

TIAGO RODRIGO KEPE

**A TUNING APPROACH BASED ON EVOLUTIONARY
ALGORITHM AND DATA SAMPLING FOR BOOSTING
PERFORMANCE OF MAPREDUCE PROGRAMS**

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

TIAGO RODRIGO KEPE

**A TUNING APPROACH BASED ON EVOLUTIONARY
ALGORITHM AND DATA SAMPLING FOR BOOSTING
PERFORMANCE OF MAPREDUCE PROGRAMS**

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

K38t

Kepe, Tiago Rodrigo

A tuning approach based on evolutionary algorithm and data sampling for boosting performance of mapreduce programs / Tiago Rodrigo Kepe. – Curitiba, 2013.

51f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa Interdisciplinar de Pós-graduação em Informática, 2013.

Orientador: Eduardo C. de Almeida .

Bibliografia: p. 47-51.

1. Bancos de dados. 2. Processamento eletrônico de dados - Processamento distribuído. 3. Algoritmos genéticos. I. Universidade Federal do Paraná. II. Almeida, Eduardo C. de. III. Título.

CDD: 005.75



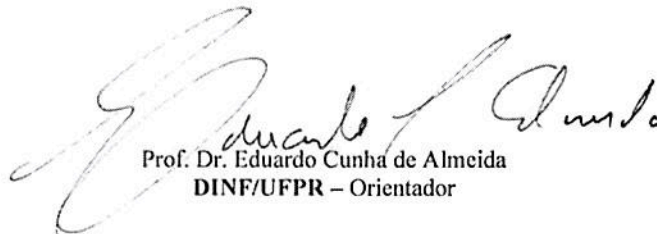
Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

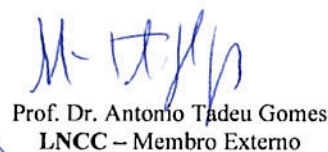
Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Tiago Rodrigo Kepe, avaliamos o trabalho intitulado, "*A Tuning Approach Based on Evolutionary Algorithm and Data Sampling for Boosting Performance of MapReduce Programs*", cuja defesa foi realizada no dia 25 de agosto de 2014, às 10:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:

aprovação do candidato. **reprovação** do candidato.

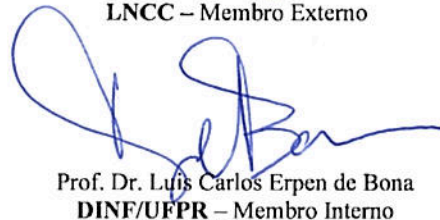
Curitiba, 25 de agosto de 2014.



Prof. Dr. Eduardo Cunha de Almeida
DINF/UFPR – Orientador



Prof. Dr. Antonio Tadeu Gomes
LNCC – Membro Externo



Prof. Dr. Luis Carlos Erpen de Bona
DINF/UFPR – Membro Interno



*Digo ao Senhor: Tu és o meu Senhor,
outro bem não possuo, senão a ti somente.*

*O Senhor é a porção da minha herança
e o meu cálice; tu és o arrimo da minha sorte.*

*Tu me farás ver os caminhos da vida;
na tua presença há plenitude de alegria,
nas tua destra, delícias perpetuamente.*

Salmos: 16:2,5,11.

Dedico esta dissertação ao Senhor, meu Deus,
que guiou todos os meus passos até esse momento.

AGRADECIMENTOS

Agradeço primeiramente a Deus, autor e consumidor da minha fé. Sou grato pela vida da minha esposa por sua compreensão, apoio e incentivo durante todo meu mestrado.

Ao Paulo Vinicius meu amável filho que foi um presente de Deus para me dar ânimo e, assim, concluir este trabalho.

Agradeço meus pais, dona Rosilene e seu Julio Kepe, pelos ensinamentos fundamentais para minha vida cristã, pelo amor e dedicação incontestáveis.

Agradeço meus segundos pais, Marlene e Paulo Dourado, que prontamente nos ajudaram inúmeras vezes, sem exitar.

Agradeço ao meu orientador Eduardo Almeida pela paciência, dedicação e valiosas dicas que conduziram à conclusão deste trabalho.

(Tiago Rodrigo Kepe)

CONTENTS

LIST OF FIGURE	v
LIST OF TABLES	vi
RESUMO	viii
ABSTRACT	ix
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Contribution	3
1.4 Outline	3
2 BACKGROUND AND RELATED WORK	5
2.1 Key-value model	5
2.2 MapReduce	5
2.3 Hadoop	6
2.3.1 Job processing	8
2.4 MapReduce related tuning techniques	10
2.5 Sampling techniques	12
2.5.1 Sampling approaches on MR	12
3 GENETIC AND BACTERIOLOGIC ALGORITHMS FOR TESTING	14
3.1 Genetic Algorithm	14
3.2 Algorithm for testing	16
3.2.1 Context transformation	17
3.2.2 Bacteriological Algorithm	18

4	SAMPLING ON HADOOP	21
4.1	Motivation for sampling	21
4.2	Sampling applied by Big Data solutions	22
4.2.1	Reservoir Sampling	23
4.3	KSample - Dynamic Reservoir Sampling Algorithm	26
4.3.1	KSample on Hadoop (Distributed KSample)	30
5	FRAMEWORK FOR TESTING	32
5.1	Framework Overview	32
5.2	Front-end	33
5.2.1	DSL Design Methodology	33
5.2.2	The DSL for tuning MapReduce programs	34
5.3	Engine	37
5.3.1	Sampler component	38
5.3.2	AutoConf component	38
5.3.3	Core component	38
5.4	Back-end	39
6	EXPERIMENTS	40
6.1	Bacteriological Algorithm against Genetic Algorithm	40
6.2	Configuration quality	42
6.3	Cost of the tuning process	43
7	CONCLUSION AND FUTURE WORK	45
	BIBLIOGRAPHY	51

LIST OF FIGURES

2.1	MapReduce framework [13].	7
3.1	Context transformation.	17
4.1	KSample's execution.	28
5.1	Framework's flow.	32
5.2	Engine processing.	38
6.1	BA against GA.	41
6.2	WordCount's configuration quality.	42
6.3	Grep's configuration quality.	43
6.4	Cost of each generation.	44
6.5	Agregate cost per generation.	44
7.1	Case to study.	46

LIST OF TABLES

2.1	MapReduce processing.	6
2.2	Regular expression example.	10
5.1	DSL design patterns [32].	34

NOMENCLATURE

BA - Bacteriological Algorithm

DBMS - Database Management Systems

DW - Data Warehouse

EA - Evolutionary Algorithms

GA - Genetic Algorithm

GPL - General-Purpose Language

HDFS - Hadoop Distributed File System

MR - MapReduce

RESUMO

O software de processamento de dados Apache Hadoop está introduzido em um ambiente complexo composto de enormes cluster de máquinas, grandes conjuntos de dados e vários programas de processamento. Administrar tal ambiente demanda tempo, é dispendioso e requer usuários experts. Por isso, falta de conhecimento pode ocasionar falhas de configurações degradando a performance do cluster de processamento. Realmente, usuários gastam muito tempo configurando o ambiente em vez de focar na análise dos dados. Para resolver questões de má configuração nós propomos uma solução, cujo objetivo é ajustar parâmetros de desempenho de programas executados sobre o Hadoop em ambientes Big Data. Para alcançar isto, nosso mecanismo de ajuste de desempenho inspira-se em duas ideias-chave: (1) um algoritmo evolucionário para gerar e testar novas configurações de jobs, e (2) amostragem de dados para reduzir o custo do processo de ajuste de desempenho. A partir dessas ideias desenvolvemos um framework para testar configurações usuais de programas e obter uma nova configuração mais ajustada ao estado atual do ambiente. Resultados experimentais mostram ganho na performance de jobs comparado com as configurações padrão e “regras de ouro” do Hadoop. Além disso, os experimentos comprovam a acurácia da nossa solução no que se refere ao custo para obter uma melhor configuração e a qualidade da configuração alcançada.

Palavras chaves: Big Data, MapReduce, Hadoop, Ajuste

ABSTRACT

The Apache Hadoop data processing software is immersed in a complex environment composed of huge machine clusters, large data sets, and several processing jobs. Managing a Hadoop environment is time consuming, toilsome and requires expert users. Thus, lack of knowledge may entail misconfigurations degrading the cluster performance. Indeed, users spend a lot of time tuning the system instead of focusing on data analysis. To address misconfiguration issues we propose a solution implemented on top of Hadoop. The goal is presenting a tuning mechanism for Hadoop jobs on Big Data environments. To achieve this, our tuning mechanism is inspired by two key ideas: (1) an evolutionary algorithm to generate and test new job configurations, and (2) data sampling to reduce the cost of the tuning process. From these ideas we developed a framework for testing usual job configurations and get a new configuration suitable to the current state of the environment. Experimental results show gains in job performance against the Hadoop's default configuration and the rules of thumb. Besides, the experiments prove the accuracy of our solution which is the relation between the cost to obtain a better configuration and the quality of the configuration reached.

Keywords: Big Data, MapReduce, Hadoop, Tuning

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

Currently, companies and research scientific institutes are awash in an ocean of data. In that context “Big Data” has emerged, and important endeavors have been put into the investigation of new technologies to handle massive data sets. Hence, those companies and research institutes are investing a lot of research efforts in distributed and parallel computing to mine “Big Data”. The key-value model has been demonstrated as a powerful solution to enable that applications enjoy data parallelism. For example, the MapReduce (MR) programming model based on the key-value model has become the industry de facto standard for parallel processing on Big Data. Attractive features such as scalability and flexibility motivate many large companies such as Facebook, Google, Yahoo and research institutes to adopt this new programming model.

The Google company in 2003 and 2004 disclosed the Google File System [19] and the MapReduce programming framework [13] for storing and processing data on large clusters. An open source implementation of the MR framework is the Apache Hadoop which is a standardized solution to handle massive data sets. Besides Hadoop, several other implementations are available: Greenplum MapReduce [36, 21], Aster Data [4], Nokia Disco [35] and Microsoft Dryad [28].

MapReduce is 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, left to the developer of a MapReduce-based application (also called a job), lies in design decisions to split the application-specific algorithm into the two higher-order functions.

Indeed, Hadoop has an intuitive interface to implement MR jobs, but on the other hand it has a complex environment which is composed of a cluster of machines, large sets of data stored into the cluster and several MR jobs. A single MR job on Hadoop has a large number of parameters to be configured, such as: memory allocation, I/O controllers, network timeouts etc. Those parameters are bound to the available resources (e.g. input data, online machines, network bandwidth, etc.). Thus, each MR job requires proper configurations to obtain good performance. A 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 [7] that adapts to the current state of the environment. We call such configuration as adaptive configuration. Therefore, data sampling can be useful to improve the testing time of new configuration parameters, instead of processing all the data set. But, generating a representative and relevant data sampling is hard and a bad sampling may not represent several aspects related to the computation in large-scale, such as efficient resource usage, correct merge of data and intermediate data.

1.2 Objectives

Our objective is to present a tuning approach for MR jobs. The success of a tuning approach for Hadoop depends on reaching a configuration suitable to the current state of the environment. Thus, new job configurations have to be explored and tested with respect to some quality criteria, for example the job response time. Normally this process is done manually. The challenge mainly lies in two issues: (1) the large amount of stored data that can exponentially increase the time needed to test new job configurations, and (2) the large number of configuration parameters to explore.

1.3 Contribution

We present an original solution to automate configuration of long-lived jobs on Hadoop. Our approach is based on the Bacteriological Algorithm [5] using the job response time as the fitness value. The BA allows to create new job configurations, but they must be tested in order to select which one is suitable for the current state of the MR environment. To achieve testing automation, it is crucial to apply techniques to efficiently reduce the amount of data to be processed. A common approach used on DBMS (Database Management Systems) is data sampling. However, in the Big Data context, data sampling is challenging due to the large amount of data: not only the sampling must be done in a distributed fashion, but also data storage. Hence, we created a novel sampling algorithm called KSample based on the Reservoir Sampling Algorithm. KSample requires as parameter a percentage of the population to be sampled, and we proved, mathematically, that it samples at least this percentage of the population. We implemented KSample on MapReduce to perform distributed data sampling on unstructured data, without knowledge of the data set size. Also, we provide a user interface to facilitate the interaction with the tuning process and data sampling through a domain-specific language (DSL).

In summary, our proposal intends to establish a framework to automate Hadoop job configuration, through the following contributions:

- an evolutionary algorithm on Hadoop context to obtain a better job configuration driven by the job execution time;
- a distributed data sampling algorithm on Hadoop clusters, considering the row as data sampling unit;
- a domain specific language for users to interact with the tuning process.

1.4 Outline

Chapter 2 introduces the fundamental concepts of the key-value model, the MR programming model and the Hadoop framework, as well some related work. Chapter 3

presents the evolutionary algorithm to choose job configurations. Chapter 4 presents the distributed algorithm to generate data sampling. Chapter 5 presents our framework with its modules. Chapter 6 discusses the experiments and results reached by our solution. Chapter 7 concludes this work bringing some topics for discussion and future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter introduces some concepts that are used in the subsequent sections: the key-value model, the MapReduce programming model and the Hadoop framework. It also discusses about some related work on tuning MR jobs and data sampling.

2.1 Key-value model

The key-value model is a simple model for data storage that forgoes any aggregation or creation of data schema. The key-value model stores data in an unstructured way, thus all details of data is done at runtime by the running jobs. It thus contrast with other models that store data intrinsically structured, such as the relational model [11], in which the simplistic notion of tables, attributes and relations define the data structure for the storage, and the hierarchical data model [40], in which the links that connect the records define a data structure. The key-value model doesn't have an elaborated data structure in order to enjoy data parallelism.

2.2 MapReduce

In 2003 and 2004 Google company disclosed the Google File System [19] and the MapReduce programming framework [13] for storing and processing data on large clusters. The MapReduce is inspired by the **Map** and **Reduce** primitives present in Lisp and Haskell. Hence, programmers can focus only on the creation of the two higher-order functions to solve a specific problem and to handle the data. The map function receives a set of $\langle key, value \rangle$ pairs and produces an intermediate set of $\langle key, value \rangle$ pairs. The framework is responsible for aggregating all intermediate values which share the same key, putting them in a list, forming new pairs $\langle key, list(values) \rangle$ and passing them to the

reduce function which processes these values in order to form a new smaller set of values:

map	$k1, v1$	\rightarrow	$set(k2, v2)$
reduce	$k2, list(v2)$	\rightarrow	$set(v3)$

Table 2.1: MapReduce processing.

When the map results are already available in memory, a local function *Combiner* may be used for optimization reasons. Such function combines all values for a given key, 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 runs map functions. The Combiner may be seen as a *mini-reduce* function, which operates only on data generated by one machine.

We define the MapReduce as programming model and its implementation as the MapReduce framework, this solution allows many database processes to be written in a simple way, [29]. Vast amount of data is splitted and assigned to a set of computers, called computer cluster, to improve performance through parallelism. The goal is to reduce the complexity, thus users can focus on the main problem that is the data processing. The MR framework, see Figure 2.1, delegates nodes in the cluster to execute the map and reduce functions, splits the input files, and distributes the files to the respective nodes. The map computing is run in parallel. As mappers terminate their outputs are gathered to form the intermediate $\langle key, value \rangle$ pairs. After that the reducers receive those pairs as input and run the reduce function (the reducers also operate in parallel). All steps are orchestrated by a master machine.

2.3 Hadoop

Recently, the MapReduce programming model have been implemented by several frameworks such as Greenplum MapReduce [36, 21], Aster Data [4], Nokia Disco [35], Microsoft Dryad [28], and Hadoop [25].

The introduction of the MR framework boosted the creation of an open source alternative to Google's solutions. The most popular open source MapReduce implementation

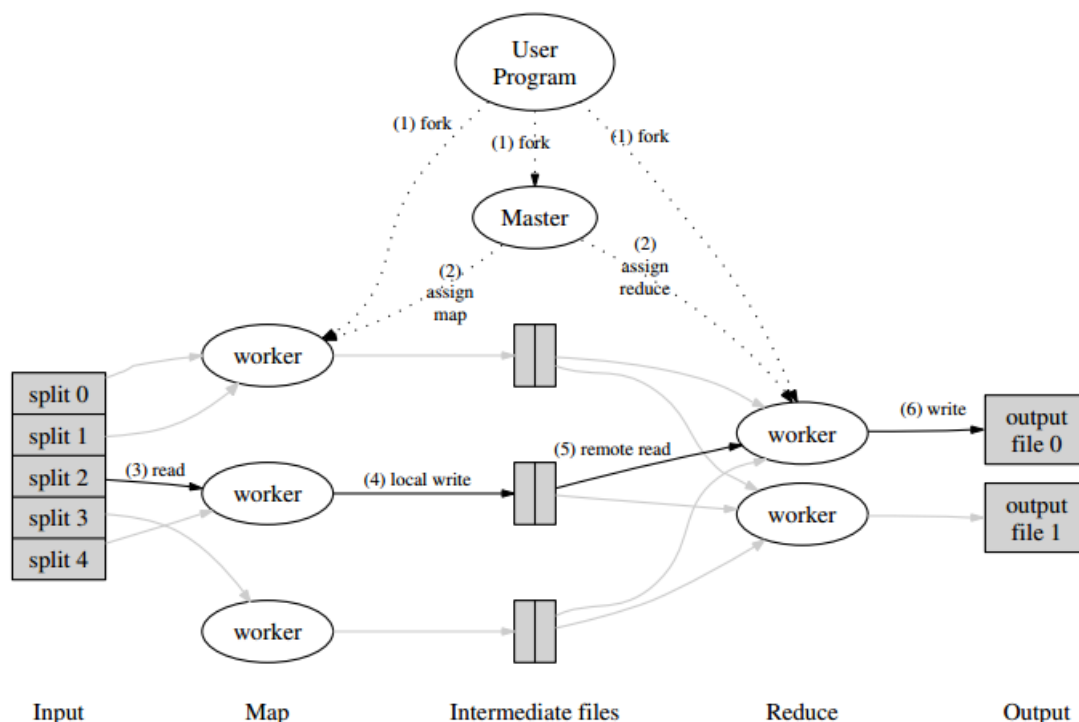


Figure 2.1: MapReduce framework [13].

is the Apache Hadoop framework. It implements an engine of MapReduce and its own system file called Hadoop Distributed File System (HDFS). Hadoop is a framework for reliable, scalable and distributed computing. It provides an intuitive interface to create MR programs (the jobs), by defining the map and reduce functions (the tasks). It was designed to allow users to focus on the implementation of those functions, without worrying about issues involving the distributed computing, such as: file splitting, replication, fault tolerance and task distribution.

Hadoop is composed of two main components:

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

The HDFS stores all files in blocks. The block size is configurable per file, and all blocks of a file must have the same block size except for the last block. It is divided into two components: the *NameNode* and the *DataNode*. The NameNode is placed in the master machine. It stores all the metadata and manages all the DataNodes. The DataNode stores the data. When a DataNode starts it connects to the NameNode, then

responds to requests from the NameNode for file system operations.

The MapReduce engine is responsible for parallel processing. It is composed of a *JobTracker* which lies into the master machine, and several *TaskTrackers* which lie into the slave machines, also known as workers. The JobTracker receives job submissions from users, and it designates TaskTrackers to compute map and reduce tasks with its respective input blocks. A worker that processes a map task is called mapper and a worker that processes a 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 which implements the map and reduce functions. Users submits jobs to the master machine with its input directory stored into the HDFS which contains all files to be processed (inserted previously in the HDFS). Hadoop respects the premise minimal data motion which consists in moving the computation where the data are stored, thus reducing network I/O to ensure local disk I/O as much as possible. The data distribution amongst nodes occur while the files are stored into the HDFS, whether new data nodes are added to the cluster Hadoop doesn't rebalance the data automatically, but it provides the balancer tool to perform data rebalance manually.

Thus, the master requests information to the NameNode about the blocks and file locations, and then deploys copies of the job across several workers. With the block information the map task is scheduled to a set of workers with its respective input blocks, respecting the feature minimal data motion. Mappers process each input blocks, generate key/value intermediate pairs and append them to intermediate files. When the mapper instance terminate it notifies the master who splits the intermediate files in blocks and shuffles them for the reducers to process. When all reducer instances terminate the processing, they append their results to the final output file.

A well known example of a MapReduce job is the Grep application, present in Listing 1, which receives as input several textual documents and generates as 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 patterns in the files. The reducer is responsible for summing the number of occurrences for each pattern.

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 purposes). The body of the method uses the class **Pattern** to describe a desired pattern, and the class **Matcher** to find this pattern. For each pattern match found the pair $\langle matching, 1 \rangle$ is emitted 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 [25].

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 purposes). 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 matchings 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 [25].

An example of inputs and outputs of both functions when applied to a simple sentence is presented in Table 2.2. We applied the following regular expression:

“**[a-z]*o[a-z]***”, which 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 2.2: Regular expression example.

2.4 MapReduce related tuning techniques

Simulators have been used to predict job behaviors. The MRPerf [43] is one simulator to comprehend the MapReduce sensibility of job performance while applying platform parameters, network topology, node resources and failure rates. It was implemented using Network Simulator 2 (NS-2) for network emulation, and the DiskSim simulator for advanced disk simulation. Another simulator is the WaxElephant [39] proposed to build a more complex MR simulation environment based on the following capabilities: (1) load real MapReduce workloads derived from logs belonging to production Hadoop clusters, and replaying the job for these workloads; (2) auto configure Hadoop parameters which

affect the job performance; (3) executing job simulation at the task-level; (4) comparing different job scheduling policies in Hadoop; and (5) simulating and analysing Hadoop clusters.

Simulators emulate big data environments, thus they can be used to find job proper settings. Nevertheless, they are based on models to predict job behavior using information provided by users, existing workloads, log files and other relevant sources. Since, models try to abstract away from details of systems, they may present representativeness limitations, by not simulating events that only happen in the real world.

Another solution branch is profile-driven tuning, as used in [24, 26]. The Starfish system generates statistical summaries from a MR job execution using dynamic instrumentation. This is a technique that injects additional bytecodes in runtime to monitor specific Hadoop Java classes. Job profiles are generated from resultant statistics of the monitoring data during full or partial job execution. New profiles are generated from existing ones using estimation techniques based on modeling and simulation of MapReduce job execution [26]. The profile contains summary information about the runtime behavior of the job being tuned, and it is assumed to come from a previous execution of the same job. Another technique correlated to profiling was proposed by Popescu et al. [37], in which statistics from input data and log information about a prior job execution are used for predicting job behavior using machine learning models.

Profile-driven tuning requires a previous job execution. When a new job is submitted, Starfish runs it and uses dynamic instrumentation to collect statistical data which may cause overload in the job execution. Furthermore, statistical data might not reflect the real data and may contain biases. For instance, they may have been collected from a partial job execution at the moment that it was processing files composed of large lines, but most of job input files might be composed of short lines. Hence the statistical data are biased because they don't represent the real data, thus some IO and memory controllers may be misconfigured.

An alternative is the rules of thumb which have been created to adjust Hadoop environments. They were created based on administrators and developers knowledge [44, 27,

8, 10]. This approach is simple and fast to be applied, but not individually accurate, because these rules are generic, aiming to be applied in all jobs without considering specific job behaviors.

2.5 Sampling techniques

Data sampling is a popular approach to improve analysis of huge data sets. It has been applied over relational databases to infer information on entire data population. One of the most used data sampling techniques is Random Sampling [38], which consists in selecting a pre-determined amount of data randomly. In the literature there are several other techniques such as Stratified Random Sampling, which splits data into strata where each element in a stratum has the same chance of being selected [12]. Another technique is Systematic Sampling, which consists in firstly selecting randomly one element of the population, then from this element the next K elements are selected in sequence, the number K may be either chosen randomly or based on some criteria. The systematic process continues until the sample is completed [17].

2.5.1 Sampling approaches on MR

This section presents data sampling approaches on Hadoop for different purposes. All of them performs data sampling on Big Data, considering aspects related to distributed computing and storage, which aspects we are interested.

BlinkDB is a query engine for running interactive SQL queries on large volumes of data. It focuses on running short exploratory queries to provide trade-off query accuracy for response time [1]. To achieve that goal BlinkDB is composed of two main modules: (i) Sample Creation, which creates sets of stratified samples based on past queries behaviors. The idea is to cache samples for future queries considering query history. (ii) Sample Selection chooses dynamically the best sample that satisfies query's error/response time constraints.

Liven and Kanza [31] designed a distributed algorithm based on the stratified sample

technique using MR. It aims to create multi-survey stratified samples over online social networks, considering specific constraints and costs to share individuals among surveys. The reduce function implements the unified-sampler algorithm and performs the selection of elements from strata created by the map and combiner functions. The unified-sampler algorithm depicted requires the population size from where the stratum was extracted.

EARL [30] is a framework for Hadoop to run queries on samples, thus reducing the query response time constrained by user thresholds. The framework is based on the bootstrapping sample technique, which consists in creating samples following a error threshold. If the current sample achieves a high error, then a new sample is created with an increased sample size. This process continues until reaching user thresholds. That technique is known as resampling. On each created sample a function of interest is computed in order to estimate accuracy. To generate samples the algorithm depicted needs the sample size and the number of resamples.

Those data sampling approaches require prior knowledge about the data or the query history. BlinkDB needs the queries history to build the stratified samples, the unified-sampler algorithm presented by Liven and Kanza requires the population size, and EARL requires the sample size and the number of resamples. However, for our data sampling approach we are interested in using naive input parameters in order to abstract the complexity away from the users, like a percentage of the data population.

CHAPTER 3

GENETIC AND BACTERIOLOGIC ALGORITHMS FOR TESTING

In this chapter we firstly present the Genetic Algorithm which is the base for the Bacteriological Algorithm (BA) used to generate and select Hadoop's job configurations. Second, we present the BA and a context transformation to accomplish it on Hadoop's context.

3.1 Genetic Algorithm

The Genetic Algorithm (GA) is a search heuristic which mimics the process of natural selection. It belongs to the large family of the Evolutionary Algorithms (EA) . The GA has been adopted to solve problems in many fields involving search, optimization and machine learning [20]. It generates solutions by applying evolutionary mechanisms or operators, such as reproduction, crossover and mutation which aim to adapt individuals to a certain environment.

GA is an iterative algorithm that starts from an initial population of candidate solutions (individuals) evolving this population toward a better solution. In each iteration of the algorithm the fitness of every individuals are computed. The fitness is a specific function belonging to the optimization problem and drives the GA toward a better solution. Then the natural operators are put in practice to generate a new population, which also is called a new *generation*. The fitness of the previous individuals are evaluated, and the algorithm terminates by reaching a desired fitness or a certain number of generations.

The natural operators perform changes on individuals in the gene level, a well known example of individual is a string composed of the characters 0s and 1s, and a gene is a specific character. The GA process is implemented by the Algorithm 1 and the natural operators are explained below:

- **Reproduction:** copies the individuals to participate at the next stage (the crossover step). They are chosen based on their ability to adapt to the environment, which can be calculated according with a function $F(x)$, also known as the fitness of the individual. The individual's fitness measures how much that individual is adapted to the environment.

The reproduction process aims to fill an empty population that will be the next generation. Based on the previous generation the best individuals are selected to form a set of prospective individuals that may be propagated to the next generation. New individuals are randomly chosen from the set of prospective individuals and copied to the new population. The addition of a new individual is influenced by the fitness value, i.e., as the individual fitness is greater as its number of copies tends to be greater into the next generation.

For example, the reproduction could be similar to create a roulette whose size is the sum of the fitness values of all prospective individuals. Each prospective individual receives roulette's slots according to its fitness value. For instance, the set of prospective individuals is composed of the individuals A and B whose fitnesses are $F(A)=10$ and $F(B)=20$, hence the roulette's size is $F(A) + F(B) = 30$, with A receiving 10 slots and B 20. Thus, the probability to clone A is $\frac{10}{30} = \frac{1}{3}$ and B is $\frac{20}{30} = \frac{2}{3}$, as the individual fitness is greater as its number of copies tends to be greater. The roulette wheel is spin and an individual is cloned according to the slot that the roulette stopped, and that individual is added to the new population. The roulette continues to spin until filling the next generation.

- **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. The genes of two individuals are genetically combined to generate another resultant individual, thus the new individual inherits characteristics of both parents. More precisely, in the genetic algorithm two individuals are chosen randomly (A, B), and an integer k between 1 and the individual's

gene number (n) is also 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:** occurs after the crossover on genes of the new individuals. The natural process consists basically in changing 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).

Algorithm 1: Genetic Algorithm

Input : *Pop* Initial population
Input : *NumberGen* Number of generations
Input : *Fitness* Desired fitness value
Output: *BestIndiv* The best individual reached

```

repeat
  for each indiv ∈ Pop do
    CalcFitness(indiv)
  Reproduction(Pop)
  Crossover(Pop)
  Mutation(Pop)
until NumberGen ∨ Fitness ;
BestIndiv ← getBestIndividual(Pop)
return BestIndiv

```

3.2 Algorithm for testing

To perform tuning of MR jobs we inspired on the Bacteriological Algorithm (BA), which is an adaptation of the GA to improve its convergence, and was empirically proved that the BA has a greater convergence than the GA, according with Baudry et al [5]. Thus that is the reason for us in adopting the BA. The BA implemented by Baudry et al [5] works at the individual level, our contribution was to adapt the BA on the gene level. Therefore, our BA implementation has the gene as the atomic unit, a group of genes forming an individual (bacterium), and a group of individuals forming a population (bacteria).

3.2.1 Context transformation

Our context is focused on the Hadoop environment which has its particularities. Thus, a context transformation is mandatory to implement the BA on Hadoop. The transformation is based on the following definitions:

Definition 1 (*Knob*): a specific Hadoop's configuration parameter such as: `mapred.reduce.tasks`.

Definition 2 (*Set of Knobs*): a set of Hadoop's configuration parameter to set up a MR job.

Definition 3 (*Sets of Knobs*): different configuration alternatives to set up a job (composed of one or more set of knobs).

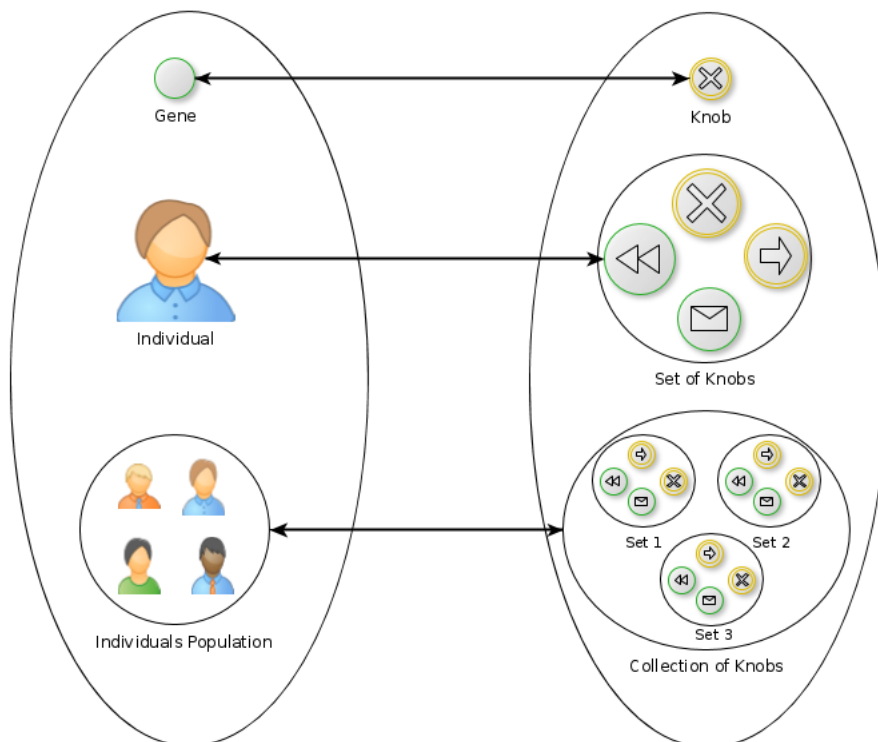


Figure 3.1: Context transformation.

In the context transformation, each component of genetic context was translated to one component of Hadoop environment. Figure 3.1 shows that a gene is transformed

in a knob, an individual (which is a set of genes) is transformed in a set of knobs, and an individuals population is transformed in sets of knobs. An interesting characteristic of that transformation is its bijection property, a component in the genetic domain is translated to another component in the Hadoop domain. Besides, the transformation also has an inversion property, i.e, all components in Hadoop domain can be translated to the respective components in genetic domain.

3.2.2 Bacteriological Algorithm

The BA was implemented by Baudry et al. [5] to improve the convergence of the GA. It introduces a new mechanism called Memorization that is responsible for memorizing the best individuals created along the generations. This new mechanism might appear as a small modification, but actually reflects a crucial change on GA behavior. Besides, the crossover operator is absent from BA because of the peculiar biological behavior of the bacteria. In terms of natural process a bacterium reproduces itself asexually, consequently there is not crossover between two individuals, because the reproduction process consists in duplicating the DNA of a bacterium followed by a division to form two new bacteria.

Our BA's implementation has four mechanism: Fitness computation, Memorization, Reproduction and Mutation:

- **Fitness computation:** the fitness as in GA is one way to differentiate the abilities of each individual to adapt to the environment. The calculation depends on several criteria defined by the programmer, and it is used to select the best individuals for the next generation. We used as fitness value **the job execution time**, i.e. the job is configured with a configuration generated by the BA and it runs on the cluster, after finishing its execution time is assigned to the individual's fitness value.
- **Memorization:** is the main mechanism introduced by the BA. It is responsible for memorizing the best individual generated by the process of adaptation. As the process continues, the population improves more quickly its capacity of adaptation. If a generation generates bad individuals, i.e. individuals with low fitness values, it

means individuals couldn't adapt well to the environment, then the memorization operator ignores this generation, and uses the best individual from past generations to the next generation in order to avoid regressions in the process.

- **Reproduction:** a new empty population is created, the best individual is cloned as many times as the last generation size and put into the new population. This population is then passed for the mutation operator.
- **Mutation:** is responsible for generating new individuals. Each individual created for the reproduction operator is mutated. The mutation operator changes one or several genes in order to improve the adaptation of the bacteria population to the environment. These new individuals are evaluated by their fitness against the fitness value of the overall best individual.

The algorithm in high-level of abstraction is described by the Algorithm 2. It requires as input an initial sets of knobs, the number of generations to be reached and the desired fitness value. The initial population could be composed of the latest job settings, and other settings of interest. The number of generations is freely defined by the user, and the desired fitness value could be the last job execution time. For each set of knobs belonging to the current population is calculated its fitness value (in our case the fitness value is the job execution time when configured with this set of knobs), after the best set of knobs reached is memorized, then the reproduction creates a new population composed of clones of the best set of knobs. The mutation operator interacts with the new population, and for each set of knobs one or more knobs are randomly selected and changed. The bacteriological process continues until the number of generations or the desired fitness value is reached.

Algorithm 2: Bacteriological Algorithm

Input : *Pop* Initial population

Input : *NumberGen* Number of generations

Input : *Fitness* Desired fitness value

Output: *BestIndiv* The best individual reached

repeat

for each *indiv* \in *Pop* **do**

 | CalcFitness(*indiv*)

 Memorization(*Pop*, *BestIndiv*)

Pop \leftarrow *Reproduction*(*BestIndiv*)

Pop \leftarrow *Mutation*(*Pop*)

until *NumberGen* \vee *Fitness* ;

return *BestIndiv*

CHAPTER 4

SAMPLING ON HADOOP

In this chapter we present in the Section 4.1 our motivation and challenges for data sampling on Big Data. Section 4.2 shows some data sampling methods already applied by Big Data solutions. Section 4.2.1 introduces the reservoir sampling method. In Section 4.3 we present our dynamic reservoir sampling algorithm (KSample), and Section 4.3.1 shows the distributed KSample implemented using the MR programming model.

4.1 Motivation for sampling

One of the most highlighted property on Big Data environments is the massive amount of data that is handled. The data is gathered in a continuous and frequent flow, and new data might constantly comes from new sources in new formats. There are challenges not only in volume, but also in variety and velocity in how the data arrive. These challenges may affect the job behavior causing different patterns of resources usage. For instance, new data formats might lead the job to make more IO operations, hence the IO controllers configuration needs to be adjusted. This kind of issue is also very challenging from a testing point of view [7].

The BA process allows to create new configurations, but they must be tested in order to select which one is suitable for the current state of the MR environment. To achieve testing automation, it is crucial to apply techniques to efficiently reduce the amount of data to be processed. A common approach used on DBMS is data sampling. However, in the Big Data context, data sampling is challenging due to the large amount of data: not only the sampling must be done in a distributed fashion, but also data storage.

Therefore, we created a data sampling algorithm robust enough to be distributed and to work on unstructured data. The data sampling is essential to improve the BA's response time. Whilst the BA generates new settings along the execution, our data

sampling approach reduces the cost to test these settings.

4.2 Sampling applied by Big Data solutions

On Big Data environments there are some implementations of data sampling. One example is MonetDB which is a column-oriented database management system designed to hold data in main-memory, and a scalable solution to process large sets of data [34]. This DBMS supports data sample and uses the algorithm A, which is based on a random sample technique [33].

Algorithm A selects n records from a data set of size N such that $0 \leq n \leq N$. For each record that will be inserted in the sample, it chooses randomly a number V that is uniformly distributed between 0 and 1. Based on V , n and N , a number s is calculated. A set of records called S is created from the s first records of the data set. From S a record is randomly chosen and put into the sample. After, the records present in S are skipped from the data set, and this process continues until completing the sample [42]. That algorithm behaves as the stratified random sampling. The creation of the set S can translate to a stratum that contains neighbors records where a record is chosen.

Hive is another database management system that performs data sampling based on random methods. It was created to manage the data stored on Hadoop, allowing ad-hoc queries (which are translated to Hadoop MR jobs), data summarization, and analysis of large data sets. Thus, Hive is considered a Data Warehouse (DW) for Hadoop [3]. It samples at row or block size level. The row level consists in choosing randomly the rows according with the column name. If the column name is not defined, then the entire row is selected. If it is defined the choice can be driven by the Bucketized Table in which the sample is done only on the buckets that contain the specified column [2]. The block size sample is also performed randomly 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 stores the Hadoop data as a data warehouse suiting queries submitted by users. Moreover, the clustering by bucket and block size requires a prior structuring of

data, hence several information about the data are previously known by Hive.

4.2.1 Reservoir Sampling

Hadoop adopts the key-value model to store data. Initially, pure data are stored, i.e., the data are stored without any handling. The data are inserted into files, where the key is a sequential line number and the value is the own line content. There is no predefined schema on Hadoop. Thus, it stores unstructured data, hence is challenging to develop data sampling methods on it. According to [41, 9, 23] that challenge about unstructured data stream can be addressed with Reservoir Sampling.

The Reservoir Sampling algorithm is also a random algorithm. It aims to process a stream of items of large and unknown length, randomly it chooses item(s) from this stream, each item is equally likely to be selected and it has to be iterated only once [9], i.e. it randomly chooses k elements from a stream containing N items which is either unknown or too large to fit in memory.

To understand the solution an example can clarify the idea. Suppose we have a reservoir sampling of size equal 1 (i.e. $K = 1$), and we have to get one item such that all items have the same probability to be chosen.

In the first round, when the first item comes, the probability is $P(1^{st})^{1stRound} = 1$, because the stream length is 1 at the moment and we do not know if the stream finished. Thus, the first element always will be caught. When the next element comes (2^{nd} item), the first element has been holding and we need to choose in continuing to hold it or replacing the 1^{st} by the 2^{nd} . So, the probability in choosing the 2^{nd} element is $P(2^{nd}) = \frac{1}{2}$ because the stream length is 2 at the moment. On the other hand, the probability $P(1^{st})^{2ndRound}$ to continue holding the 1^{st} is the probability in choosing it in the last round multiplied by the probability of not choosing the 2^{nd} , which is $P(1^{st})^{2ndRound} = P(1^{st})^{1stRound} \times \overline{P(2)} = 1 \times (1 - P(2)) = 1 \times (1 - \frac{1}{2}) = 1 \times \frac{1}{2} = \frac{1}{2}$. Thus, the probability of the 1^{st} and the 2^{nd} element in the second round is $\frac{1}{2}$.

In the next round, when the third element comes, we need to decide if we continue holding the chosen element in the last round or if we'll 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 that it is the first element, its probability in the third round is the probability of it in the second round multiplied by the probability of not choosing the third element: $P(1)^{3rdRound} = P(1)^{2ndRound} \times \overline{P(3)} = \frac{1}{2} \times (1 - P(3)) = \frac{1}{2} \times (1 - \frac{1}{3}) = \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$. 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 N th round the probability of all elements is $1/N$. We can prove that idea by induction using a reservoir of K size:

- We want to prove that the probability of any element in being in the reservoir after N rounds is $\frac{K}{N}$.

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

The probability of the K^{st} elements is $P(K^{st}) = \frac{K}{N} = 1$.

- *Induction Hypothesis (I.H.):*

Suppose the probability of the N^{st} elements in the round N^{th} is

$$P(N^{st}) = \frac{K}{N}, \quad N > K.$$

- *Induction Step:* round $N + 1$.

$P((N + 1)^{th}) = \frac{K}{N+1}$ is the probability in choosing the $(N + 1)^{th}$ element to put into the reservoir, because the reservoir size is K and the stream length is $N + 1$ at the moment.

The probability to remove an element (E_{remove}) already inserted in the reservoir is:

$$\begin{aligned} P(E_{remove}) &= (\text{the probability in choosing the } (N + 1)^{th} \text{ element}) \\ &\quad \times (\text{the probability in removing an element} \\ &\quad \text{of the reservoir}) \end{aligned}$$

$$P(E_{remove}) = P((N + 1)^{th}) \times \frac{1}{K} = \frac{K}{N + 1} \times \frac{1}{K} = \frac{1}{N + 1}$$

Hence, the probability of any element kept in the reservoir:

$$P(E_{keep}) = 1 - P(E_{remove}) = 1 - \frac{1}{N+1} = \frac{N}{N+1}$$

Thus, the probability of any previous element being in the reservoir after the round $(N+1)^{th}$ is:

$$\begin{aligned} P(E_{being}) &= (\text{probability of any previous element} \\ &\quad \text{has been chosen in the last } N^{st} \text{ rounds}) \\ &\quad \times (P(E_{keep})) \\ P(E_{being}) &= \frac{K}{N} \times \frac{N}{N+1}, \text{ by the I.H.} \\ P(E_{being}) &= \frac{K}{N+1} \end{aligned}$$

Then, the probability of the $(N+1)^{st}$ elements in the round $N+1$ is $\frac{K}{N+1}$.

An implementation of the reservoir algorithm is presented in the Algorithm 3. The goal is to build a reservoir which is smaller than the memory. It receives as parameter the number K that is the resultant sample size and the data *stream* that constantly receives new data. Initially, the resultant sampling is assigned with the first K elements. Then the algorithm aims to calculate the probability of the i^{th} element to be inserted into the reservoir starting from the $(K+1)^{th}$, that probability is $P(i^{th}) = \frac{K}{i}$. After a random number (*rand*) uniformly distributed between 0 and 1 is chosen, if $rand < P(i^{th})$, then the i^{th} element is added to a random position in the resultant sample.

Algorithm 3: Reservoir Sample Algorithm

Input : k : sample size

Input : $stream$: data stream with undefined length

Output: $reservoir[k]$

for $i = 1 \rightarrow k$ **do**
 | $reservoir[i] \leftarrow stream[i]$
 $numElements \leftarrow k$

while $stream \neq EOF$ **do**
 | $numElements \leftarrow numElements + 1$
 | $probability \leftarrow k/numElements$
 | $rand \leftarrow Random(0,1)$
 | **if** $rand < probability$ **then**
 | | $pos \leftarrow Random(1, k)$
 | | $reservoir[pos] \leftarrow stream[numElements]$

return $reservoir$

4.3 KSample - Dynamic Reservoir Sampling Algorithm

As defining the reservoir sample size (i.e. the number K) is hard on unstructured data, because the large amount of data may hinder choosing a representative reservoir sample. Therefore, our approach consists of receiving a percentage as parameter of the population to be sampled. For instance, 10% is received as input, then our algorithm will create a reservoir sample that represents 10% of the input data records.

KSample works on unstructured data, the atomic unit for the algorithm freely can be defined such as: record, row, byte, file, etc. The K-Sample is inspired on the reservoir sample technique and works with an undefined reservoir size.

Algorithm 4: K-Sample Algorithm

Input : *percentage*: percentage for sampling
Input : *stream*: data stream with undefined length
Output: *reservoir*[]
 $sLength \leftarrow 0$
 $slotRound \leftarrow 0$
while *stream* \neq *EOF* **do**
 $sLength^{++}$
 if *reservoir.size()* $<$ (*percentage* \times *sLength*) **then**
 reservoir.newSlot()
 $slotRound \leftarrow 0$
 $slotRound^{++}$
 $probability \leftarrow \frac{1}{slotRound}$
 $rand \leftarrow \text{Random}(0, 1)$
 if $rand \leq probability$ **then**
 $reservoir.currentSlot \leftarrow stream[sLength]$
return *reservoir*

The Algorithm 4 receives a percentage number to represent the population (which is converted to the interval $]0, 1]$) and the data stream. It starts with an empty reservoir. As needed a new slot is added to the sample reservoir, thus the reservoir grows dynamically and on demand. If the reservoir size is less than *percentage* multiplied by the stream length (*sLength*), then a new slot is created into the reservoir, and new elements will be addressed for this slot following the Algorithm 3 (with the reservoir size equal 1) until a new slot being created. We thus treat any slot as a mini-reservoir with $K = 1$.

The figure 4.1 shows an example of the KSample's execution with a sample percentage of 30%. In the first round, when the first element (*E1*) comes, a new slot (*slot1*) is created and *E1* fills it. In the second round is not needed to create a new slot because the reservoir is containing at least 30% of the population, then *E1* is replaced by *E2*. In the third round also is not needed to create a new slot, so *E3* is discarded. When *E4* comes KSample detects that is needed to create the *slot2* because if it was not created the reservoir will not contain at least 30% of the population. After that, *E4* is assigned to the *slot2*. Then *E5*

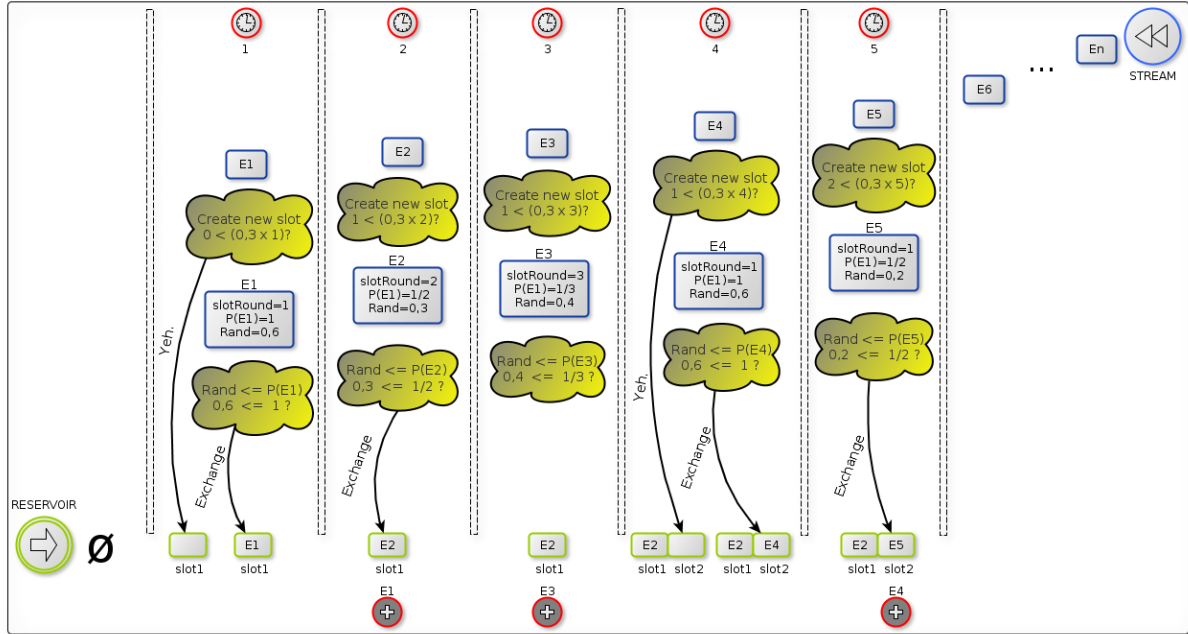


Figure 4.1: KSample's execution.

comes and replaces E_4 , and the process continues until all elements arrive. The diagram contains every statistical calculations, and conditional decisions to create a new slot and to replace elements inserted in the reservoir.

KSample is based on the fact that the reservoir always holds at least the *percentage* of elements, in our execution example 30%. That is the invariant property of the KSample, i.e. independent of the current stream length, it ensures that at any step the reservoir will always hold at least the *percentage* (as example 30%) of elements that arrived. Also, the KSample ensures that every element has the same probability to be inserted into the current slot, that is inherited from the reservoir algorithm as can see in the prove of Section 4.2.1, as long as it treats any slot as a mini-reservoir of size 1. We can prove the invariant property by proving that the KSample creates a new slot in the right moment, i.e. if the KSample didn't create a new slot, the reservoir wouldn't contain less than the *percentage* of elements from the stream. Let's prove that property by induction:

- *Notations:*

R: reservoir size.

P: percentage of the stream.

L: stream length.

- *Base Case:* When E_1 comes.

$R = 0$, the algorithm has to decide in creating or not a new slot, for this it checks the condition ($R < (P \times L)$).

As $P \in]0, 1]$ and $L = 1$, then $(P \times L) \in]0, 1]$.

Consequently, $R < (P \times L)$ is true and a new slot is created, thus the reservoir will hold E_1 and it will have at least P percentage of elements from the stream.

- *Induction Hypothesis (I.H.):*

Suppose in the step $\#N$ after the E_n element arriving the reservoir holds at least P percentage of elements from the stream.

- *Induction Step:* step $\#(N + 1)$.

We have to prove two cases:

1. *Create a new slot:*

For this case the condition: $R < (P \times L)$ has to be true. By the I.H., in the last step (step $\#N$) the reservoir holds at least $P\%$ of elements from the stream, as the KSample will create a new slot and it will hold the element E_{n+1} (following the reservoir sample algorithm), then, certainly, adding a new slot the reservoir size will increase, thus it will continue holding at least $P\%$ of elements from the stream.

2. *Don't create a new slot:*

The KSample decided not creating a new slot, then $R \geq (P \times L)$, means that the reservoir is holding $P\%$ or more elements from the stream, otherwise would be the case 1.

Therefore, after the step $\#(N + 1)$ the reservoir size is invariant in any round of the KSample, because it will contain at least $P\%$ of elements from the stream.

For any slot the KSample follows the Algorithm 3 with a reservoir size 1. For each element which competes by the same slot, its probability in keeping in this slot is the same. Globally, the probability of all elements might be different, one example could be the last slot, if the stream length is odd and the sample percentage is 50%, then the last element will insert in the last slot with probability 1, all other previous elements were inserted into the reservoir with probability $\frac{1}{2}$. However, that fact isn't a problem in our context, i.e. Big Data, because we expect to sample large data sets, and, among countless elements, an unique element must not cause major impacts.

4.3.1 KSample on Hadoop (Distributed KSample)

We built the KSample on Hadoop to take advantage of its architecture for distributed computing and storage. We considered a row as atomic unit for data sampling. Algorithm 5 depicts the map function. It receives as input a set of files (Δ) stored in the cluster. The map processes each file (δ) $\in \Delta$, and for each line (ℓ) $\in \delta$ a random number (Γ) uniformly distributed between 0 and 1, i.e. $\Gamma \in]0, 1]$, is selected and an intermediate $\langle \Gamma, \ell \rangle$ pair is emitted.

The reduce function, depicted in Algorithm 6, receives the key emitted by the map function and a list of values aggregated by the shuffle phase. A sample percentage (ρ) is obtained from either the context or configuration file. The KSample is run using ρ and *values* as input, the KSample's reservoir resultant is returned as output of the reduce function.

Algorithm 5: Map function

Input : $\Delta = \{\delta_0, \delta_1, \delta_2 \dots \delta_n\}$ set of files.
Output: *map* $\langle key, value \rangle$ resultant list of key-value.
map $\leftarrow \{\}$
for each $\delta \in \Delta$ **do**
 for each $\ell \in \delta$ **do**
 $\Gamma \leftarrow Random(0, 1)$
 map.put(Γ, ℓ)
return *map*

After the map phase, Hadoop sorts the intermediate keys and merges the values which

share the same key. As map keys are randomly chosen, the lines of files are automatically dispersed in the Hadoop flow, that fact avoids the neighboring lines compete by the same slot when reducers run the KSample. The MR KSample can be classified as a Stratified Random Sampling, because each reservoir belonging to a reduce instance can be seen as a random stratum. After the reduce phase, Hadoop gathers all reducers' reservoirs and creates a global reservoir that contains at least $\rho\%$ of the input data. That property is guaranteed by the KSample's proof, which ensures that each reducer reservoir contains at least $\rho\%$ of the stream.

Algorithm 6: KSample's reduce function

Input : *key* key emitted by the map function.

Input : *values* list of values aggregated by shuffle phase.

Output: \mathcal{R} list of resultant values.

$\rho \leftarrow$ get sample percentage from context or config file.

$\mathcal{R} \leftarrow KSample(\rho, values)$

return \mathcal{R}

CHAPTER 5

FRAMEWORK FOR TESTING

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

5.1 Framework Overview

In figure 5.1 we show all components together. First of all, the user creates a file containing the properties, arguments and the initial population of set of knobs. The file is submitted to the front-end which performs lexical, syntatic and semantic analysis to validate it. After that, the file is parsed and the information sent to the engine component. The engine activates the BA to genere and test new job configurations until it reaches the desired criteria. The resultant configuration is saved in a file by the back-end component.

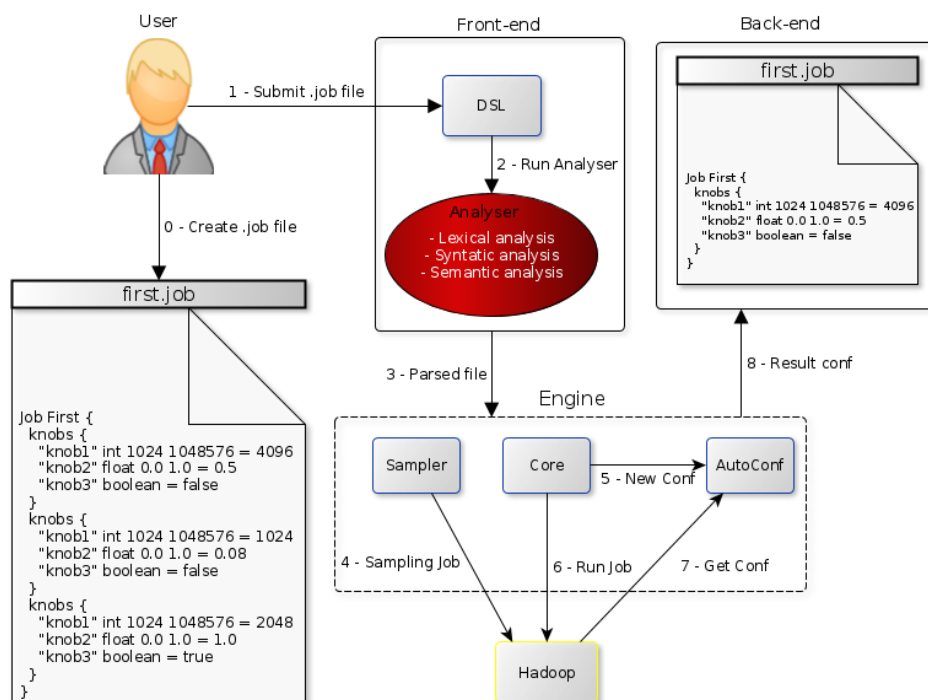


Figure 5.1: Framework's flow.

An interesting feature is that the resultant file can be used as input for the next round of the tuning process, the user could only submit it as input. Therefore, the framework can work as an incremental software to improve its last result.

5.2 Front-end

Our front-end is inspired by a *DSL*, which is a way to elucidate a specific context through appropriate notations and abstractions [15]. A 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 users. There are a lot of examples of DSLs in different domains: LEX, YACC, Make, SQL, HTML, CSS, LATEX, etc. [6].

A DSL normally focuses on a specific domain. But it can also cover different domains or a single, yet broader domain. Such DSL is called general-purpose language (GPL) , because its expressiveness power is not restrict to an exclusive domain. Examples of such GPLs are Cobol and Fortran, which could be viewed as languages focused on the broader domains of business and scientific programming, respectively [15].

5.2.1 DSL Design Methodology

The first step to create a new DSL consists of identifying the problem domain. Depending on the context, it is not trivial to abstract the complete knowledge about the domain, because developers must have a deep prior knowledge on the context, and they have to consider all variables and intrinsic aspects belonging to the domain. Furthermore, sometimes the context can cover more than one domain (for example the GPLs). In other cases, the domain is simple, and performing appropriate notations and abstractions is trivial. In both cases, the foreknowledge of the developers is the factor that influences the quality of the resulting DSL.

After identifying the problem domain developers must abstract all relevant aspects from it. For example *VHDL*, which groups basic logical components like: gates circuits, bus, control signals and logical operators. From these components is possible to create complex logic circuits since a ALU (Arithmetic Logic Unit), register bank until a complex microprocessor.

The next step consists in designing a DSL that expresses applications in the domain. A DSL will have limited concepts which are all focused on the specific domain. To design a DSL, it is necessary to analyse the relationship between it and the existing languages. According to [32], there are some design patterns to develop a DSL based on existing languages. These are represented in Table 5.1.

Pattern	Description
Language exploitation	DSL uses (part of) existing GPL or DSL. Important subpatterns: <ul style="list-style-type: none"> • 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.

Table 5.1: DSL design patterns [32].

In the implementation, a library with the semantic notations is built together with a compiler that performs the lexical, syntactic and semantic analysis, after converting the DSL programs to a sequence of library calls. Generally the library and the compiler are built with support of tools or framework developed for this purpose. *Xtext* [16] and *Groovy* [22, 14] are examples of tools to develop DSLs.

5.2.2 The DSL for tuning MapReduce programs

We designed a DSL from scratch based on the **Xtext** framework [16]. This framework requires the definition of a grammar and rules for the specific domain. As our domain is

the tuning process of Hadoop jobs, we put efforts to describe rules to represent all aspects and components required for the job tuning. Then, our DSL is presented in Listing 3 and has the following components:

- The job with its properties and its arguments to be executed;
- The initial sets of knobs;
- The knob with own type, minimum and maximum thresholds and its initial value.

Then a user can write a `.job` file following our DSL, as can see by the action **0 - Create .job file** in Figure 5.1, for instance the file presented in Listing 4. This file provides all information needed by the job “grep-search” described in Section 2.3.1. The tuning process requires some properties like: the path to the MR job jar file, the job class name, the HDFS directory path to store the data sample, the sample percentage, the algorithm (BA or GA), the number of generations to use as criteria for the algorithm and the population size. The grep-search job requires three arguments: the HDFS input directory path, the HDFS output directory path and the regular expression.

After the arguments there are two sets of knobs composing the initial population for the algorithm. The first one could be the current job configuration, and the second one could be the rules of thumb.

Each set of knobs is composed of knobs type: integer and float, which need of the maximum, minimum and the initial value. For instance, the knob `mapred.reduce.tasks` has the minimum value = 1, the maximum = 10 and the initial value = 5.

Also, users might be interested in testing new configurations that would be easy to accomplish, they just can add a new set of knobs, hence the front-end covers as much use cases as the user needs.

```

DomainModel:
  job=Job;

Job:
  'Job' name=STRING '{'
    properties+=Properties
    arguments+=Arguments
    setKnobs+=Knobs*
  '}'
;

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

Property:
  name=ID Value
;

Value: STRING;

Arguments:
  'Arguments' '{'
    args+=Argument*
  '}'
;

Argument:
  Value
;

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

Knob:
  Type
;

Type:
  IntType | FloatType | BoolType
;

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

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

Float:
  INT* '.' INT*
;

BoolType:
  'boolean' name=STRING '=' Boolean
;
Boolean:
  'true' | 'false'
;

```

Listing 3: Our DSL.

```

Job "grep-search" {
  Properties {
    jarPath "/home/tkepe/bin/hadoop-1.2.0/hadoop-examples-1.2.0-SNAPSHOT.jar"
    jobClassName "grep"
    pathSampleDirHDFS "/kepe/grep/sample"
    samplePercent "0.0"
    algorithm "BA"
    numGenerations "10"
    populationSize "4"
  }
  Arguments {
    "/kepe/grep/in"
    "/kepe/grep/out"
    "[a-z]*o[a-z]*"
  }

  Knobs {
    int "mapred.reduce.tasks" 1 10 = 5
    int "io.sort.mb" 1 100 = 90
    int "io.sort.factor" 1 100 = 90
    int "mapred.inmem.merge.threshold" 0 1000 = 90
    float "io.sort.record.percent" 0.0 1.0 = 0.9
    float "io.sort.spill.percent" 0.0 1.0 = 0.9
    float "mapred.job.reduce.input.buffer.percent" 0.0 1.0 = 0.9
    float "mapred.job.shuffle.input.buffer.percent" 0.0 1.0 = 0.9
    float "mapred.job.shuffle.merge.percent" 0.0 1.0 = 0.9
  }
  Knobs {
    int "mapred.reduce.tasks" 1 10 = 5
    int "io.sort.mb" 1 100 = 50
    int "io.sort.factor" 1 100 = 50
    int "mapred.inmem.merge.threshold" 0 1000 = 50
    float "io.sort.spill.percent" 0.0 1.0 = 0.5
    float "io.sort.record.percent" 0.0 1.0 = 0.5
    float "mapred.job.shuffle.merge.percent" 0.0 1.0 = 0.5
    float "mapred.job.shuffle.input.buffer.percent" 0.0 1.0 = 0.5
    float "mapred.job.reduce.input.buffer.percent" 0.0 1.0 = 0.5
  }
}

```

Listing 4: .job file example.

After, users can submit the .job file to our framework, represented by the action **1 - Submit .job file**. For this, they can just run a jar file and passing as parameter the .job file. Then the framework will invoke the Analyser, action **2 - Run Analyser**, who will perform lexical, syntatic and semantic analysis to validate the file, and the Analyser also will parse it, action **3 - Parsed file**, to send to the Engine.

5.3 Engine

The engine is divided in three components: the component to generate data samples, the component to auto configure Hadoop, and the core component which generates job configurations based on the BA.

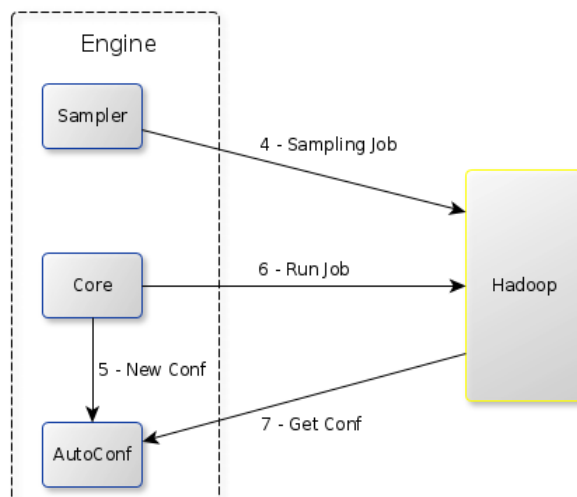


Figure 5.2: Engine processing.

5.3.1 Sampler component

The sampler is responsible for generating data samples. It sends a command to Hadoop in order to run the sampling job, with its output being stored in the *pathSampleDirHDFS* defined by the user in the .job file. This step is represented by the action **4 - Sampling Job**.

5.3.2 AutoConf component

The AutoConf is responsible for saving new configurations generated by the Core (**5 - New Conf**). It receives requests from Hadoop to provide the current configuration that the job will use, as seen in the action **7 - Get Conf**. Although users can assign configurations directly in Hadoop config files (core-site.xml, mapred-site.xml, hdfs-site.xml, etc), running our framework Hadoop only will use the job configuration provide by the AutoConf.

5.3.3 Core component

The core component generates job configurations using the BA or GA. Each new job configuration (**5 - New Conf**) is saved in the AutoConf component. After that, the job is submitted to the Hadoop through the action **6 - Run Job**. Hadoop requests to the

AutoConf the current job configuration and runs the job. In the end the job execution time is assigned to the fitness value, and the new configuration may be added or not to the list of best configurations reached. The actions *5, 6 and 7* in Figure 5.2 occur until the algorithm finishes, i.e., until a criteria is reached.

5.4 Back-end

The back-end, at the moment, is just saving the best configuration reached in a file with the same format of the input file.

CHAPTER 6

EXPERIMENTS

In this chapter we present the experimental validation of our work. The Section 6.1 presents a comparison between the GA and BA. Section 6.2 exposes the quality of the resultant configurations reached by our solution. Section 6.3 reveals the benefit of using data sampling which is corroborated by the cost to obtain a new configuration.

All experiments were done in a cluster composed of three machines, where one machine is the master and the other two the slaves. The machines' settings are: Linux Mint 13 Maya SO, 3GB of ram memory, Sata Disk with at least 250GB.

The cluster was populated using the job *randomtextwriter* embedded in the Hadoop's example jar file. It generates 10GB of random text into each slave machine, totalizing 20GB of data stored on the cluster. All experiments were executed using 10 generations as criteria to the algorithm. For the tests based on data sampling (**BA + Sampling and GA + Sampling**), we first ran our approach on 20% of the whole data. Second, the purpose has been to use the best configuration reached in each generation of the algorithm, and executed the job configured with them on all data. Other tests (**BA + All Data and GA + All data**) were performed directly on the entire data.

6.1 Bacteriological Algorithm against Genetic Algorithm

We executed the BA and GA in order to confront the quality of the configurations reached by both. We used the WordCount job to perform this comparison. Figure 6.1 presents results of six experiments:

1. **BA + All data**: experiment using our approach BA without data sampling, i.e. it was applied on all data.
2. **BA + Sampling**: experiment using our approach BA and data sampling.

3. **GA + All data**: experiment using GA without data sampling, i.e. it was applied on all data.
4. **GA + Sampling**: experiment using GA and data sampling.
5. **Default Config**: Hadoop's default configuration.
6. **Rules of thumb**: the rules of thumb [10, 8, 27, 44].

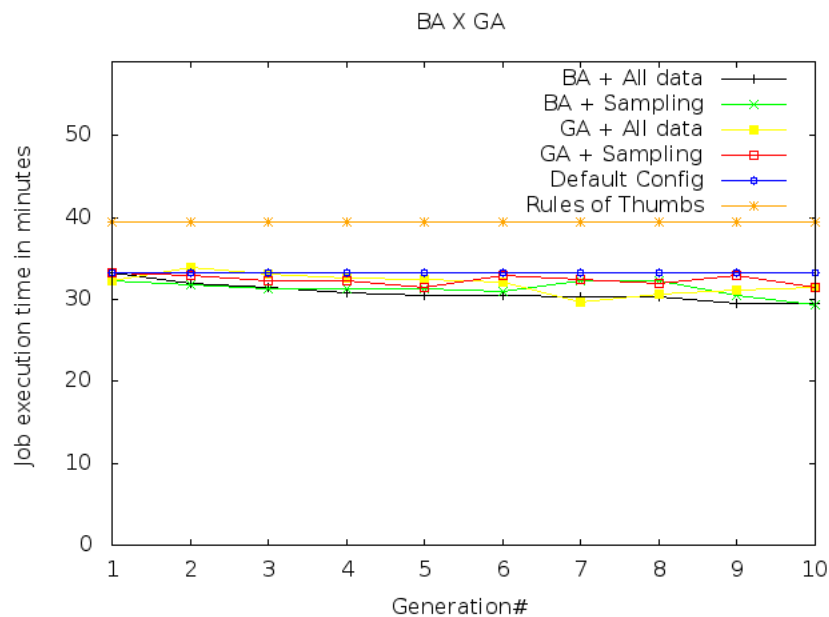


Figure 6.1: BA against GA.

The X axis contains the number of generation and the Y axis the job execution time. The points in the chart express the quality of the configuration in each generation, the quality is measured by the job execution time. We can see that the last configurations reached by both experiments using BA were better than the configurations reached by the experiments using GA. The GA contains regressions in the quality of the configurations reached. For instance the GA without sampling (GA + All data) reached its best configuration in the generation 7, but the generations 8, 9 and 10 worsened that configuration. On the other hand, the BA based on all data (BA + All data) didn't regress. The best configuration was reached in the generation 9, and in the generation 10 a worse configuration was generated, but the BA memorized the configuration of the 9th generation and avoided the regression in the process.

In the next section we bring up a discussion about the configuration quality against the Hadoop's default configuration.

6.2 Configuration quality

In this section we present the quality of the configuration reached for the jobs: WordCount and Grep compared against their default configuration and the rules of thumb presented in [10, 8, 27, 44]. For the WordCount, see Figure 6.2. The configurations reached in every generation were better than Hadoop's default configuration and the rules of thumb. In the last generation the job execution time using the configuration reached is 3 minutes less than the default configuration, which represents a gain of almost 10%. We can observe that the rules of thumb reached the worst performance. It is worth noting that they were worse than the default configuration, reflecting that the rules of thumb are generic and they may not be accurate for all jobs.

As can see in the Figure 6.3, the configurations reached by our approach using data sampling improved the grep job execution time in almost 1 minute.

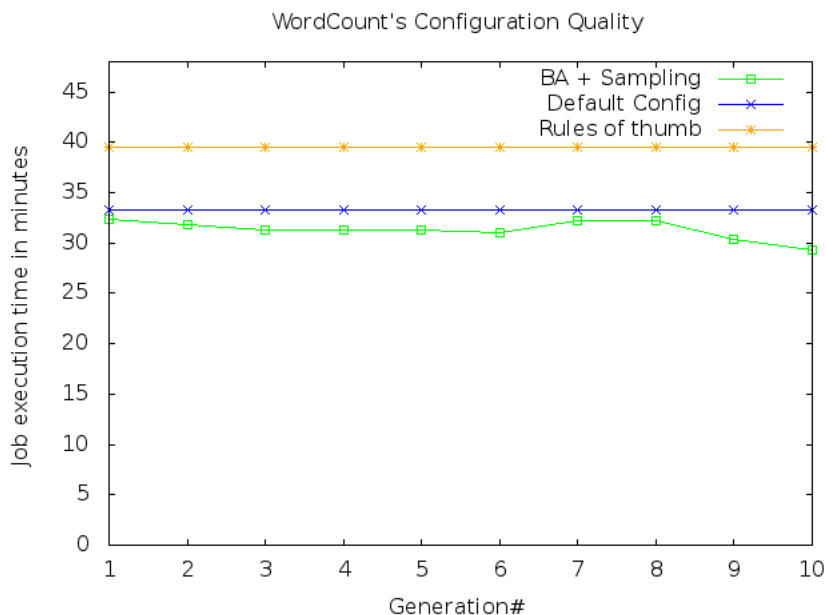


Figure 6.2: WordCount's configuration quality.

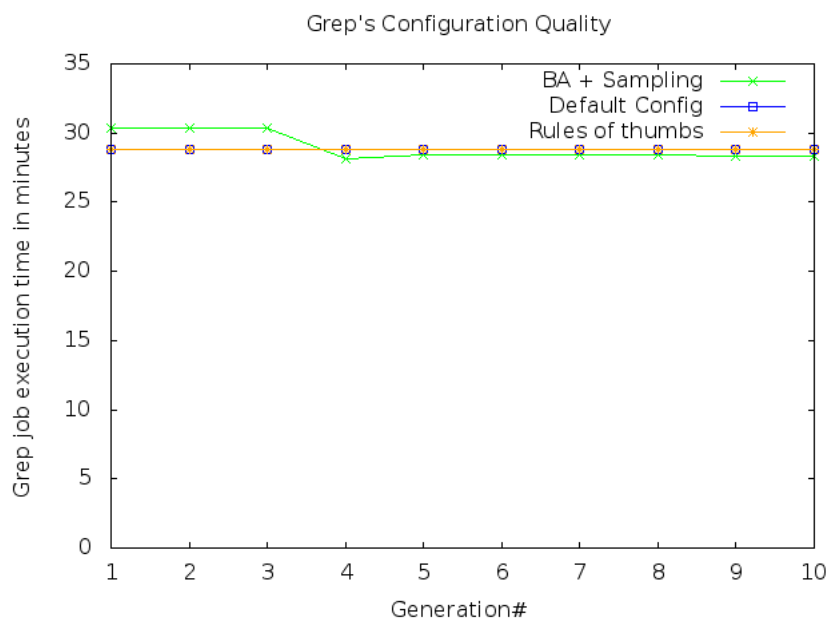


Figure 6.3: Grep's configuration quality.

6.3 Cost of the tuning process

We measured the cost to run the tuning process using the WordCount job. Figure 6.4 shows the cost per generation using our approach on all data and on the samples. Our approach applied on data samples in every generation was at least 1 hour less costly than applied on all data.

The aggregate cost in Figure 6.5 reveals a foremost result to justify the usage of data sampling. With data sampling our approach was almost 20 hours faster than the tuning process on all data. Also, as we can see on the graphics in the Section 6.2 the quality of the WordCount and Grep job configuration reached by our approach using BA and data sampling was better than the approach applied on all data.

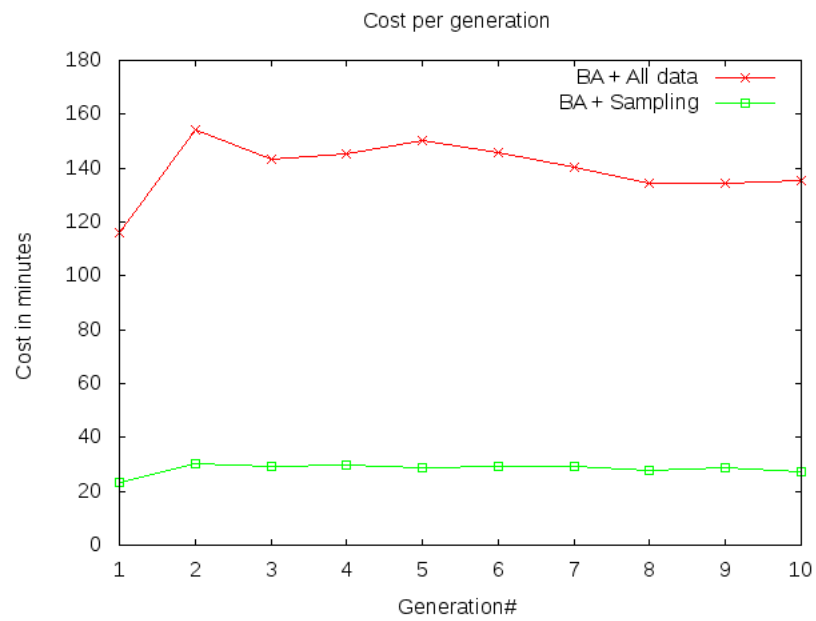


Figure 6.4: Cost of each generation.

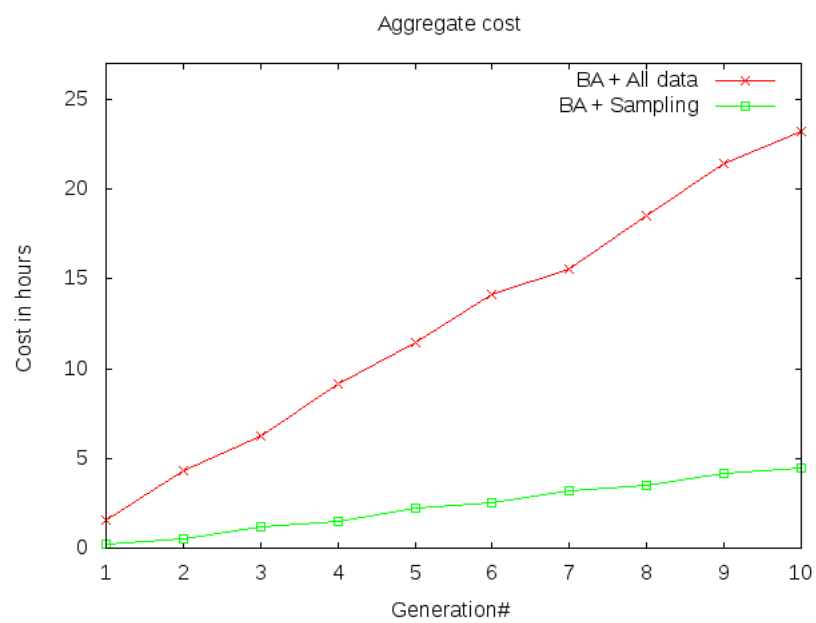


Figure 6.5: Agregate cost per generation.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Hadoop is a popular data processing solution for Big Data purposes. In the last years it has been adopted for many companies to handle massive data sets in order to infer information from their databases. The open source community, research scientific institutes, and several enterprises are spending efforts to enhance Hadoop, due to the inherent complexity present in Big Data solutions, i.e. large machine clusters, massive amount of data that grow up in volume, variety and velocity, several processing jobs etc. The Hadoop's performance is quite sensitive to changes in the environment aggravated for the large number of configuration parameters, greater than 250. Our solution aims to tuning Hadoop jobs, considering the fact that data engineers and Hadoop users have wasted too much time configuring jobs instead of focusing on data analysis.

With our solution Hadoop users will not worry about tuning every specific parameter for each MR job. Our evolutionary algorithm can generate new tuning configurations and test them by running the jobs on data samples. Data sampling is a proven technique to reduce cost when querying data. Our KSample algorithm is a robust data sampling algorithm which doesn't depend of complex knowledge about the data set, queries history and even its structure. It allows that users just provide an intuitive parameter of input (a percentual number of the population) without worrying about complex parameters, such as: resultant sample size, population size, number of samples etc. Using our framework users can just create a .job file following our DSL, as described in the Section 5.2.2, that our framework undertakes in choosing a better configuration for the job according to the current state of the MR environment.

Experimental investigation showed, as can be seen in Chapter 6, the accuracy of our solution, i.e. the relation between the quality of the configuration reached and the cost to obtain it, which reveals a gain in jobs performance with a relatively low cost.

During the experiments we realized that some configurations reached from data sampling didn't obtain a expected performance when applied on the whole data. For instance the WordCount, as shown the Figure 7.1, reveals two discrepant points meaning that the generation 7 and 8 reached good configurations when applied on the samples, but they don't reflect the expected performance when applied on all data. We have to investigate why these configurations had a different behavior when applied on all data.

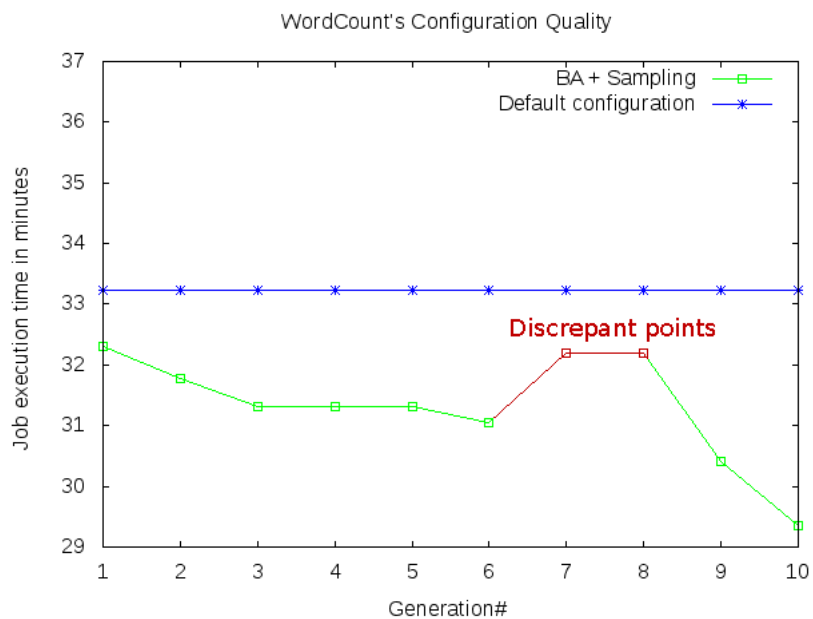


Figure 7.1: Case to study.

In this work we considered just the job execution time as fitness value. In the future we should be considering other metrics, like: IO operation, CPU, Memory and Network usage. Due users might be more concerned in resource usages than in response time, as cloud service providers offer plans based on “pay as you go”, such as Amazon EC2 [18].

In addition, we can implement and test a new algorithm composed of a mix of GA and BA, i.e. implementing the GA using the BA's memorization operator, thus we can use the features of both algorithms.

BIBLIOGRAPHY

- [1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, New York, NY, USA, 2013. ACM.
- [2] apache.org. Language manual sampling. <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Sampling>, 2013. Accessed on 11th July 2013.
- [3] apache.org. Welcome to hive! <http://hive.apache.org/>, 2013. Accessed on 23th September 2013.
- [4] Inc. Aster Data Systems. In-database mapreduce for rich analytics. <http://www.asterdata.com/resources/mapreduce.php>, 2013. Accessed on 3rd October 2013.
- [5] 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.
- [6] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(1):711–721, August 1986.
- [7] 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.
- [8] Cloudera. 7 tips for improving mapreduce performance. <http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance>, 2014. Accessed at 17th August 2014.
- [9] Cloudera, Inc. Algorithms every data scientist should know: Reservoir sampling. <http://blog.cloudera.com/blog/2013/04/>

- hadoop-stratified-randosampling-algorithm, 2013. Accessed on 14th July 2013.
- [10] cloudera.org. Configuration parameters: What can you just ignore? <http://blog.cloudera.com/blog/2009/03/configuration-parameters-what-can-you-just-ignore>, 2014. Accessed on 17th August 2014.
- [11] Edgar Frank Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [12] P.G. de Vries. *Sampling theory for forest inventory: a teach-yourself course*. Springer-Verlag, 1986.
- [13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*. USENIX Association, 2004.
- [14] Fergal Dearle. *Groovy for Domain-Specific Languages*. june 2010.
- [15] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN NOTICES*, 35:26–36, 2000.
- [16] eclipse.org. Xtext. <http://www.eclipse.org/Xtext>, 2013. Accessed on 1st April 2013.
- [17] explorable.com. Systematic sampling. <https://explorable.com/systematic-sampling>, 2014. Accessed on 5th September 2014.
- [18] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *SIGMOD Rec.*, 38(1):43–48, junho 2009.
- [19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, outubro 2003.

- [20] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
- [21] Greenplum. A unified engine for rdbms and mapreduce. <http://docs.huihoo.com/greenplum/Greenplum-MapReduce-Whitepaper.pdf>, 2013. Accessed on 3rd October 2013.
- [22] groovy.codehaus.org. Domain-specific languages with groovy. <http://groovy.codehaus.org/Writing+Domain-Specific+Languages>, 2013. Accessed on 1st April 2013.
- [23] Greg Grothaus. Reservoir sampling - sampling from a stream of elements. <http://gregable.com/2007/10/reservoir-sampling.html>, 2013. Accessed on 6th August 2013.
- [24] 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.
- [25] hadoop.apache.org. Apache hadoop. <http://hadoop.apache.org/>, 2013. Accessed on 24th March 2013.
- [26] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.
- [27] Intel. Optimizing hadoop* deployments. Technical report, 2010.
- [28] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, 2007.
- [29] Jennifer Widom Jeffrey D. Ullman, Hector Garcia-Molina. *Database systems - the complete book*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2009.
- [30] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proc. VLDB Endow.*, 5(10):1028–1039, junho 2012.

- [31] Roy Levin and Yaron Kanza. Stratified-sampling over social networks using mapreduce. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, 2014. ACM.
- [32] 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.
- [33] monetdb.org. Database sampling. <http://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/Sampling>, 2013. Accessed on 6th August 2013.
- [34] monetdb.org. Monetdb. <http://www.monetdb.org>, 2013. Accessed on 2nd July 2013.
- [35] 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.
- [36] Pivotal. Pivotal greenplum database. <http://gopivotal.com/pivotal-products/data/pivotal-greenplum-database>, 2013. Accessed on 3rd October 2013.
- [37] Adrian Daniel Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. Same queries, different data: Can we predict runtime performance? Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *ICDE Workshops*. IEEE Computer Society, 2012.
- [38] Randomsampling.org. Random sampling. <http://www.randomsampling.org>, 2013. Accessed on 2nd July 2013.
- [39] Zujie Ren, Zhijun Liu, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. Wax-elephant: A realistic hadoop simulator for parameters tuning and scalability analysis. *Proceedings of the 2012 Seventh ChinaGrid Annual Conference*, CHINAGRID '12, Washington, DC, USA, 2012. IEEE Computer Society.
- [40] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

- [41] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
- [42] Jeffrey Scott Vitter. Faster methods for random sampling. *Commun. ACM*, 27(7):703–718, 1984.
- [43] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. *17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2009, September 21-23, 2009, South Kensington Campus, Imperial College London*. IEEE, 2009.
- [44] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

TIAGO RODRIGO KEPE

**A TUNING APPROACH BASED ON EVOLUTIONARY
ALGORITHM AND DATA SAMPLING FOR BOOSTING
PERFORMANCE OF MAPREDUCE PROGRAMS**

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