EDSON RAMIRO LUCAS FILHO

HIVEQL SELF-TUNING

CURITIBA

2013

EDSON RAMIRO LUCAS FILHO

HIVEQL SELF-TUNING

Dissertation presented as partial requisite to obtain the Master's degree. M.Sc. program in Informatics, Universidade Federal do Paraná.
Orientador: Prof. Dr. Eduardo Cunha de Almeida
Co-Orientador: Prof. Dr. Luis Eduardo S. Oliveira

CURITIBA

2013

L933h

Lucas Filho, Edson Ramiro
HiveQL self-tuning / Edson Ramiro Lucas Filho. – Curitiba, 2013.
44f. : il. color. ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Tecnologia, Programa de Pós-graduação em Informática, 2013.
Orientador: Eduardo Cunha de Almeida -- Co-orientador: Luis Eduardo S. Oliveira.
Bibliografia: p. 41-44.

1.Controle automático. 2. Sistema de controle ajustável. 3. Bancos de dados. I. Universidade Federal do Paraná. II. Almeida, Eduardo Cunha. III. Oliveira, Luis Eduardo S. IV. Título.

CDD: 005.745028564

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 Edson Ramiro Lucas Filho, avaliamos o trabalho intitulado, *"HiveQL Self-Tuning"*, cuja defesa foi realizada no dia 27 de agosto de 2013, às 14:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela **aprovação** do candidato.

Curitiba, 27 de agosto de 2013.

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

Profa. Dra. Nádia Puchalski Kozievitch
UTFPR – Membro Externo

Prof. Dr. Luiz Eduardo Soares de Oliveira
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

DEDICATION

Não cesses de falar deste Livro da Lei; antes, medita nele dia e noite, para que tenhas cuidado de fazer segundo tudo quanto nele está escrito; então, farás prosperar o teu caminho e serás bem-sucedido. Não to mandei eu? Sê forte e corajoso; não temas, nem te espantes, porque o SENHOR, teu Deus, é contigo por onde quer que andares. Josué 1.8-9.

dedico esta dissertação ao meu Deus,
O DEUS Criador, Autor de toda a vida.

ACKNOWLEDGMENTS

A Deus, Autor da vida, sem o qual nem respirar posso.

Ao meu pai, Edson Ramiro Lucas, por incentivar-me sempre a estudar.

Ao meu avô, Alcino Modanese, por me dar sempre o conforto de um lar.

Ao Pastor Paulino Cordeiro e Pastora Roseli Cordeiro, por ensinar-me o caminho de Cristo Jesus.

À professora Leticia Mara Peres pelas ideias que em muito ajudaram na condução das pesquisas relizadas.

Ao professor Marcos Didonet del Fabro pelas indicações de tecnologias e soluções para compor este trabalho.

Ao professor Eduardo Cunha de Almeida e ao professor Luiz Eduardo S. Oliveira pela orientação.

Agradeço também a todos, os colegas do café e do laboratório, que, direta ou indiretamente, contribuíram para a ralização deste trabalho.

Agradeço ao CNPq por uma bolsa de pesquisa - 481715/2011-8, Fundação Araucária - 20874 e SERPRO.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

LIST OF ACRONYMS

AMD Advanced Micro Devices

ATA AT Attachment Interface

CLI Command Line Interface

CPU Central Processing Unit

DAG Direct Acyclic Graph

DBGen Database Generator

GFS Google File System

GHz Gigahertz

HDFS Hadoop Distributed File System

HiveQL Hive Query Language

HQL HiveQL Query

HTTP Hypertext Transfer Protocol

I/O Input/Output

JDBC Java Database Connectivity

JSON JavaScript Object Notation

LCSS Longest Common Subsequence

ODBC Open Database Connectivity

PXQL PerfXPlain Query Language

RPM Rotations per Minute

RSS Random Recursive Search

SGE Sun Grid Engine

SQL Structured Query Language

TPC-H Transaction Processing Performance Council - Benchmark H

RESUMO

Bancos de dados construídos sobre MapReduce, tais como o Hive e Pig, traduzem suas consultas para um ou mais programas MapReduce. Tais programas são organizados em um Grafo Acíclico Dirigido (GAD) e são executados seguindo sua ordem de dependência no GAD. O desempenho dos programas MapReduce depende diretamente da otimização (i.e., sintonia) dos parâmetros de configuração definidos no código-fonte. Sistemas como Hive e Pig traduzem consultas para programas sem otimizar estes parâmetros. Existem soluções que buscam a melhor configuração para programas MapReduce, entretanto, tais soluções precisam coletar informação de suporte durante a execução ou simulação das consultas para realizar a predição de melhor configuração. Coletar informação de suporte pode adicionar uma sobrecarga no processo de otimização do programa, mesmo quando o tamanho do dado de entrada é muito grande, ou quando usando apenas uma fração. Nossa hipótese é que pode-se evitar a coleta de informação de suporte por agrupar consultas que tenham a mesma assinatura de código para, então, otimizar seus parâmetros com uma mesma configuração. Nesta dissertação nós apresentamos uma abordagem de auto-sintonia para sistemas de *data warehouse* construídos sobre MapReduce. Nossa abordagem analisa em tempo de execução as consultas, extraindo as assinaturas de código (i.e., operadores de consulta como GroupBy e Select) e agrupando as consultas que exibem as mesmas assinaturas de código. Ao agrupar os programas MapReduce, nossa solução aplica uma configuração única para cada assinatura de código, baseando-se nas regras-de-ouro. Durante os experimentos nós observamos a existência de um limite no qual a otimização realizada com as regras-de-ouro, ou mesmo com a nossa abordagem, não é eficaz para consultas abaixo deste certo limite. Nós validamos a nossa abordagem por meio de experimentação executando o TPC-H Benchmark.

Palavras chave: Hadoop; MapReduce; Auto-Sintonia.

ABSTRACT

In MapReduce, performance of the programs directly depends on tuning parameters manually set within their source-code by programmers. In the database context, MapReduce query front-ends, including Hive and Pig, automatically translate MapReduce programs from SQL-like queries written in HiveQL. However, these front-ends only care about translating queries and do not care about including tuning parameters. Different solutions seek for the appropriated setup for MapReduce queries, but they need to collect support information after execution or simulation. In the one hand, if there is no tuning of MapReduce queries, their response time increase due to waste of computer resources. In the other hand, collecting support information may add a costly overhead whether the size of the input data grows large, or even when using a fraction of the input data. Our hypothesis is that we can avoid collecting support information by finding queries with the same code signature and tuning them with similar configuration setup. In this dissertation, we present a HiveQL self-tuning approach for MapReduce data warehouse systems based on clustering queries that exhibit the same characteristics in terms of query operators. Our approach uses dynamic analysis to extract characteristics from running queries to build similarity clusters. By clustering the queries, our mechanism leverages tuning information gathered in advance, such as the rules-of-thumb, to allow on-the-fly adaptation of queries setup. During our experimentation we observed the existence of a threshold at which tuning with the rules-of-thumb is not effective. We validated our approach through experimentation running the TPC-H benchmark.

Key-words: Hadoop; MapReduce; Self-Tuning.

CHAPTER 1

INTRODUCTION

The MapReduce programming model [6] presents an alternative to the parallel database systems to building programs that process large amounts of data across large clusters. The Apache Hadoop framework [22] is a popular open-source implementation of MapReduce that serves as the foundation for an ecosystem of data intensive systems, including Hive, Pig, Mahout, Nutch, HBase. The Apache Hive [4] data warehouse system built on top of Hadoop comes along with a SQL-like language called HiveQL. To execute a query into Hadoop, Hive translates a HiveQL query into a Directed Acyclic Graph (DAG) of stages, where each stage is a complete Hadoop program and comprises of a set of references to input data and a collection of operators (e.g., TableScan, Join, MapJoin, Select) that we consider as the code signature. In Hive, the stages of a same query share the same configuration setup, although they have different signatures that may lead to a different behavior, such as disk access or network usage.

1.1 Background

Computing resources of MapReduce machine clusters can exhibit heterogeneous characteristics and fluctuating loads. Query front-ends such as Hive [4] and Pig [9] do not care about tuning setups to squeeze performance from these machines and MapReduce backends do not have self-tuning facilities like the ones from relational database systems for automatic tuning. Generally, tuning is made by system administrators or developers who may not grasp those load fluctuations or are novice to the MapReduce data processing model. While, there are tuning systems that help Hadoop administrators and developers in searching for the best setup values, setup tuning is yet done manually within the source-code of the programs. Once the task of applying the best setup values is delegated to the programmer, even with help of tuning-systems, misconfiguration may happen and

lead to poor performance. In addition, it is impossible to set a multi-configured query manually considering the high number of variables, such as the variation on the size of the intermediate tables generated by Hive during query execution.

1.2 Objective

Both Hadoop and Hive provide together more than four hundred configuration knobs that can be tuned to boost performance. Tuning these knobs is a hard task due to the number of variables involved that may vary from cluster workload and input data size to algorithms for compression and degree of parallelism. We identified two main problems concerning Hadoop/Hive tuning: (1) choosing the best setup to boost performance and (2) finding similar stages in order to apply acquainted setup. Different solutions [26, 18, 19, 14], including CPU workload pattern matching, and workload profiling may be used to addresses both problems. However, they need to collect support information from execution or simulation of the programs in order to seek for the appropriated setup. In the one hand, if there is no tuning of HiveQL queries (and stages), their response time increase due to waste of computer resources. In the other hand, collecting support information may add a costly overhead whether the size of the input data grows large, or even when using a fraction of the input data. Our objective is to provide a self-tuning system for HiveQL queries that avoids the collection of support information and tune the queries transparently to the Hadoop administrator or developer.

1.3 Challenges

The support information collected from the query execution gives an insight about its resource consumption pattern. The tuning systems that use heuristics, knowledge base or cost-based approaches along with the support information to seek for the best tuning need to execute or simulate the queries in order collect such information and generate the appropriated tuning. The support information remains as the guidance to search for the appropriated tuning. The challenge of tuning queries without collecting such information

remains on the lack of the guidance provided by the support information. One problem of tuning systems based on support information is that they may add a costly overhead for tuning *Ad-Hoc* queries, which are processed only once. Tuning *Ad-hoc* queries based on support information implies that these queries must be executed twice just for tuning.

1.4 Contribution

In this dissertation, we present a HiveQL Self-Tuning system called AutoConf, which address the second tuning problem. Our hypothesis is that we can avoid collecting support information by clustering stages with the same code signature and tuning them with the same configuration setup. Our approach uses dynamic analysis to extract characteristics from running stages and build similarity clusters. By clustering the stages, our system leverages tuning information gathered in advance, such as the rules-of-thumb, to allow on-the-fly adaptation of the stages. We validated our approach through experimentation running the TPC-H benchmark. During our experimentation we observed the existence of a threshold at which tuning with the rules-of-thumb is not effective, even using our system. The remainder of this dissertation is organized as follows. We introduce the MapReduce programming model and an overview on Hive query execution flow in Section 2. We present the related work in Section 3. We describe our solution in Section 4. The analysis and corresponding results are presented in Section 5. Finally, we conclude the dissertation and present future work in Section 6.

CHAPTER 2

MAPREDUCE

In this dissertation we focus on the Hadoop implementation once the various implementations differs in architectural details. We briefly describe the MapReduce programming model in Section 2.1. The components of the Hadoop framework, are presented in Section 2.2. Finally, we give an overview of Hive query execution flow in Section 2.3.

2.1 Programming Model

The MapReduce programming model is based on the Functional programming paradigm, which decomposes the solution in a set of functions. The MapReduce provides predefined functions such as *Map*, *Partition*, *Comparison*, *Reduce* and *Combiner* to perform computation over a given data bulk. Most of the MapReduce computation is based on *Map* and *Reduce* functions, which are two main high-order functions that perform computation based on *key* and *value* pairs.

The *Map* and *Reduce* functions are divided into several subphases. The *Map* function is divided into *Reading*, *Map Processing*, *Spilling*, and *Merging* subphases, and the *Reduce* function is divided into *Shuffling*, *Sorting*, *Reduce Processing*, and *Writing* subphases. The *Map* function maps the input data into a list of *key* and *value* pairs. These pairs are grouped and processed by the *Reduce* functions, where a *Reduce* function may process a unique *key* or a list of *keys*. The result is a list grouped by *keys* and their corresponding *values* (see Equations 2.1, 2.2).

$$Map(key1, value1) \rightarrow list(key2, value2) \tag{2.1}$$

$$Reduce(key2, list(value2)) \rightarrow (key2, list(value3)) \tag{2.2}$$

To illustrate the MapReduce model, we use the WordCount example bundled with the official distribution of Hadoop. The WordCount calculates the occurrence of each word in a given text, similar to the *wc* command of Unix systems. Suppose we have one terabyte of pure text as *input data* and we want to count the occurrences of each word in the given text. Thus, the first step is to load the *input data* into Hadoop.

The master node coordinates the WordCount execution, and as it receives the *input data*, chop the *input data* into several pieces called *splits*. While the master node generates the *splits*, it keeps sending these *splits* to the slave nodes of the cluster, which are responsible to store the *splits* locally with a default size of 64 megabytes each. Administrators and developers may tune this size to higher values depending on the application and the overall input size. The load process finishes after the slave nodes have received their own *splits*.

In a second step, the *WordCount* is submitted to the master node for execution, which, in turn, reads the *Map* and *Reduce* functions and sends them to the slave nodes. The master node indicates to the slaves that the processing can start. Then, the slave nodes start to perform the *Map* and *Reduce* functions over their *splits*. This process takes the computation to the data, instead of take the data to the computation.

The Code 2.1 is the *Map* function from the WordCount example bundled along with the official Hadoop distribution. The *key* argument is the document name (i.e., the input data shared across the slave nodes) and the *value* is the content of each *split*. In order to count the occurrences of each word in the given *input data*, the *Map* function emits a pair with the word w and the value 1 for each word in the text *value*. Each slave node execute its own *Map* function. Each *Map* instance being executed produces a list of pairs called *intermediate-pairs* (e.g., $\{\{cat, 1\}, \{rat, 1\}, \{the, 1\}\}$).

```
1 map(String key, String value)
2     // key: Document name
3     // value: Document contents
4     for each word w in value:
5         EmitIntermediate(w, '1');
```

Code 2.1: Map function excerpt from the WordCount [6].

The *Reduce* presented in Code 2.2 is the *Reduce* function from the same WordCount example. After the processing of the *Map* function by the slave nodes, the *Reduce* function starts to process the *intermediate-pairs*. But, before the *Reduce* starts, the *intermediate-pairs* with the same *key* are grouped by a function called *Partitioner*, which is responsible to determine which *Reducer* instance will process a determined *key*. The *Partitioner* function is executed in the *Shuffle* phase. Usually the programmers do not modify the default *Partitioner* function.

```
1 reduce(String key, Iterator values)
2     // key: a word
3     // values: a list of counts
4     for each v in values:
5         results += ParseInt(v);
6     Emit(AsString(result));
```

Code 2.2: Reduce function excerpt from the WordCount [6].

Each *Reduce* fetches the assigned *intermediate-pairs* from the *Partitioner* via Hyper Text Markup Language (HTTP) into memory. The *Reduce* periodically merges these *intermediate-pairs* to disk. In the case of compression for the *intermediate-pairs* is turned on, each set of *intermediate-pairs* that cames from the *Partitioner* is decompressed into memory.

Each variable *value* in the *intermediate-pairs* has the value 1. The *intermediate-pairs* with the same *key* are processed by the same *Reduce* instance, which receives the *intermediate-pairs* and sums up the *values*. The Figure 2.1 shows the data flow during the WordCount processing.

Finally, the result of each *Reduce* is a pair with the *key* and the sum of the value *value* ($\{key, sum(values)\}$). The *sum(values)* corresponds to the occurrences of each word in the given text. The output pairs from all the reduce instances (i.e., or a list of pairs from one reduce instance in case there are more than one key in the same *Reduce*) are merged to compound the final result. Many other kinds of computation such as graph and machine learning algorithms are handled by the MapReduce model. The prerequisite to use MapReduce is to rewrite the algorithms to use the model of key and value pairs imposed by the Map and Reduce functions.

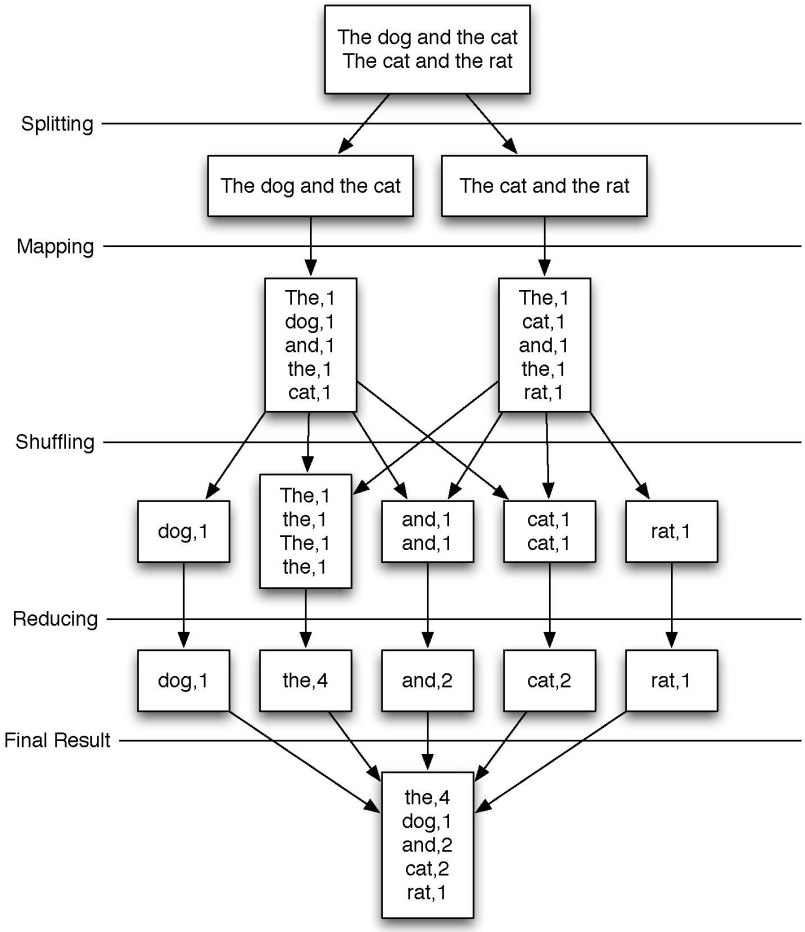

Figure 2.1: The WordCount Data Flow Example.

2.2 Architecture

The Hadoop framework is an open-source implementation of the MapReduce programming model, designed to process large amounts of data over clusters of commodity machines. It is also designed to scale up from single servers to thousands of machines, where each machine offers local storage and computation.

When administrators or developers use Hadoop to develop distributed software they neither care about deployment across the cluster nor treat the common problems related to distributed applications such as: synchronization, reconciliation, concurrence, fault tolerance and scalability. Instead of caring about these common problems, the administrator or developer configure how the framework must act with the Hadoop program, e.g., the number of replicas in HDFS, the number of map and reduce instances per slave node, buffer sizes and scheduling algorithms. When these configurations are not set by the administrator or developer, the Hadoop framework assigns default values.

For tuning purposes, configuration parameters are called tuning knobs, while their assigned values are called setup values. After the *Map* and *Reduce* functions have been wrote by the developer and the tuning knobs were set up, the Hadoop program is executed by the framework. In Hadoop, a program in execution is called job.

Figure 2.2 illustrate the Hadoop architecture. The Hadoop framework consists of a distributed file system, described in Section 2.2.2, and the processing engine, described in Section 2.2.1.

2.2.1 The Processing Engine

The Hadoop framework accepts simultaneous job submissions, from different users, organizing all jobs into a queue. The *JobTracker* is the coordinator of the Hadoop processing engine and is executed in the master node. The *JobTracker* divide each job into several instances of *Map* and *Reduce* functions, which are called *tasks*.

Each slave node runs the processing engine client called *TaskTracker*. Each *TaskTracker* is configured with a set of *slots* to indicate the number of *tasks* it can accept.

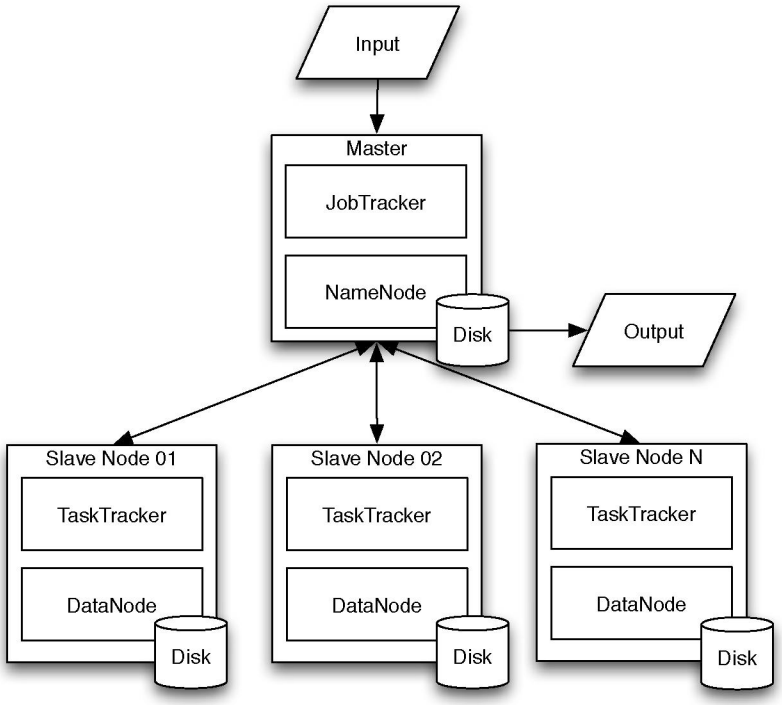

Figure 2.2: The Hadoop architecture.

The *JobTracker* coordinates the execution of the delivered *tasks*, ordering when each *task* must be processed and which is the *TaskTracker* that will process.

Each slave node has its own *splits* saved locally. The *tasks* (i.e., the *Map* and *Reduce* functions) received from the *JobTracker* by the *TaskTracker* consume the local *splits*. As the *tasks* related to the *Map* function finishes, the *JobTracker* orders the *TaskTrackers* to perform the *tasks* related to the *Reduce* function.

2.2.2 The Distributed File System

The Hadoop framework uses a distributed file system to share data among the nodes. The Hadoop Distributed File System (HDFS) is an open-source implementation of the Google File System (GFS) [10], bundled along with the official Hadoop distribution. The HDFS coordinator is called *NameNode*. It keeps the directory tree of all files stored in the file system and tracks the location of each file in the cluster. The *NameNode* is responsible to receive the input data, chop it into several *splits* and store all *splits* into the HDFS.

Each slave node has the HDFS client called *DataNode*, which is responsible for storing

the data into local disks. The *DataNode* instances talks to each other to replicate data. Whenever a *job* have to locate a file, it queries the *NameNode* for the machine address that stores the data. After the *job* has received a machine address it contacts the machine directly to get the data.

2.3 Hive query execution overview

The Apache Hive [4] (or simply Hive) is a data warehouse system built on top of Hadoop, which comes along with a SQL-like language called HiveQL. In the Hive data warehouse system queries are submitted via interfaces such as JDBC (Java Database Connectivity), ODBC (Open Database Connectivity) or Hive CLI (i.e., Hive command line). As we illustrate in Figure 2.3, Hive receives a *query sentence* and send it to the *Compiler*, which is the Hive component responsible to translate the *query sentence* into a *logical query plan*. The *logical query plan* consists a DAG of *stages*, where each *stage* is a complete MapReduce program with a collection of *operators* and a set of input data. The *operators* are minimum processing units inside Hive and implements SQL-like functionalities such as Join, Select and GroupBy. The input data are tables and intermediate tables existing in or generated by Hive during the query process.

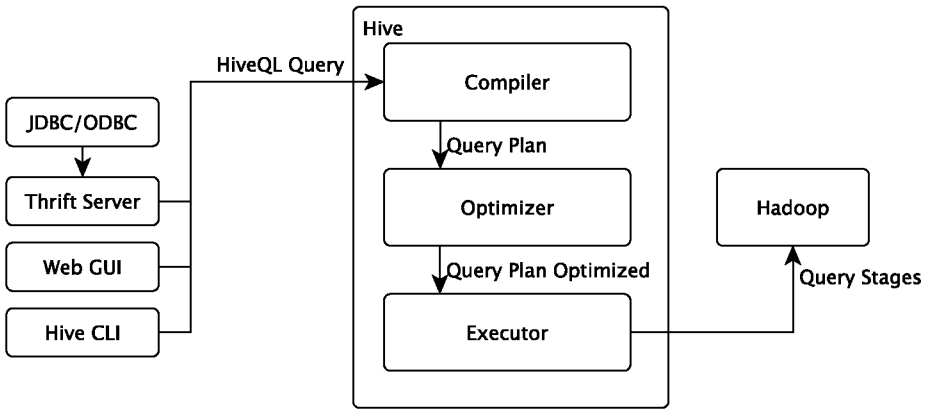

Figure 2.3: HiveQL query execution flow inside Hive.

After the *query sentence* translation, the *Compiler* sends the *logical query plan* to the *Optimizer*, which performs multiple passes rewriting it in order to generate an *optimized query plan*. The *optimized query plan* is sent to the *Executor*, which submits *query stages*

from the *optimized query plan* to the Hadoop framework. The submission of *query stages* follows the topological order from the *optimized query plan*. While processing a query, Hive applies the same configuration for all the *stages* inside the same DAG. However, the *stages* within the same DAG may have a distinct collection of *operators* and a different dataset, which may lead the *stages* to different behaviors. Thus, the overall query tuning depends on a *many-to-one* configuration.

```
1  SELECT P_BRAND, P_TYPE, P_SIZE,
2     COUNT(DISTINCT PS_SUPPKEY) AS SUPPLIER_CNT
3  FROM PARTSUPP, PART
4  WHERE P_PARTKEY = PS_PARTKEY
5  AND P_BRAND <> 'Brand#45'
6  AND P_TYPE NOT LIKE 'MEDIUM POLISHED%%'
7  AND P_SIZE IN (49,14,23,45,19,3,36,9)
8  AND PS_SUPPKEY NOT IN
9  (SELECT S_SUPPKEY FROM SUPPLIER
10 WHERE S_COMMENT
11 LIKE '%%Customer%%Complaints%%')
12 GROUP BY P_BRAND, P_TYPE, P_SIZE
13 ORDER BY SUPPLIER_CNT DESC,
14          P_BRAND, P_TYPE, P_SIZE
```

Code 2.3: TPC-H query 16 [32].

Code 2.3 depicts the standard Query-16 from the TPC-H [32] benchmark that finds out how many suppliers can supply parts with given attributes. Code 2.4 is the equivalent query translated to HiveQL. As Hive does not support[1] IN, EXISTS or subqueries in the WHERE clause, the output of subqueries must be saved into temporary tables, resulting in three HiveQL queries.

Figure 2.4 illustrates the workflow among the three HiveQL queries from Code 2.4, and details the dependence between the *stages*. The first node in Figure 2.4 is the *HiveQL Query-1* that refers to first query from Code 2.4, the *HiveQL Query-2* refers to the second query from Code 2.4 and the *HiveQL Query-3* refers to the third query from Code 2.4. Inside each query node we illustrate the *stages*. In the *HiveQL Query-1* we have the *Stage-1*, which implements the *operators*[2] TableScan, Filter, Select and FileSink. Also

[1]Ticket to implement support for correlated subqueries in the WHERE clause *https://issues.apache.org/jira/browse/HIVE-1799*.

[2]The complete list of operators can be found in https://github.com/apache/hive.

in *HiveQL Query-1* we have the *Stage-7*, which depends on *Stage-1* and consists of the *Stage-4*, *Stage-3* and *Stage-5*. Also in *HiveQL Query-1*, the *Stage-6* depends on *Stage-5*. The *Stage-0* depends on *Stage-3*, *Stage-4* and *Stage-6*. Finally, we have the *Stage-2*, which depends on *Stage-0* and is the last stage which aggregates the output and save it into the intermediate table *supplier_tmp*. The other two queries have an analogous behavior.

The stages with the *MapReduceLocalWork*, *MapReduce*, *Move* and *Stats-Aggr* operators are executed locally and do not need to be sent to the Hadoop framework for distributed execution. In this dissertation, we are only tuning the *stages* that are sent to Hadoop (e.g., ReduceSink, Extract, Filter). In Figure 2.4 we observe 18 *stages* for HiveQL Query 16, but only five are executed as jobs on Hadoop (i.e., *HiveQL Query-1, Stage-1*; *HiveQL Query-2, Stages-3,5*; and *HiveQL Query-3, Stages-1,2*).

```
1  -- HiveQL Query-1
2  INSERT OVERWRITE TABLE supplier_tmp
3  SELECT s_suppkey
4  FROM supplier
5  WHERE NOT s_comment
6  LIKE '%Customer%Complaints%';
7
8  -- HiveQL Query-2
9  INSERT OVERWRITE TABLE q16_tmp
10 SELECT p_brand, p_type, p_size, ps_suppkey
11 FROM partsupp ps JOIN part p
12 ON p.p_partkey = ps.ps_partkey
13 AND p.p_brand <> 'Brand#45'
14 AND NOT p.p_type
15 LIKE 'MEDIUM POLISHED%'
16 JOIN supplier_tmp s
17 ON ps.ps_suppkey = s.s_suppkey;
18
19 -- HiveQL Query-3
20 INSERT OVERWRITE
21 TABLE q16_parts_supplier_relationship
22 SELECT p_brand, p_type, p_size,
23    COUNT(distinct ps_suppkey) AS supplier_cnt
24 FROM (SELECT *
25    FROM q16_tmp
26    WHERE p_size = 49
27    OR p_size = 14 OR p_size = 23
28    OR p_size = 45 OR p_size = 19
29    OR p_size = 3 OR p_size = 36
30    OR p_size = 9 ) q16_all
31 GROUP by p_brand, p_type, p_size
32 ORDER by supplier_cnt DESC,
33          p_brand, p_type, p_size;
```

Code 2.4: TPC-H 16 query translated to HiveQL [29].

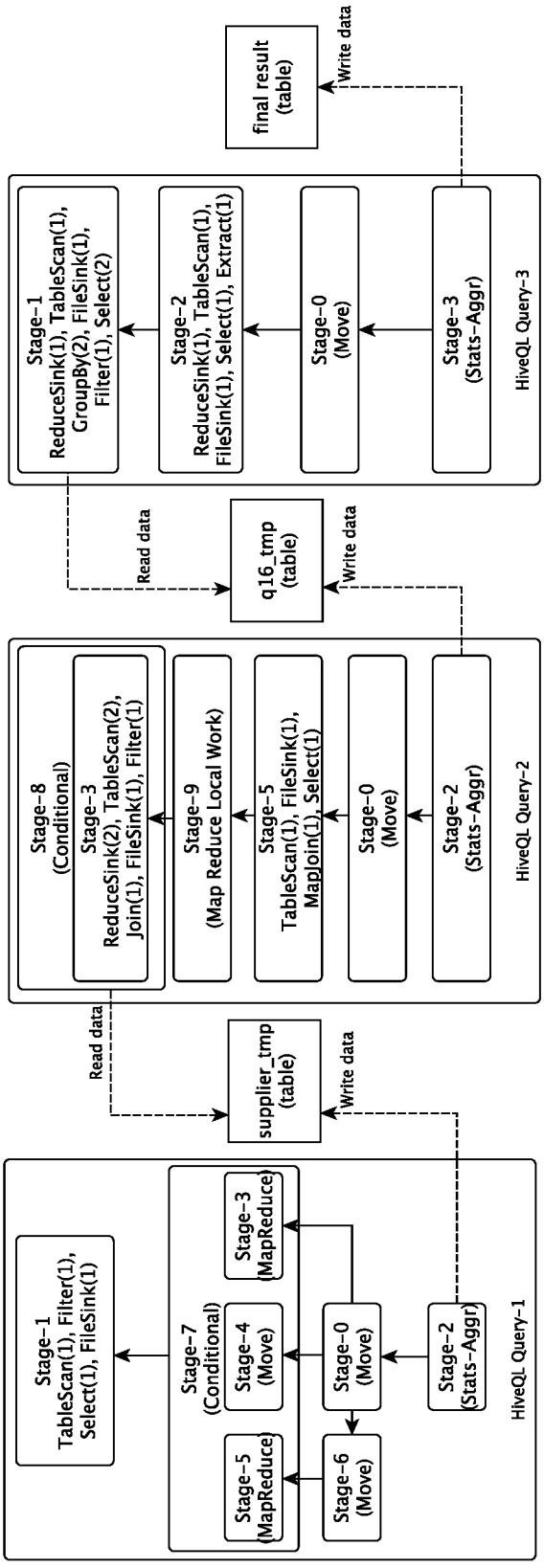

Figure 2.4: This figure illustrates the stage workflow produced by Hive after the translation of the TPC-H query 16 written in HiveQL to the Direct Acyclic Graph of stages. Full lines represent the dependency among the *stages*. Dashed lines represent the read and write operations over tables.

CHAPTER 3

RELATED WORK

The Hadoop framework and Hive provide together more than four hundred tuning knobs that can be used to boost performance. However, setting the appropriated setup is challenging due to the number of variables involved that vary from cluster workload and input data size to algorithms for compression and degree of parallelism. Furthermore, some Hadoop applications consists of chained jobs such as the PageRank, Indexing, Bayes Classification and Hive queries. Each job into a chain may have a completely different behavior from any other job in the same chain and should receive a specific tuning. Indeed, Yang et Al. [34] exposed the correlation among tuning knobs, such as the *io.sort.factor* (i.e., the number of streams to merge at once while sorting files), which influences on the *maximum.reduce.tasks* (i.e., the maximum concurrent reduce tasks per *TaskTracker*).

Taking into account this scenario, once the task of applying the best setup is delegated to the developer, misconfiguration may happen and lead to poor performance. The related work consists of the four main tuning approaches that aim to discover the best setup values. In Section 3.1 we present the *Rule-based* tuning systems. In Section 3.2 we present the systems based on *Simulation*. In Section 3.3 we present the tuning systems based on *Log Analysis*. In Section 3.4 we present the tuning systems based on *Profiling*. Finally, we briefly discuss about the presented tuning systems in Section 3.5.

3.1 Rule-based

Vaidya [30] is a sub-project of Hadoop, which main goal is to diagnose the performance of Hadoop jobs. It is a rule based performance diagnostic tool that performs a "post-mortem" analysis of each job execution. Vaidya collects and parses statistics about the executions from log and configuration files. Then, Vaidya executes predefined rules against these statistics to diagnose performance problems. The Vaidya system generates a report

to the administrator or developers based on the results from the execution of the rules.

Another attempt to achieve better performance by applying better setup are the rules-of-thumb, which have been proposed by the Hadoop community based on administrators and developers experience. The rules-of-thumbs such as Intel [16], AMD [3] and Cloudera [20] tips are presented in Table 3.2. In Table 3.1 we show the tuning knobs exposed by Yang et. Al. [34] as the group of knobs that has more influence on performance. The main problem of the rules-of-thumb is that they are not intended to be precise or reliable for every job once they are based on administrators and developers experiences.

3.2 Simulation

Simulating Hadoop jobs under determined conditions (e.g., cluster workload, scheduling algorithms, hardware and different input data) allows varying and choosing the best setups accordingly. However, simulation requires an accurate system, which may not address events that only happens in real clusters. There are efforts to build simulation systems to predict optimal setup including WaxElephant [25], MRPerf [33, 5], SimMapReduce [28] and HSim [12, 21].

The WaxElephant [25] has four main features: (1) loading Hadoop workloads derived from logs, and replaying the jobs from these workloads; (2) synthesizing workloads and executing them based on their statistical characteristics, (3) identifying the best setup, and (4) analyzing the scalability of the cluster.

The MRPerf [33, 5] is a simulation approach designed to explore the impact of MapReduce setups, which captures the various settings of MapReduce clusters such as tuning parameters and application design in order to answer questions, such as: does a given setup yield a desired I/O throughput? MRPerf is based on Network Simulator 2 [23] (NS-2) and DiskSim [11].

The SimMapReduce [28] is based on GridSim [24] and SimJava [7]. It is designed for resource management and performance evaluation. The HSim [12, 21] models the Hadoop knobs that can affect node's behavior and uses these models to tune the performance of a Hadoop cluster.

Tuning Knob	Default Value	Description
dfs.replication	2	The number of block replication.
dfs.block.size	64 mb	The block size for new files.
io.sort.mb	100 mb	The total amount of buffer memory while sorting files.
mapred.child.java.opts	200	Java options for the task tracker child processes.
io.sort.record.percent	0.05	The percentage of io.sort.mb dedicated to track record boundaries.
io.sort.spill.percent	0.80	The soft limit in either the buffer or record collection buffers.
io.sort.factor	10	The number of streams to merge at once while sorting files.
mapred.compress.map.output	false	If the outputs of the maps should be compressed before being sent across the network.
io.file.buffer.size	4096	The size of buffer for use in sequence files.
maximum.map.tasks	2	The maximum concurrent map tasks per TaskTracker.
maximum.reduce.tasks	2	The maximum concurrent reduce tasks per TaskTracker.
mapred.reduce.parallel.copies	5	The number of parallel transfers run by reduce during the shuffle phase.
mapred.job.shuffle.input.buffer.percent	0.70	The percentage of memory to be allocated from the maximum heap size to storing map outputs during the shuffle.
mapred.job.shuffle.merge.percent	0.66	The percentage of total memory allocated to store in-memory map outputs.
mapred.job.reduce.input.buffer.percent	0.0	The percentage of memory relative to the maximum heap size to retain map outputs when shuffle is concluded.
mapred.output.compress	false	Compress the job output.
mapred.output.compression.type	record	Compress the map output at level of record or block.
mapred.map.output.compression.codec	DefaultCodec	Coded used to compress map output.

Table 3.1: List of Hadoop tuning knobs with their default values.

These simulation approaches focus on the simulation of cluster's environment, enabling the replacement of internal structures for development of new modules and/or testing.

Tuning Knob	Rule-of-thumb
io.sort.mb	(metadata size of 16 bytes * (block size in bytes / average record size)) + block size in mb (e.g., for a block of 128mb, $((16*(128*(2^{20})/100))/(2^{20}))+128$)
io.sort.factor	Ensure that is large enough to allow full use of buffer (i.e., io.sort.mb) space.
io.sort.record.percent	16 / (16 * (average record size, i.e., divide map output bytes by map output records), e.g., 16/(16*100))
io.sort.spill.percent	Threshold at which io.sort.mb buffer starts to be spilled to the disc. Large values avoid extra disc operations.
io.file.buffer.size	The size of this buffer should probably be a multiple of hardware page size (i.e., 4096 on Intel x86), and it determines how much data is buffered during read and write operations.
mapred.job.reduce.input.buffer.percent	Retain map outputs before sending them to the final reduce function of the reduce phase.
mapred.job.shuffle.input.buffer.percent	Increasing this buffer avoid spills to disc at copying map's output.
mapred.job.shuffle.merge.percent	Threshold at which mapred.job.reduce.input.buffer.percent starts to be spilled to the disc. Large values avoid extra disc operations.
mapred.reduce.parallel.copies	Large values for large input data may enhance the copy of intermediate data. But large values increase CPU usage.
mapred.output.compress	true
mapred.output.compression.type	Record
mapred.output.compression.codec	Best compression related to the input data (e.g., SnappyCodec).
mapred.compress.map.output	true
mapred.map.output.compression.codec	Best compression related to the input data (e.g., SnappyCodec).

Table 3.2: Rules-of-thumbs.

However, these systems are not specific designed to search for the best tuning parameters.

3.3 Log Analysis

The Hadoop execution logs several information that can be used for performance prediction, including how many maps and reduce tasks have been executed, the number of bytes

produced per each processing phase and the time spent in each phase. Tuning systems, including PerfXPlain [17], Mochi [27], MR-Scope [15], Theia [8] and Rumen [31] base their tuning approaches on log file analysis. The main problem of log file analysis it that the developer must wait the complete execution of the job to known the best setup.

The PerfXPlain [17] tuning system introduces the PXQL language, which allows users to formulate queries about the performance of MapReduce jobs and tasks. It consists of a pair of jobs and three predicates. The first two predicates describe the observed behavior of the jobs with their description provided by the user. The third predicate is the expected behavior and its description provided by the user.

Given the two jobs (J_1, J_2), and the predicates (p_1, p_2, p_3), where the predicate $p_1(J_1, J_2) = true$, and $p_2(J_1, J_2) = true$, but $p_3(J_1, J_2) = false$. The following the PXQL syntax presented in Code 3.1 performs the PXQL, and a possible result is the $p2$ predicate to be of the form OBSERVED duration_compare = SIMILAR and the $p3$ to be of the form EXPECTED duration_compare = GREATTHAN. It means that the J_1 and J_2 had a similar execution time, but the user expected J_1 (i.e., *duration_compare*) to be slower than J_2 (i.e., *duration_compare*). The key idea of PerfXPlain is to identify the reasons why the jobs J_1 and J_2 performed as observed rather than performing as expected.

```
1 FOR J1, J2 WHERE J1.JobID = ? and J2.JOBID = ? DESPITE p1 OBSERVED p2 EXPECTED p3
```

Code 3.1: PXQL Example [17].

Mochi [27] is a visual log file analysis tool for debugging performance of Hadoop jobs. It constructs visualizations about the cluster from log entries collected from each cluster node during jobs executions. These visualizations consists of space (i.e., nodes), time (i.e., duration, times, execution sequences) and volume (i.e., size of data processed). The visualizations are correlated across the nodes to build a unique causal representation, i.e., a graph with vertices's representing processing stages and data items, and edges representing durations and volumes. Figure 3.3 represents the path of a single job execution from the graph. Finding jobs with similar path of execution enables administrators and developers to share tuning. Also, visualizing the performance of the Hadoop jobs enables administrators and developers to adjust tuning manually. However, this tuning is applied

to the whole job execution.

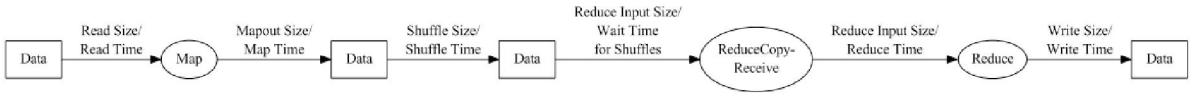

Figure 3.1: Path of a single job execution from the graph [27]

MR-Scope [15] is a tracing system, which provides a visualization of on-going jobs and a visualization of the distribution of the file system blocks and its replicas. The main goal of MR-Scope is digging Hadoop to trace the sub-phases of every job, showing (1) the output size in order to achieve better policies for data distribution, and (2) the time spent per each job. The authors point out that observing these two points are important to take a snapshot of the cluster's performance in order to adopt any optimization method.

The Rumen [31] system is a sub-project of Hadoop designed to extract and analyze log file entries from past Hadoop jobs. It is a built-in tool in Hadoop that performs log parsing and analysis at job level. Once the Hadoop logs are often insufficient for simulation and benchmarking, Rumen uses job information, such as job execution time and job failures to produce condensed logs in JSON format. Rumen generates log information that can be used for debug, performance diagnoses, or to feed simulator and benchmark systems such as GridMix [1] and Mumak [2].

3.4 Profiling

The profiling approach consists of collecting concise information from jobs executions to create *job profiles*, which are used along with search heuristics, knowledge bases or cost-based approaches to search for the best setup. The tuning systems based on the profiling approach, includes Starfish [14], AROMA [19], Adaptive Framework [18] and CPU Pattern Matching [26]. These systems are based on mechanisms to collect support information gathered during job execution.

The support information represents a set of resource consumption characteristics, such as: CPU, network and disk consumption as well as statistics from the *Map* and *Reduce*

phases and sub phases. Table 3.3 shows the support information collected by each profiling-based tuning system.

Tuning System	Support information
Starfish	Statistical information about *Map* and *Reduce* phases and sub phases.
Adaptive Framework	Resources usage; environment and Hadoop configuration.
CPU Pattern Matching	CPU usage pattern.
AROMA	CPU, network and disk usage pattern.

Table 3.3: The support information collected by the profiling-based tuning systems.

Starfish collects support information from an user-defined fraction of the *Map* and *Reduce* tasks. After the creation of the *profile*, Starfish uses the Random Recursive Search (RSS) algorithm along with the support information to sift through the possible setup space in order to find the best setup. The RSS uses a cost model defined by Herodotou et Al. [13], called *What-if* engine, to determine whether the setup is better or not.

The Adaptive Framework, AROMA and CPU Pattern Matching systems are analogous, varying in the type of support information used to construct the *profile* and in the cost model. Table 3.4 shows the methods and tools used by each tuning system to collect support information. Table 3.5 presents the algorithms used by each tuning system to search for the best setup values through the possible setup space.

Tuning System	Method or tool used to collect support information.
Starfish	Dynamic instrumentation (BTrace)
Adaptive Framework	Sun Grid Engine (SGE)
CPU Pattern Matching	SysStat tool
AROMA	Hadoop logs and dstat tool

Table 3.4: The method or tools used by the profiling-based tuning systems to collect support information.

The AROMA tuning system groups jobs into clusters enabling the usage of different performance model per cluster. It learns only one performance model once a new cluster is identified. One problem of the AROMA is that it needs to keep a staging cluster to collect the support information. Moreover, it runs the new jobs in the staging cluster with a fraction of the input data (i.e., 10% of the total input data size), which may lead

Tuning System	Algorithms used to search for best setup values.
Starfish	Recursive Random Search
Adaptive Framework	Utility Function
CPU Pattern Matching	Dynamic Time Warping
AROMA	Support Vector Machine

Table 3.5: The methods used by the profiling-based tuning systems to search for the best setup values.

to different resource consumption patterns.

A *job profile* represents the resource consumption pattern, based on the quantity and quality of support information collected. Construct *job profiles* with enough information increases the overhead of the overall tuning process. As example, Starfish adds a minimum overhead of less than 5% of the overall execution time to profile 10% of the tasks. But Starfish takes 50% of the overall execution time to profile 100% of the tasks. Table 3.6 shows the methods used by each tuning system to enable comparison among jobs in order to retrieve the best setup values from past *profiles*.

Tuning Approach	How does it compare jobs?
Starfish	Using job profiles
Adaptive Framework	Searching through a knowledge base
CPU Pattern Matching	Calculating Correlation Coefficient among CPU patterns
AROMA	Clustering jobs with k-medoid with LCSS

Table 3.6: Method used to compare jobs in order to reuse or search for the best setup values.

3.5 Discussion

Log Analysis [17, 27, 15, 8, 31], Profiling [14, 19, 18, 26], Simulation [5, 33, 28, 12, 25, 21] and Rule-based [20, 16, 3, 30] approaches search for the best setup values analyzing support information collected during the execution or simulation of the jobs, or from log files.

The Log Analysis approach needs the complete execution of the job to predict best setups. This approach is optimal to find bottlenecks in the infrastructure and failures on hardware and software components. However, the best setups found can only be applied

in further jobs and not in the current ones.

The Rules-of-thumbs [20, 16, 3] (i.e., excluding Vaidya) is the only approach that performs tuning in advance and do not require support information. However, the Rules-of-thumbs are not intended to be precise or reliable for every job once they are based on administrators and developers experiences.

The *Profiling* approach needs to collect support information during jobs execution, which add a costly overhead in the whole tuning process. The Simulation approach focus on the simulation of the environment to enable the replacement of internal structures for development of new modules and/or testing. The simulation-based tuning systems are not specific designed for tuning, have several limitations in simulating and add a costly overhead, once the job must be executed (i.e., simulated) to be tuned.

Figure 3.2 shows the tuning life-cycle followed by the current approaches. As illustrated, in the studied approaches the administrator or developer must send the job to execution and, afterwards, use the best setup found. In our approach, we present a new tuning life-cycle to avoid the execution or simulation of the jobs and to perform tuning to be applied in the current job being executed.

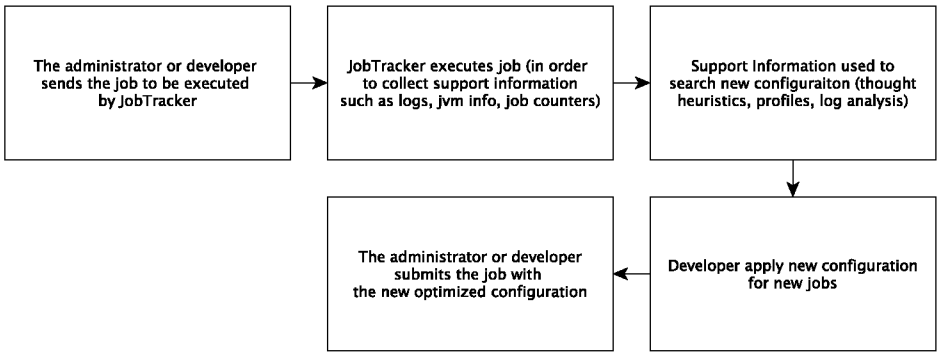

Figure 3.2: Tuning life-cycle of the presented Hadoop tuning systems.

In our approach, the user sends the job to be executed by the Hadoop framework with or without any configuration. The *JobTracker* receives the job and sends it to our solution, called AutoConf. The AutoConf extracts the collection of operators from the given job and search for the corresponding *cluster*. In case there is a corresponding *cluster* with same collection of operators, AutoConf applies tuning in the job. In case there are not a

corresponding cluster, the job is executed with the configuration set by the administrator or developer (See detailed description of AutoConf execution in Section 4). Note that in our approach we do not need to execute the job in the Hadoop cluster to collect support information. Instead, we use the information provided by the job (i.e., its operators).

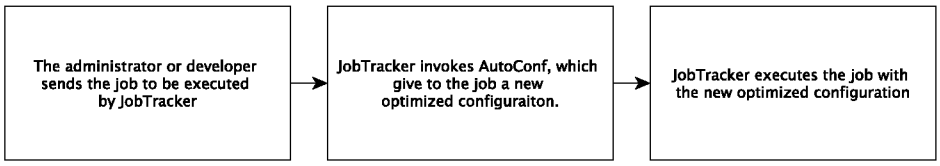

Figure 3.3: Tuning life-cycle of our approach.

CHAPTER 4

HIVEQL SELF-TUNING

Our HiveQL Self-Tuning approach is complementary to the existing Hadoop tuning systems, once it relies on the decision of which query (or stage) should receive determined tuning based on the code signature. Our approach relies on *many-to-one* configuration, adapting the setup values during query execution for the internal stages of each query, i.e., per-stage tuning. We present in Section 4.1 the definition of the set of *clusters*. We detail the clustering algorithm in Section 4.2. We present the *Intra-Query* and *Inter-Query* Tuning in Section 4.3. Finally, we present the architecture of our solution 4.4.

4.1 Stage Clusters

Our approach uses dynamic analysis, i.e., extract the collection of operators from the stages of the query, during query execution in order to identify the code signature of each stage and to perform a per-stage tuning. We clustered the stages with the same code signature (i.e., collection of *operators*) from a common database workload. The resulting set of *clusters* of code signatures is defined as the set $\mathbb{C}$, where each *cluster* c_i is of the form $\{\varphi, \omega\}$. We define φ as a collection of *operators* used per various *stages* across the queries, and ω are the setup values to be applied.

4.2 Clustering Algorithm

Algorithm 1 applies a per-stage tuning for each running *query*. Given the set of *clusters* $\mathbb{C}$, for each given *query* in the space of possible queries $\mathbb{Q}$, while exists a query q_i to be processed, get the *stages* $\{s_0, \ldots, s_n\}$ from q_i, extract the collection of operators $\varphi = \{Op_0, \ldots, Op_n\}$ from running stage s_j. In case there are $\{\varphi, \omega\}$ in $\mathbb{C}$, where *operators* match with φ, retrieve ω and apply in s_j. In case $\varphi \notin \mathbb{C}$, add φ in $\mathbb{C}$. Otherwise,

ω for the new collection of φ is provided by an external tuning system (e.g., Starfish, rules-of-thumb).

Algorithm 1: HiveQL Tuning

$\mathbb{C} = \{ c_0, \ldots, c_k : \text{c is a } \textit{cluster}, \text{ where } \textit{cluster} \text{ is a list of the form } \{\varphi, \omega\}, \text{ where the } \varphi \text{ are the index of the list and the } \textit{setup values} \text{ are the appropriate tuning for the referent group of } \varphi$

$\mathbb{Q} = \{ q_0, \ldots, q_k : \text{q is a HiveQL query } \}$

$\text{query} = \{ s_0, \ldots, s_n : s_i \text{ is a stage} \}$

$\text{stage} = \{ op_0, \ldots, op_n : op_i \text{ is an operator} \}$

while $\exists\ q_i \in \mathbb{Q}$ **do**

  for $s \in q_i$ **do**

    $operators \leftarrow$ extract φ from s_j

    if $operators \in \mathbb{C}$ **then**

      $s_j \leftarrow \omega$ from C_φ

    else

      $\mathbb{C} \leftarrow$ create new cluster base on φ

  end

end

Table 4.1 presents the 29 clusters created for the database workload provided by the TPC-H benchmark (details on experiments are further presented in Section 5). For instance, consider a query q_x with the following signature $\varphi = \{2\ \textit{MapJoin},\ 1\ \textit{Select},\ 2\ \textit{TableScan},\ 1\ \textit{FileSink}\}$. Since queries with the same signature will be clustered together, q_x will be placed in cluster 26 and will leverage the same ω along with other 3 queries (e.g., $\omega = \{io.sort.mb = 74, io.sort.factor = 7\}$).

4.3 Intra-Query and Inter-Query Tuning

Each HiveQL query is split into several stages, where each stage has a different collection of SQL-like operators and a distinct use of computational resources. In this context, tuning means applying a single configuration for each stage, i.e., *many-to-one* configuration,

CLUSTER	MAP JOIN	FILTER	PTF	SELECT	SCRIPT	COMMON JOIN	LIMIT	HASH TABLE DUMMY	FORWARD	LATERAL VIEW JOIN	LATERAL VIEW FORWARD	EXTRACT	GROUP BY	MAP	UNION	UDTF	OPERATOR	TABLE SCAN	FILE SINK	REDUCE SINK	HASH TABLE SINK	#STAGES
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	7
3	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	1	1	0	9
4	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2	1	2	0	2
5	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	0	16
6	0	0	0	1	0	0	0	0	0	0	0	0	2	0	0	0	0	1	1	1	0	1
7	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	6
8	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	1	1	1	0	1
9	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	2	1	2	0	6
10	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	2	1	2	0	2
11	0	0	0	2	0	0	0	0	0	0	0	0	0	0	2	0	0	2	2	0	0	1
12	0	0	0	2	0	0	0	0	0	0	0	0	2	0	0	0	0	1	1	1	0	7
13	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2	1	2	0	1
14	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	2
15	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	2	1	2	0	10
16	0	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	2	1	2	0	5
17	0	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	3	1	3	0	1
18	0	1	0	2	0	0	0	0	0	0	0	0	2	0	0	0	0	1	1	1	0	10
19	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2	1	2	0	1
20	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	4
21	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	4
22	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	1
23	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	3
24	1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	5
25	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	1
26	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	6
27	2	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	1
28	2	0	0	2	0	0	0	0	0	0	0	0	1	0	0	0	0	2	1	0	0	2
29	2	2	0	2	0	0	0	0	0	0	0	0	1	0	0	0	0	2	1	0	0	2

Table 4.1: The clusters and the occurrence of the operators from each stage of the 22 TPC-H queries executed for Scale Factor of 1, 10 and 50.

instead of applying a unique configuration for the whole query, i.e., *one-to-one* configuration. We named this tuning approach as *Intra-Query Tuning*, which enables one query

to have many configurations.

Figure 4.1 illustrates the network consumption pattern of *cluster-2* for the TPC-H clustering. We observe that the *stages* in *cluster-2* have similar behavior, independent of the *input data size*. Due to this pattern, these *stages* should share the same network tuning. Another characteristic of our approach is the *Inter-Query Tuning*, which enables queries to share acquainted setup values from stages of other queries.

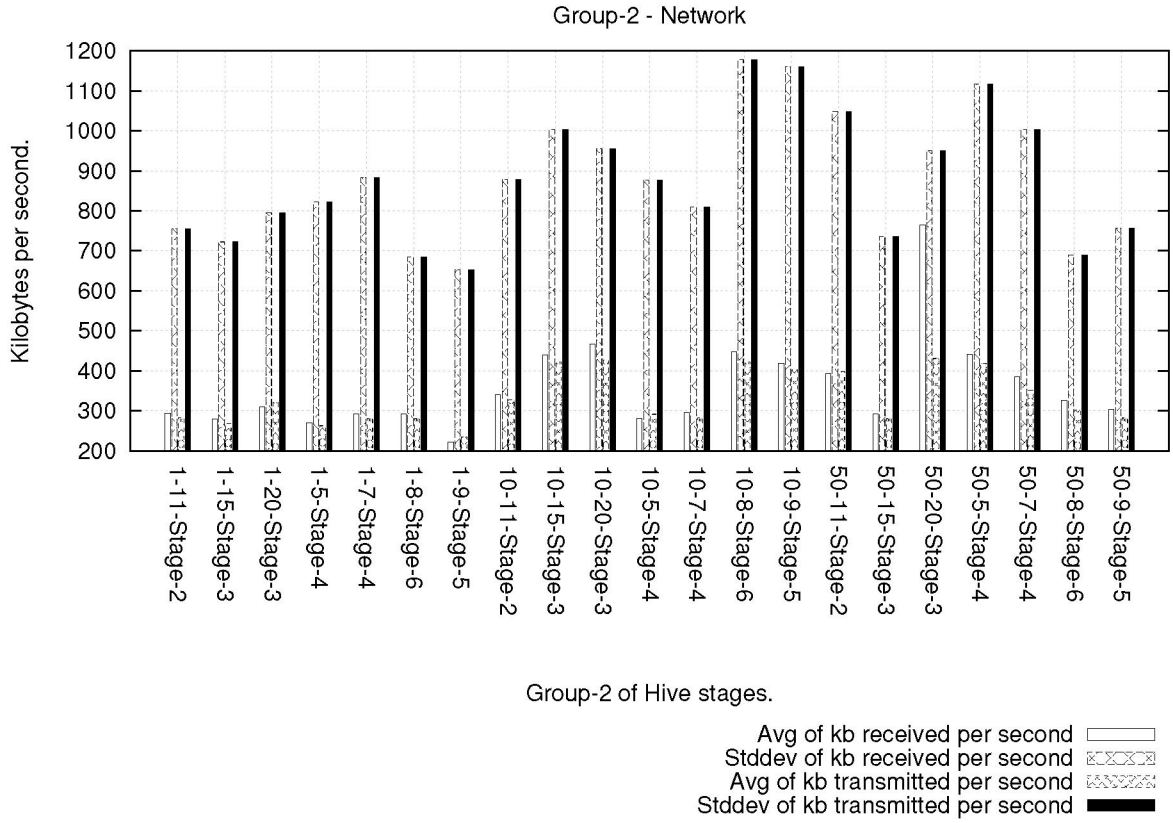

Figure 4.1: The network consumption from the stages of TPC-H queries from *cluster-2*. The X axis label is of the form Scale_factor-Query-Stage_name.

Table 4.2 shows the stages from the TPC-H Query 16 and the other TPC-H queries, in which a common signature can be found. The *HQL1* refers to the first query from the HiveQL query 16. The *HQL1-Stage1* have been clustered in *cluster-14*. The *HQL2* refers to the second query from the HiveQL query 16. The *HQL2-Stage3* have been clustered in *cluster-1*, and the *HQL2-Stage5* have been clustered in *cluster-21*. The *HQL3* refers to the third query from the HiveQL query 16. The *HQL3-Stage1* have been clustered in *cluster-18*, and the *HQL3-Stage2* have been clustered in *cluster-7*. The number of *stages*

may varying accordingly to the input data, i.e., Hive splits one *stages* in two or more stages in case the input data size is large.

	TPC-H Query 16				
TPC-H Query	HQL1-Stage1	HQL2-Stage3	HQL2-Stage5	HQL3-Stage1	HQL3-Stage2
q1	0	0	0	1	1
q2	0	0	1	0	0
q4	0	0	0	1	1
q5	0	0	1	0	0
q6	0	0	0	1	0
q8	0	0	1	0	0
q9	0	0	1	0	0
q10	0	1	0	0	0
q12	0	0	0	0	1
q13	0	0	0	0	1
q15	0	0	0	1	0
q16	1	1	1	1	1
q20	0	0	0	3	0
q21	0	0	0	1	0
q22	1	0	0	1	1

Table 4.2: Number of *stages* with same collection of *operators* between TPC-H queries and TPC-H query 16 translated to HiveQL.

4.4 AutoConf: The HiveQL Tuner

In this section, we present the architecture of our approach, called AutoConf, which is responsible for analyzing and clustering running queries. Figure 4.2 illustrates the architecture and the interaction of AutoConf with Hadoop and Hive. AutoConf consists of three modules: (1) *Feature Extractor*, which is responsible for extracting the code signatures (i.e., the collection of operators) from each query (2) *Clustering*, which is the module responsible for finding similar *clusters* for signatures, and (3) *Tuner*, which applies the appropriated tuning to queries.

The collection of operators are extracted from the query plan that is inside the job object sent to Hadoop. The *Feature Extractor* module, reads the collection of operators and sums the occurrence of each operator saving these informations into the *list of operators*. Next, the *Feature Extractor* sends the *list of operators* to the *Clustering* module.

The *Clustering* module loads the set of *clusters* $\mathbb{C}$ from the disk (explained in Section 4.1). When the *Clustering* module receives $\mathbb{C}$, it searches if there is a cluster with same

code signature. In case there is an equivalent *list of operators*, the *Clustering* module sends the job to the *Tuner* module, which, in turn, reads the setup values from disk and apply this setup to the given job. Reading the setup values from disk enables the modification of the values during execution (i.e., "on-the-fly" adaptation).

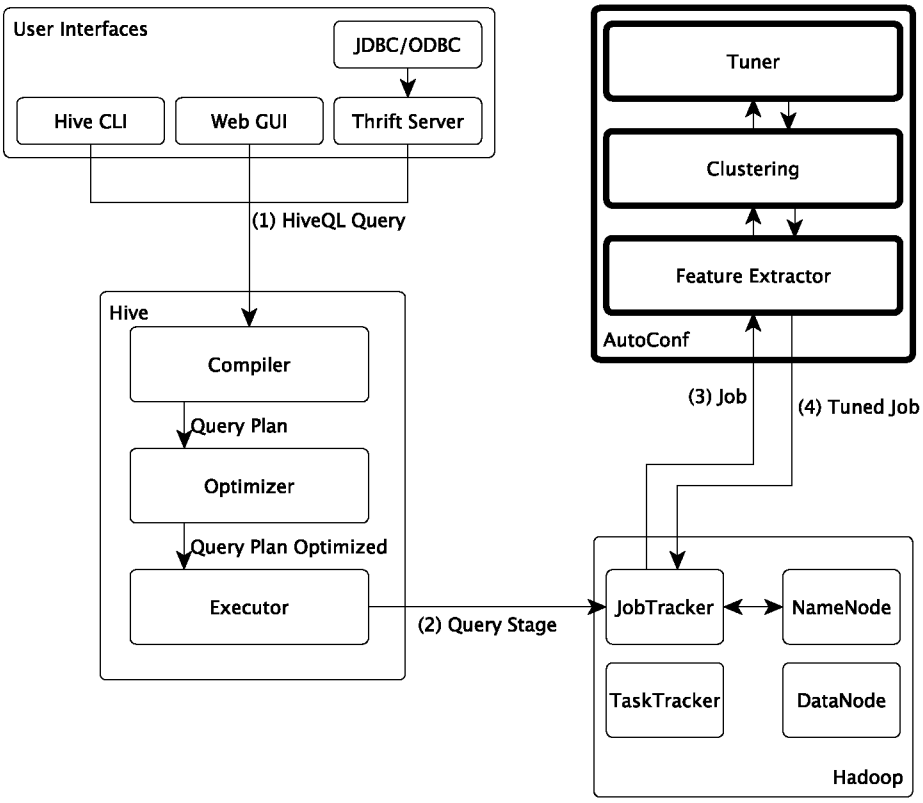

Figure 4.2: the AutoConf Architecture inside the Hadoop and Hive ecosystem.

CHAPTER 5

EXPERIMENTS

Experiments have been conducted to validate our approach. We ran all experiments on a cluster of 10 machines, where each machine is a x86_64 bits, with 2 processors Intel®Core™2 Duo CPU E8400 @ 3.00 GHz, 4Gb of memory and a hard disk of 500 Gb, 7200 RPM ATA. Experimental results were obtained by executing TPC-H against databases generated with DBGen Scale Factor of 1 Gb, 10 Gb and 50 Gb. We execute the appropriate TCP-H benchmark designed for Hive against three configuration setups, such as: the *standard Hadoop setup*, the *rules-of-thumb*, and the *rules-of-thumb* applied in a per-stage basis. We used Java version 1.6, Hadoop version 1.1.2, Hive version 0.11.0 and the TPC-H translated to HiveQL. In Section 5.1 we discuss about how we created specific tuning for each cluster. In Section 5.2 we demonstrate that some clusters are straightforward dependent of the input data size. Finally, we present the results and evaluation in Section 5.3.

5.1 Tuning the Stage Clusters

The objective of this first experimentation is to create the *clusters* and define ω for each *cluster*. During the creation of the *clusters* we collect CPU, Memory, Network and Disk information using the SysStat package to determined the consumption pattern of each *cluster*. Figure 5.1 illustrates the CPU consumption pattern for *cluster-18*. Note that the CPU load average keeps low for stages from queries executed against database with Scale Factor of 1. However, as the Scale Factor increases, more the CPU load vary. The Figure 4.1 illustrate the network consumption for stages from *cluster-2*. In this case, independent of the Scale Factor, the network load average keeps low, but the standard deviation keeps high, which means that independent of the input size the network vary all the time during the execution of stages.

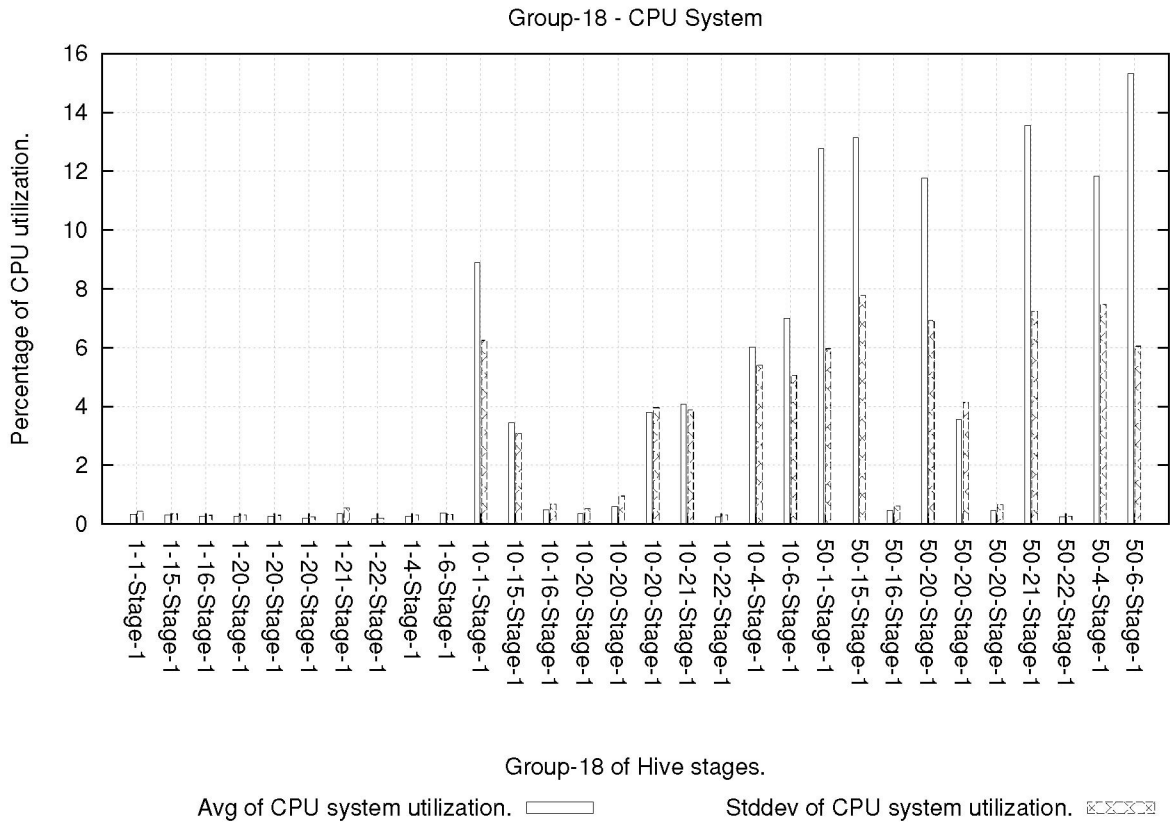

Figure 5.1: The CPU consumption from the stages of TPC-H queries from *cluster-18*. The X axis label is of the form Scale_factor-Query-Stage_name.

Based on the resource consumption pattern of each *cluster*, we defined the appropriated ω per-cluster tuning using the rules-of-thumb, i.e., tuning the CPU *tuning knobs* for the *clusters* that use more CPU than any other resource. The support information collected during the TPC-H execution have been analyzed in order to determined the use of computational resources per each *cluster*.

To conduct the analysis we divided the range of the resulting values from the support information in tertiles in order to label resource consumption pattern of the stages within the same cluster. In order to generate an optimized configuration for each cluster, we had to label the clusters accordingly to the resource cosumption pattern. As an example, in case the CPU consumption from all the stages within the same cluster goes from 0% to 16% of usage, the tertiles give us three different ranges, i.e., $\{0-5\}$, $\{6-10\}$ and $\{11-16\}$. The *clusters* where $\approx 90\%$ of the stages use $\{0-5\}\%$ of CPU have been labeled as using little CPU. The *clusters* where $\approx 90\%$ of the stages use $\{6-10\}\%$ of

CPU have been labeled as moderate use of CPU and the *clusters* where $\approx 90\%$ of the stages use $\{11 - 16\}\%$ of the CPU have been labeled as making intensive use of CPU. The analysis is analogous for network, disk and memory. Table 5.1 shows the values used to classify the resource consumption patterns for each cluster. Table 5.2 shows the result of the classification.

Use	CPU (%)	Memory (Gb)	Disk (Mb)	Network (Mb)
low	0-5	0-1	0-5	0-5
mid	5-10	1-1.5	5-10	5-10
high	>10	>15	>10	>10

Table 5.1: Range of values used to label the support information collection collected from the execution of the TPC-H queries executed agaist databases generated with DBGen, using Scale Factor of 1, 10 and 50 Gb.

We present in Table 3.1 the tuning knobs optimized in our experiments. The *low*, *mid*, and *high* labels were defined in Table 5.1. The compression knobs have been disabled during experimentation once forcing some stages to use compression and others to not use compression made some stages to fail.

5.2 Input-dependent tuning

The objective of this second experiment is to identify if the resource consumption patterns of some *clusters* change when the input data size grows. We found that the input data impacts the behavior of some *clusters*. As we observe in Figure 5.1, the more the input data grows (i.e., from 1Gb to 50Gb) the more CPU usage increases. In the other hand, in Figure 4.1 we observe that some computational resources do not change whenever the input size grows. This dependence of some *clusters* from the input data size reinforces the usage of self-tuning systems, once a Hadoop administrator or developer is not able to re-tune the same query every time the input size grows.

We did not take into account the input data size in our experiments in or order to change the configuration during execution, applying the same tuning for all *stages* clustered in the same *cluster* independent of their input data size. We applied the same tuning for all *stages* in the same cluster, once we want to verify the effectiveness of

Cluster	CPU	Memory	Disk	Network
1	low	mid	low	low
2	low	mid	low	low
3	low	mid	low	low
4	mid	low	high	high
5	low	mid	low	low
6	low	mid	low	low
7	low	mid	low	low
8	low	mid	low	low
9	high	low	high	high
10	mid	mid	high	high
11	low	low	low	low
12	high	mid	high	high
13	high	mid	high	high
14	low	low	low	low
15	high	mid	high	high
16	high	low	high	high
17	high	low	high	high
18	high	mid	high	high
19	high	low	high	high
20	low	mid	low	high
21	low	mid	low	low
22	low	mid	mid	mid
23	low	low	low	low
24	low	mid	low	mid
25	low	low	low	low
26	low	low	low	low
27	low	low	low	low
28	low	low	low	low
29	low	low	low	low

Table 5.2: The classification of the resource consumption patterns for all *clusters* from the TPC-H queries executed against databases with Scale Factor of 1, 10 and 50.

applying the rules-of-thumb using the *intra-query tuning*. Indeed, Yanpei et al. [35], after analyzing two years of logs from Cloudera and Facebook, discovered that 90% of the jobs accessed files with few gigabytes. Yanpei [35] also propose the creation of a cache police for those files.

5.3 Results and Evaluation

In this section the objective is to prove our hypothesis that we can avoid collecting support information by clustering stages with the same code signature and tuning them with the same setup. We present the TPC-H queries response time in Figures 5.2, 5.3, 5.4 and 5.5.

Table 3.1 shows the set of Hadoop tuning knobs used in our experiments. We executed five times the TPC-H queries against the Scale Factors of 1, 10, 50 and 100 Gb. The time represents the total execution time of a query, i.e., from its submission up to the return of the result. The execution time was registered with *time* package from Unix systems. The maximum and minimum values were removed from the five resulting times, then, we calculate the average of the three resulting times from each query.

Figure 5.2 shows the execution time for the TPC-H queries executed against databases generated with Scale Factor of 1 Gb. Note that there is almost no difference among the execution times for the queries using the *rules-of-thumb* and *rules-of-thumb* in a per-stage basis. Queries 6, 8, 10, 15, 18-21, had equal execution time. Most of the other queries differ in less than 3 seconds. This means that our approach had no difference from the *rules-of-thumbs* in the Scale Factor of 1. However, the queries 5, 7 and 9 executed with *standard Hadoop setup* were slow than the *rules-of-thumb* and *rules-of-thumb* in a per-stage basis.

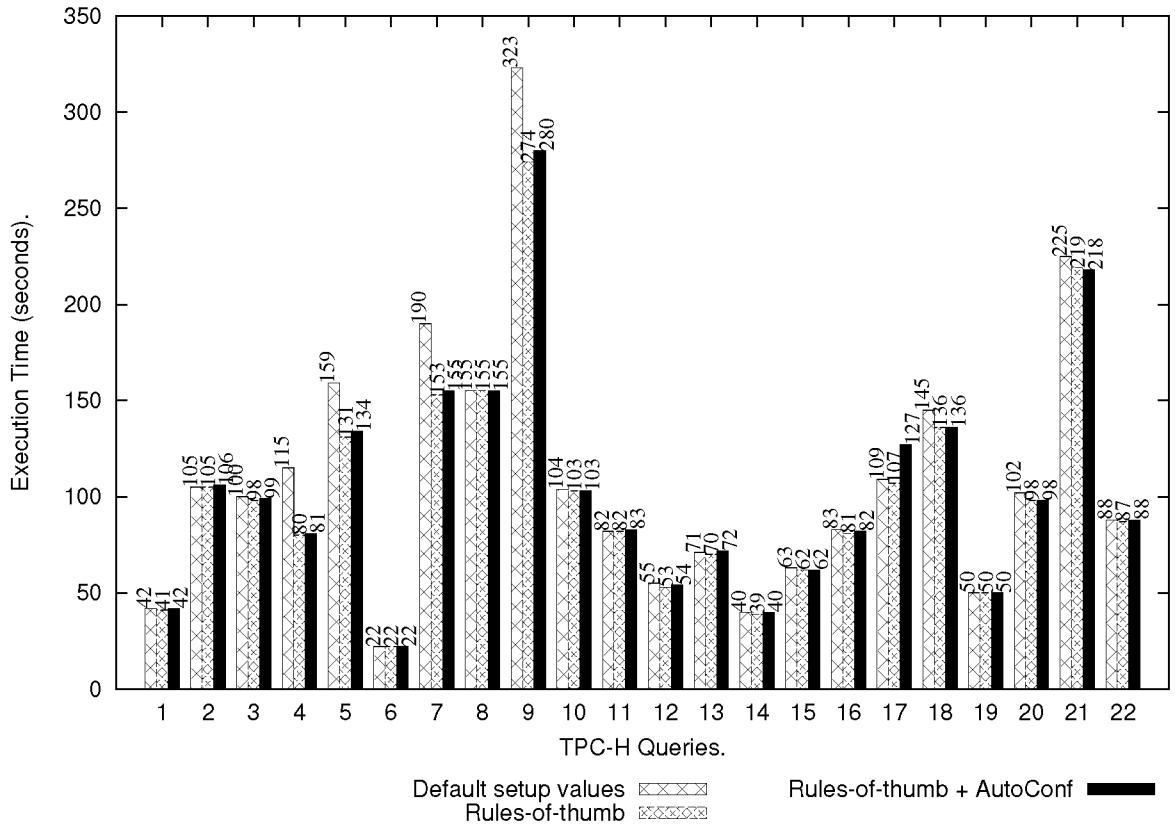

Figure 5.2: The average execution time of TPC-H queries against databases generated with DBGen for Scale Factor of 1.

Figure 5.3 shows the execution time for the TPC-H queries executed against databases generated with Scale Factor of 10 Gb. Note that all queries executed with the *rules-of-thumb* and *rules-of-thumb* in a per-stage basis had less than 10 seconds of difference in the execution time, which, in practice, means that our approach has no enhancement in queries against databases with less then 10 Gb. However, as we observe in Figure 5.3, the queries 2-5, 7-10, 12-14 and 16-22 executed with *standard Hadoop setup* were slower than the other two approaches, mainly the queries 5, 7, 9 and 21.

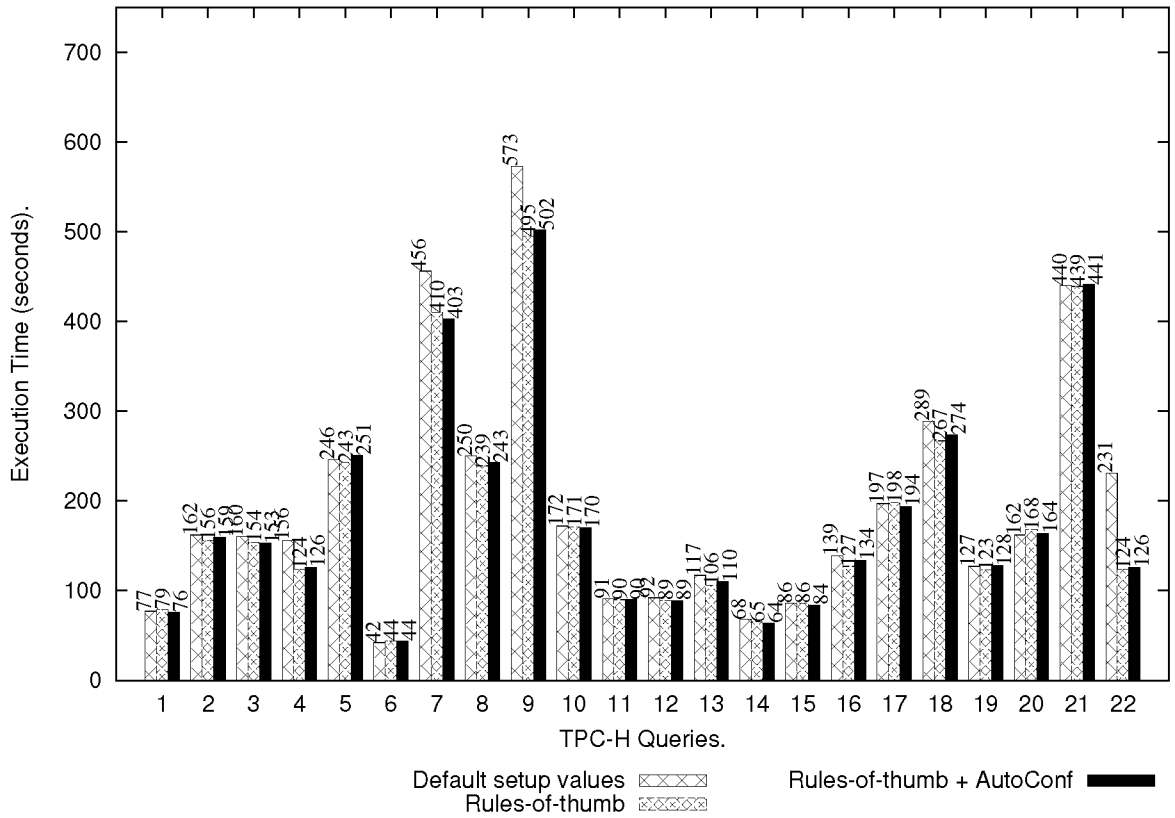

Figure 5.3: The average execution time of TPC-H queries against databases generated with DBGen for Scale Factor of 10.

Figure 5.4 shows the execution time for the TPC-H queries executed against databases generated with Scale Factor of 50 Gb. The queries executed with the *rules-of-thumbs* in a per-stage basis were decreased in 12 out of the 22 queries compared to the queries executed with the *rules-of-thumb*, including queries 2-5, 7-12, 16-18, and 22. In Figures 5.2, 5.3 and 5.4 we observe that as more the input data grows as more the queries executed with the *standard Hadoop setup* take to finish.

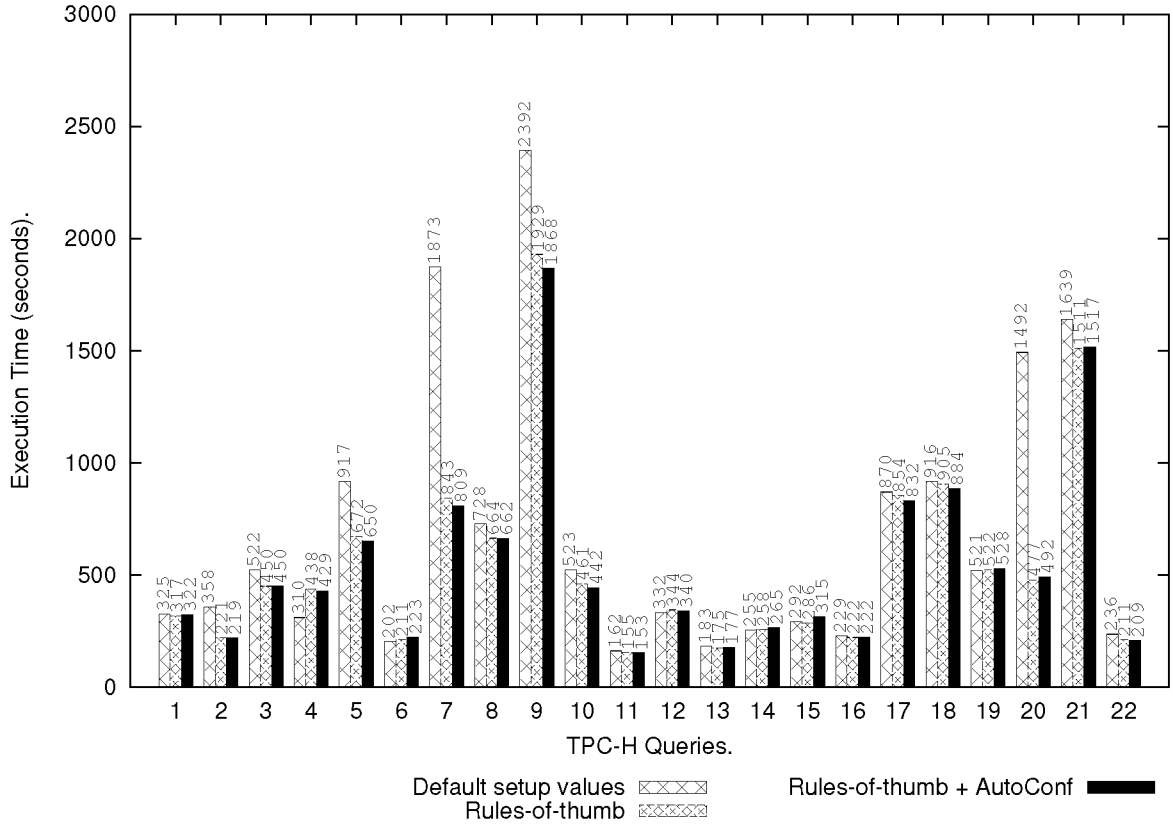

Figure 5.4: The average execution time of TPC-H queries against databases generated with DBGen for Scale Factor of 50.

Figure 5.5 shows the execution time for the TPC-H queries executed against databases generated with Scale Factor of 100 Gb. In this case, our approach improve 14 out of the 22 TPC-H queries, including queries 1-4, 6, 9-10, 12, 14, 16, and 18-22. Note that our approach enhanced query 19 in 136 seconds, query 20 in 120 seconds and query 21 389 seconds. Thus, we observed that as more the input data increases as more our approach enhances the overall query execution time.

5.4 Lessons learned

Throughout the experimentation we have learned two important lessons. First, small databases do not require much tuning effort for the Hadoop/Hive environment. Performance improvement showed small, therefore, using the *rules-of-thumb* is enough.

Second, results proved that we can tune HiveQL queries based on their code signature. Moreover, tuning can be extended to the stages of the queries, since they present different

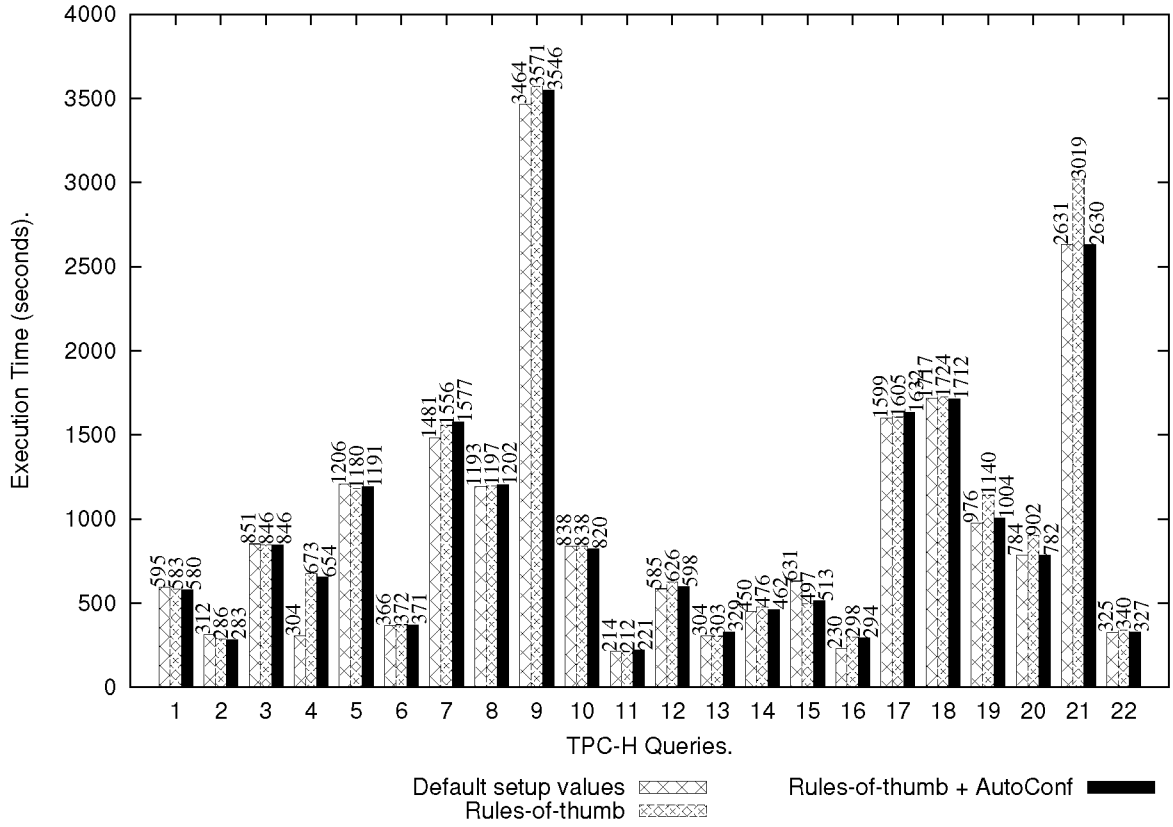

Figure 5.5: The average execution time of TPC-H queries against databases generated with DBGen for Scale Factor of 100.

behavior in terms of resource consumption. This observation proves our hypothesis.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The MapReduce programing model, and its open-source implementation, the Apache Hadoop, become in the last decade a *de facto* standard framework for processing large amounts of data in large clusters. Hive, Pig and several other data warehouse systems have been developed on top of Hadoop, taking advantage of the ability of Hadoop to run along thousands of commodity machines, and becoming an alternative to parallel database systems.

The interest in open-source mechanisms to process large amounts of data has pushed enterprises and researchers to spend efforts to optimize Hadoop and Hive systems. Our contribution is intended to be another brick in this optimization effort. Our contribution is based on the hypothesis that we can avoid collecting support information, made by the related work, by clustering query stages with the same code signature and tuning them with the appropriated configuration.

The support information collected by the related work remains as the guidance to search for the appropriated tuning. However, one problem of tuning systems based on support information is that they may add a costly overhead for tuning queries that are processed only once, such as *Ad-hoc* queries, once they have to be executed or simulated in order to collect such information. Our self-tuning system, called AutoConf, is a solution for tuning *Ad-Hoc* queries without add overhead, once it uses the information provided by the queries in advance, clustering them and applying the appropriated tuning before their execution.

In this dissertation we demonstrated that there are correlations among the stages inside different Hive queries by matching their code signature (i.e., collection of operators). By using the code signatures our approach enables the queries to share and reuse acquittance tuning. We identified 29 *clusters* (Table 4.1), which are used to optimize queries

in advance. Also, thought experimentation we noticed that the resource consumption patterns of some *clusters* change when the input data size grows. In this dissertation we did not take into account the variance on the input data size and applied the same tuning for all stages clustered in the same *cluster*.

Experimental investigation showed that the more the input data increases the more our approach decrease the overall query execution time. In addition, we observed (Figures 5.2, 5.3 and 5.4 and 5.5) that tuning query stages in small databases, even using our approach, is not effective. In our experiment we set a threshold for database sizes with less than 50 Gb. Future work is required in the following:

- Calculate, during workload execution, the threshold at which tuning is not effective.
- Provide appropriate setup based on the stages behaviors, using the information from past jobs to improve clustering.
- Provide new clustering techniques taking into account the variance on the input data size.
- Provide appropriate setup based on the input data (e.g., size, distribution) is another alternative to compose the clustering algorithm.
- Identify false-positives based on computational resource consumption of stages and/or cluster. The challenge is to identify false-positives before the execution of the stage.
- Extend our approach to a general model in order to classify not only HiveQL queries, but any incoming MapReduce job.
- Experiment AutoConf with new workloads to stimulate other operators and to create new clusters.

BIBLIOGRAPHY

[1] GridMix: benchmark for Hadoop clusters, `http://hadoop.apache.org/docs/mapreduce/current/gridmix.html`, 2013.

[2] Mumak: Map-Reduce Simulator, `https://issues.apache.org/jira/browse/MAPREDUCE-728`, 2013.

[3] AMD. Hadoop Performance Tuning Guide. Technical report, 2012.

[4] Joydeep Sen Sarma Ashish Thusoo. Hive- A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2009.

[5] A.R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–11. IEEE, September 2009.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2008.

[7] Ross Mcnab Fred Howell. simjava: A Discrete Event Simulation Library For Java.

[8] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: visual signatures for problem diagnosis in large hadoop clusters. pages 33–42, December 2012.

[9] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, August 2009.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, December 2003.

[11] Yale N. Patt Gregory R. Ganger, Bruce L. Worthington. The DiskSim Simulation Environment – Version 2.0 Reference Manual.

[12] Suhel Hammoud. MRSim: A discrete event based MapReduce simulator. In *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, volume 6, pages 2993–2997. IEEE, August 2010.

[13] Herodotos Herodotou. Hadoop Performance Models. *Statistics*, page 16, 2011.

[14] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, and Liang Dong. Starfish : A Self-tuning System for Big Data Analytics. *Systems Research*, (1862):261–272, 2011.

[15] Dachuan Huang, Xuanhua Shi, Shadi Ibrahim, Lu Lu, Hongzhang Liu, Song Wu, and Hai Jin. MR-scope: a real-time tracing tool for MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, page 849, New York, New York, USA, June 2010. ACM Press.

[16] Intel. Optimizing Hadoop Deployments. Technical report, 2010.

[17] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. PerfXplain: debugging MapReduce job performance. *Proceedings of the VLDB Endowment*, 5(7):598–609, March 2012.

[18] Martin Koehler, Yuriy Kaniovskyi, and Siegfried Benkner. An Adaptive Framework for the Execution of Data-Intensive MapReduce Applications in the Cloud, May 2011.

[19] Palden Lama and Xiaobo Zhou. AROMA: automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing - ICAC '12*, page 63, New York, New York, USA, September 2012. ACM Press.

[20] Todd Lipcon. 7 Tips for Improving MapReduce Performance, 2009.

[21] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. HSim: A MapReduce simulator in enabling Cloud Computing. *Future Generation Computer Systems*, 29(1):300–308, January 2013.

[22] Owen O Malley and Arun C Murthy. Winning a 60 Second Dash with a Yellow Elephant Hadoop implementation. *Proceedings of sort benchmark*, 1810(9):1–9, 2009.

[23] S Mccanne, S Floyd, and K Fall. ns2 (network simulator 2). http://www-nrg.ee.lbl.gov/ns/.

[24] Manzur Murshed Rajkumar Buyya. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing.

[25] Zujie Ren, Zhijun Liu, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. WaxElephant: A Realistic Hadoop Simulator for Parameters Tuning and Scalability Analysis. In *2012 Seventh ChinaGrid Annual Conference*, pages 9–16. IEEE, September 2012.

[26] Nikzad Babaii Rizvandi, Javid Taheri, and Albert Y. Zomaya. On Using Pattern Matching Algorithms in MapReduce Applications. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pages 75–80. IEEE, May 2011.

[27] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Mochi: visual log-analysis based tools for debugging hadoop. page 18, June 2009.

[28] Fei Teng, Lei Yu, and Frederic Magoulès. SimMapReduce: A Simulator for Modeling MapReduce Framework. In *2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering*, pages 277–282. IEEE, June 2011.

[29] The Apache Software Fundation. Running TPC-H queries on Hive, 2009.

[30] The Apache Software Fundation. Vaidya Guide, 2011.

[31] The Apache Software Fundation. Rumen: a tool to extract job characterization data form job tracker logs., 2013.

[32] Transaction Processing Performance Counsil. TPC-H Benchmark, 2001.

[33] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of mapreduce setups. In *Proceedings of the 1st ACM workshop on Large-Scale system and application performance - LSAP '09*, page 19, New York, New York, USA, June 2009. ACM Press.

[34] Hailong Yang, Zhongzhi Luan, Wenjun Li, and Depei Qian. MapReduce Workload Modeling with Statistical Approach. *Journal of Grid Computing*, 10(2):279–310, January 2012.

[35] Sara Alspaugh Yanpei Chen, Randy H. Katz, Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Query Processing in Big Data Systems: A Cross Industry Study of MapReduce Workloads. Technical Report 12, University of California, Berkeley, August 2012.