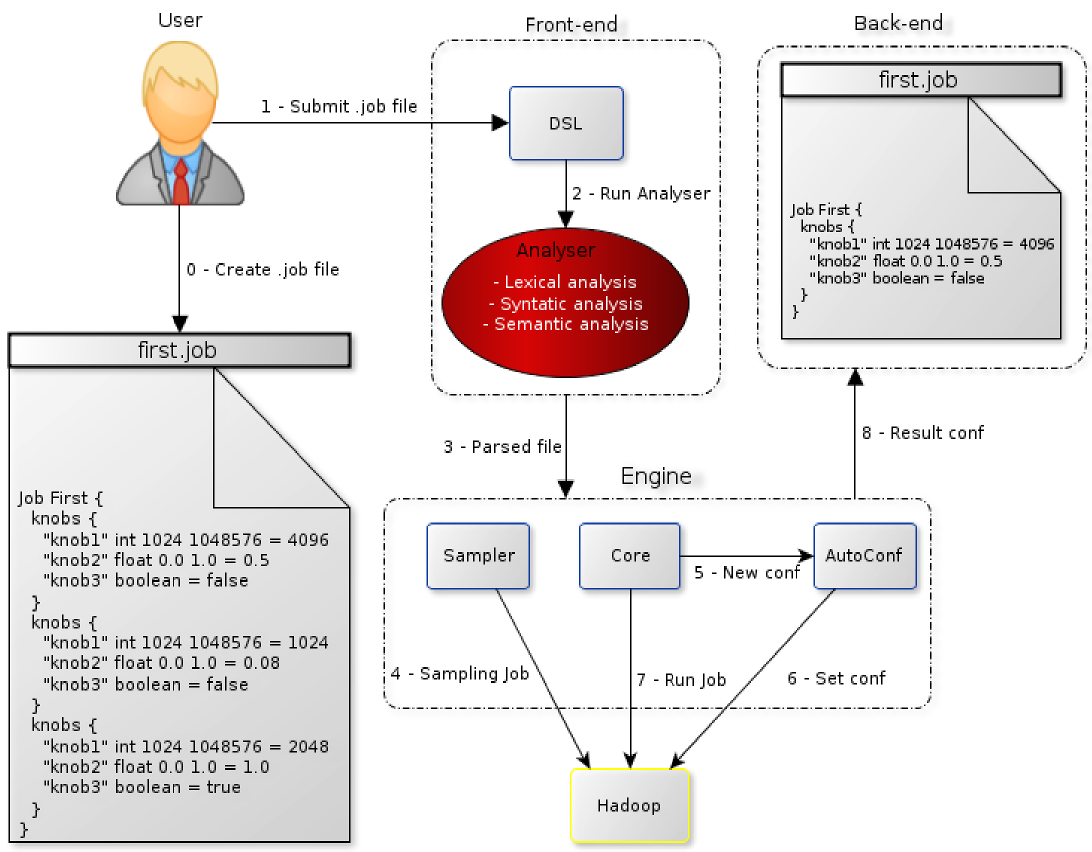
PRELIMINARY IMPLEMENTATION

In this chapter we present our preliminary implementation composited by 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 the Figure 6.1 we show all components together. Fist of all the user create 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, how already explained, active the BA to choose one good job configuration until reach 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.



```
Job First {
  knobs {
    "knob1" int 1024 1048576 = 4096
    "knob2" float 0.0 1.0 = 0.5
    "knob3" boolean = false
  }
  knobs {
    "knob1" int 1024 1048576 = 1024
    "knob2" float 0.0 1.0 = 0.08
    "knob3" boolean = false
  }
  knobs {
    "knob1" int 1024 1048576 = 2048
    "knob2" float 0.0 1.0 = 1.0
    "knob3" boolean = true
  }
}
```

```
Job First {
  knobs {
    "knob1" int 1024 1048576 = 4096
    "knob2" float 0.0 1.0 = 0.5
    "knob3" boolean = false
  }
}
```

Figure 6.1: Implementation overview.

6.2 Front-end

The front-end is the *DSL* shown in the chapter 5, it consist in say what is the job name to predict 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 and its set of initial knobs. This set of knobs forms the initial population for bacteriological algorithm, each set of knob (**knobs**) represents one individual and one single knob corresponds one gene of individual, any change is done on gene level i.e. 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: Usage of DSL Proposal

As the example shown 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 interested in testing new knobs that would be easy, just only put a new set of knobs to test, so this front-end covers enough use cases or combination cases as much 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 with all configurations generated by the core component that is responsible for choosing job configurations using BA.

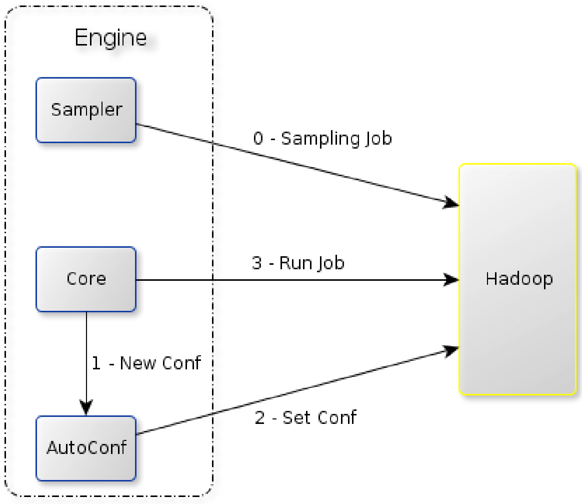


Figure 6.2: Engine processing.

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 command **0 - Sampling Job**.

6.3.2 AutoConf component

The AutoConf is one component developed by Ramiro, E. it is responsable for communicate with the hadoop and inject new job configuration, as seen in the command **2 - Set Conf**. So independent of configuration files or configurations assigned for the own job, the hadoop will use job configuration sent for AutoConf.

6.3.3 Core component

The core component chooses job configurations in order to test its performance and evaluate the latency time using the bacteriological algorithm, it sends the command **1 - New Conf** to component AutoConf after assigned the configuration, the job is submited to hadoop through the command **Run Job**, then your latency time is evaluated and the job configuration may added or not to list of good configurations. The commands sequence *1, 2 and 3* occur until the BA finish, i.e., until one criteria is reached.

6.4 Back-end

The back-end at the moment is undefined, currently the result is being saved in one file with the same format of the input file, but with an exception it contains just the best job configuration found by the BA until the criteria was reached.

CHAPTER 7

INITIAL EXPERIMENTS

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

7.1 Bacteriological algorithm convergence

Our first study case aims to show the high convergence of BA. We ran our solution in Hadoop standalone mode. The test consisted in run 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 7.1 the first round started with three random set of knobs, we can see the progress in 3 generations resulting an improvement of 6 seconds smaller than the first generation.

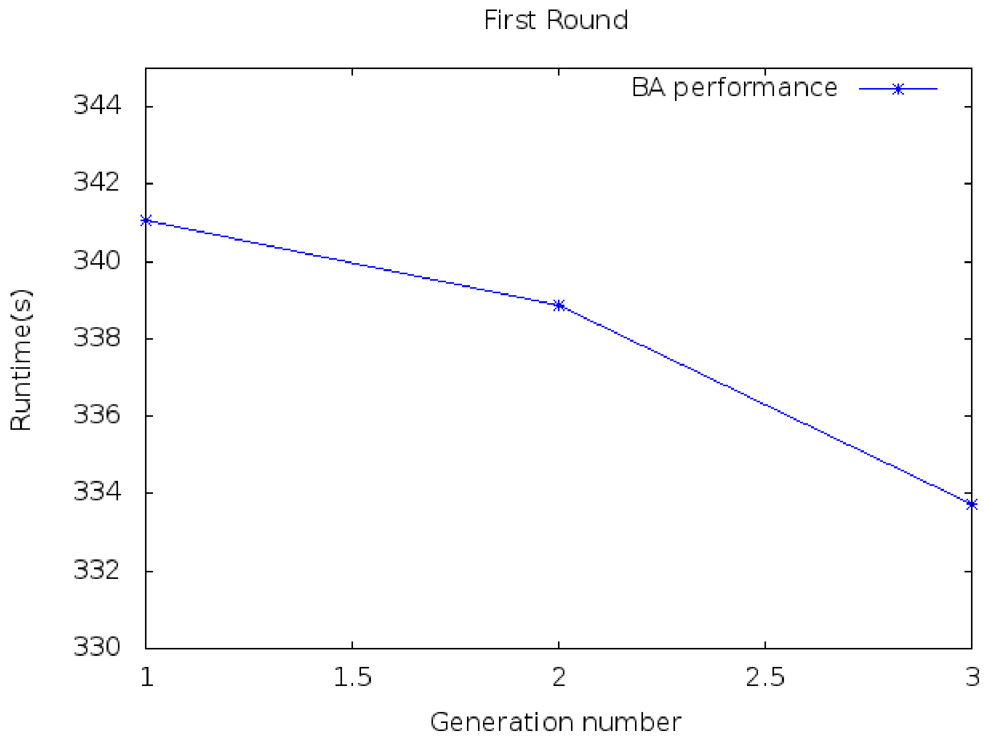


Figure 7.1: First round up to 3 generations

In 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 worst 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 50 seconds.

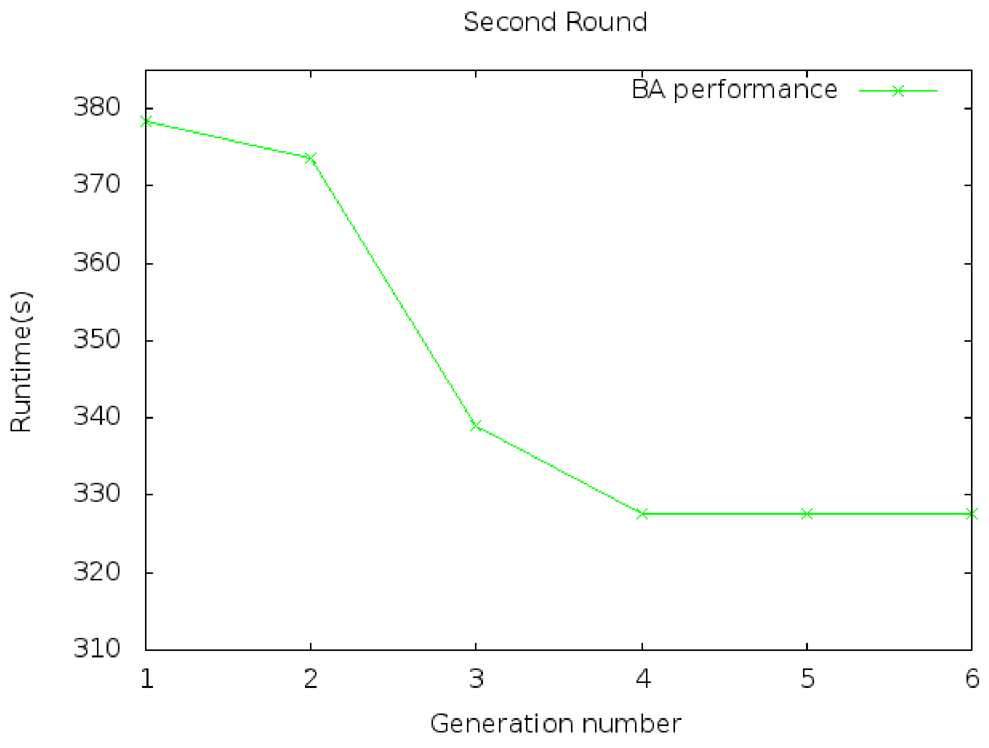


Figure 7.2: Second round up to 6 generations

In 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 till 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.

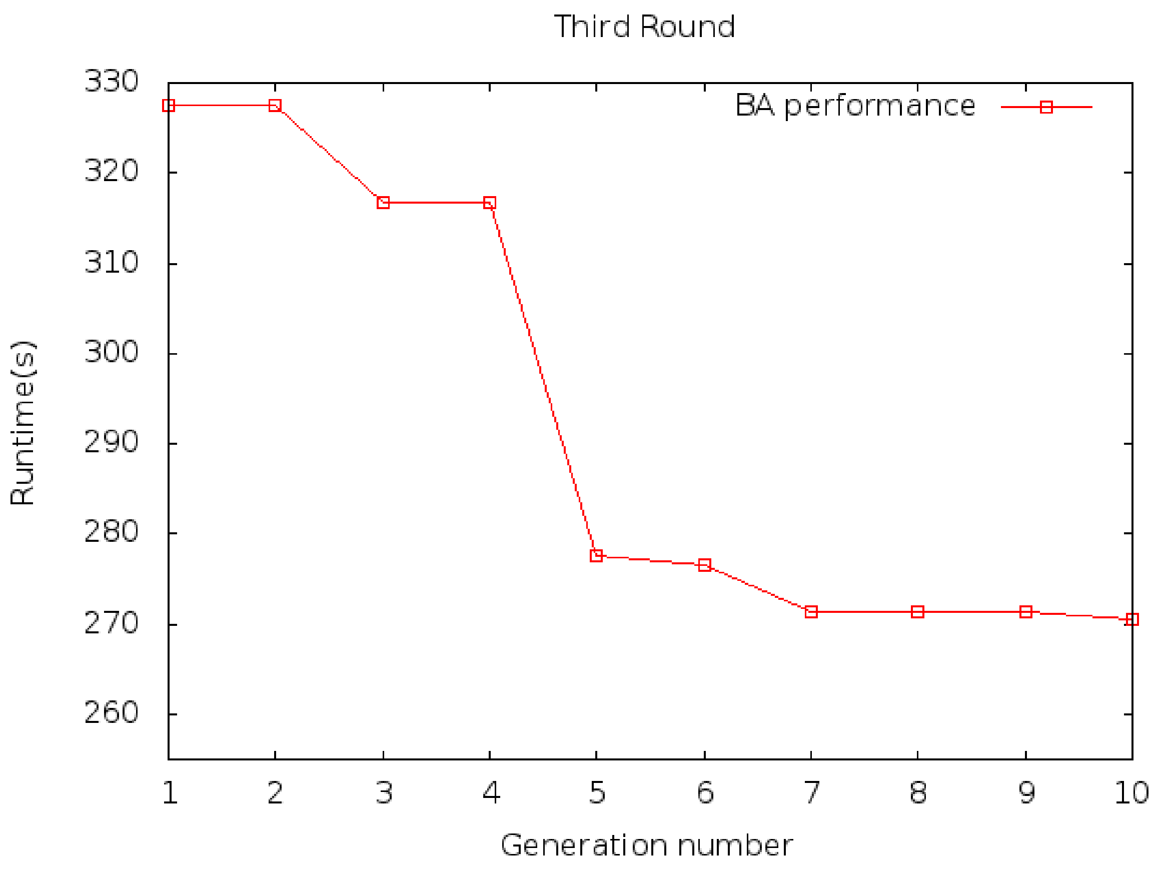


Figure 7.3: Third round up to 10 generations

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

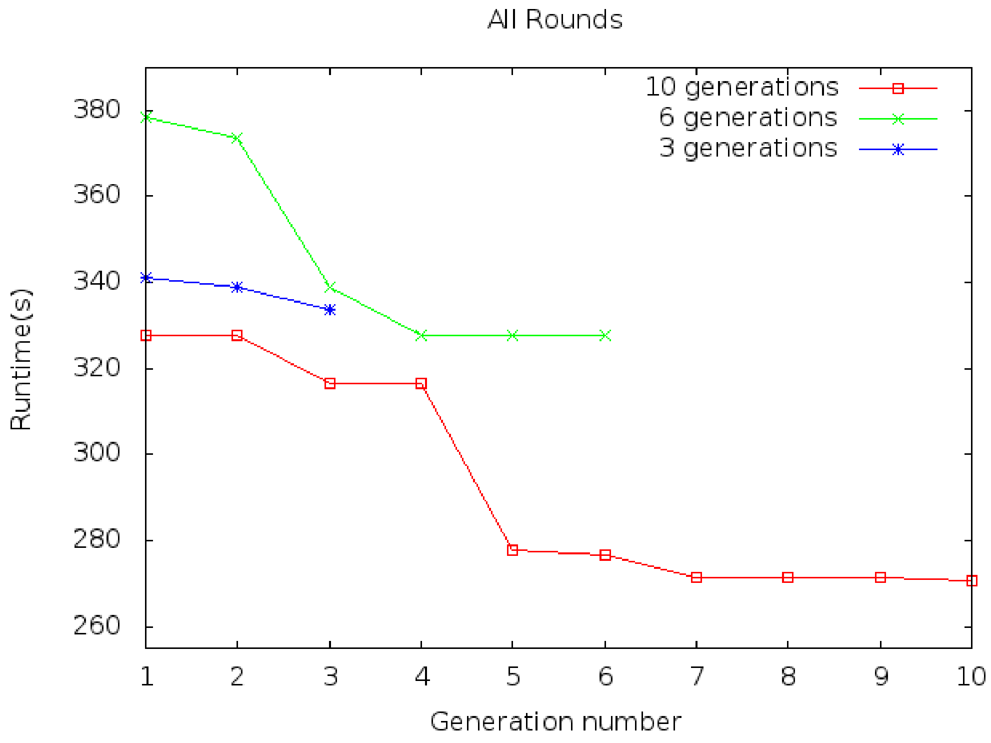


Figure 7.4: All Rounds

The graphic 7.4 is one aggregation of the all 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 percent which is a considerable improvement.

Moreover, the result achieved was influenced by the feedback provided by the software between the rounds. So, as well as the rounds pass continuous improvement occurs at runtime.

CHAPTER 8

CONCLUSION

The initial experiments shown 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 recide 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 on 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.

8.1 To do

- 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 componet 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 dinamic clusters.
- Maybe, improve the BA implementation using heuristics to orient the assignment values to the knobs.

BIBLIOGRAPHY

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

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

[3] 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.

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

[5] Yanpei Chen, Sara Alspaugh, e Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, agosto de 2012.

[6] cloudera.com. 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.

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

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

[9] 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.

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

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

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

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

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

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

[16] 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.

[17] 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 de 2011.

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

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

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

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

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

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

[24] 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.

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

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

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

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

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

[30] 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