Comparing software stacks for Big Data batch processing

João Manuel Policarpo Moreira

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor(s):  Prof. Helena Isabel de Jesus Galhardas
                Prof. Miguel Filipe Leitão Pardal

Examination Committee
Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Helena Isabel de Jesus Galhardas
Member of the Committee: Prof. José Manuel de Campos Lages Garcia Simão

November 2017
Acknowledgments

I would like to thank my supervisors, Professor Miguel Pardal and Professor Helena Galhardas for providing me help and advising me with my work, the entire USP Lab team for providing the Unicage software and presenting solutions to the problems encountered.

I would also like to express my appreciation to the GSD Cluster team for providing the hardware and software needed to setup the multiple machines used in this work.
Abstract

The recent Big Data trend leads companies to produce large volumes and many varieties of data. At the same time, companies need to access data fast, so they can make the best business decisions. The amount of data available continues to increase as most of the data will start to be produced automatically by devices, and transmitted directly machine-to-machine. All this data needs to be processed to produce valuable information.

Some examples of open-source systems that are used for processing data are Hadoop, Hive, Spark and Flink. However, these systems rely on complex software stacks, which makes processing less efficient. Consequently, these systems cannot run in computers with lower hardware specifications, which are usually placed close to sensors that are spread out around the world. Having processing closer to the data sources is one of the envisioned ways to achieve better performance for data processing.

One approach to solve this problem is to remove many layers of software and attempt to provide the same functionality with a leaner system that does not rely on complex software stacks. This is the value proposition of Unicage, a commercial system based on Unix shell scripting, that promises better performance by having a large set of simple and efficient commands that directly use the operating system mechanisms for process execution and inter-process communication.

The goal of our work was to analyse and evaluate the performance of Unicage when compared to other data processing systems, such as Hadoop and Hive. We propose LeanBench, a benchmark that covers relevant workloads composed by multiple operations, and executes operations on the various processing systems in a comparable way. Multiple tests have been performed using this benchmark, which helped to clarify if the complexity of the software stacks is indeed a significant bottleneck in data processing. The tests have allowed to conclude that all systems have advantages and disadvantages, and the best processing system choice heavily depends on the type of processing task.

**Keywords:** Big Data, Benchmarking, Data Processing, Software Stacks, Apache Hadoop, Apache Hive, Apache Spark, Apache Flink, Unicage Toolset, Unix Shell
Resumo

O progresso da tecnologia tem levado as empresas a produzir grandes volumes de dados em formatos variados, a que se convencionou chamar Big Data. Ao mesmo tempo, para tomar as melhores decisões de negócio, as empresas necessitam de obter informação de forma mais rápida. Muitos dispositivos e sensores produzem dados de forma automática. Estes dados são transmitidos entre dispositivos, o que leva a um aumento de volume de dados disponível. Todos estes dados necessitam de ser processados de forma a produzir informação com valor.

Alguns exemplos de sistemas open-source usados para processar Big Data incluem o Hadoop, Hive, Spark e Flink. No entanto, estes sistemas dependem de pilhas de software complexas, o que diminui a eficiência de processamento. Adicionalmente, torna difícil realizar o processamento em dispositivos com hardware de especificações reduzidas. Dispositivos como estes são muitas vezes usados para recolher dados directamente de sensores. Colocar o processamento de dados mais próximo das fontes de dados é uma das formas possíveis para acelerar o processamento de dados.

Uma das abordagens para resolver este problema consiste em remover camadas de software e produzir um sistema que fornece as mesmas funcionalidades, isto leva a que o sistema seja mais leve e simples, sem dependências em pilhas de software complexas. O Unicage é um sistema comercial baseado em Unix shell scripting que segue esta abordagem, prometendo melhor desempenho para processamento de dados. Este sistema consiste num conjunto de ferramentas individuais que utilizam diretamente os mecanismos de execução e comunicação entre processos do sistema operativo.

O objectivo deste trabalho foi a analisar e avaliar o desempenho do sistema Unicage, quando comparado a outros sistemas de processamento, nomeadamente o Hadoop e o Hive. Propomos e produzimos uma benchmark, a que chamamos LeanBench, que permitiu realizar vários testes de forma a clarificar se a complexidade de pilhas de software é ou não de facto um factor significativo no desempenho de processamento de Big Data. Esta benchmark inclui workloads típicas de processamento, compostas por várias operações, executadas de forma comparável em cada um dos sistemas. Os testes realizados permitiram concluir que todos os sistemas apresentam vantagens e desvantagens, e que a escolha do melhor sistema está dependente do tipo de tarefa de processamento.

Palavras-chave: Dados de grande volume, Comparação de desempenho, Processamento de dados, Pilhas de software, Apache Hadoop, Apache Hive, Apache Spark, Apache Flink, Unicage Toolset, Unix Shell
# Contents

Acknowledgments ......................................................... iii
Abstract ................................................................. v
Resumo ........................................................................ vii
List of Tables ............................................................ xi
List of Figures ............................................................. xiii

1 Introduction .............................................................. 1
  1.1 Objectives .......................................................... 3
  1.2 Main Contributions ............................................... 3
  1.3 Dissertation Outline ............................................. 3

2 Background ............................................................... 5
  2.1 Command Processing ........................................... 5
     2.1.1 Unicage & Unicage Cluster ............................ 5
     2.1.2 Microsoft Dryad ........................................ 7
  2.2 MapReduce .......................................................... 8
     2.2.1 Apache Hadoop ........................................... 8
  2.3 Data Warehousing ................................................ 10
     2.3.1 Apache Hive .............................................. 11
     2.3.2 Apache Pig ............................................... 13
  2.4 Data Stream Processing ......................................... 14
     2.4.1 Micro-Batching and Native Stream Processing . 14
     2.4.2 Apache Spark ........................................... 14
     2.4.3 Apache Flink ............................................ 16
  2.5 Comparison of Big Data Processing Systems ............. 17
  2.6 Benchmarking and Metrics .................................... 19
     2.6.1 Metrics .................................................... 19
     2.6.2 Benchmarking Tools .................................... 19
     2.6.3 Comparison of Benchmarking Tools ................ 22
  2.7 Summary ............................................................ 23
List of Tables

2.1 Sample Unicage toolset commands. ........................................... 6
2.2 Big Data processing systems summary and comparison. .................. 18
2.3 Benchmarking tool comparison. .................................................. 23

3.1 Benchmark operations for each system. ..................................... 26
3.2 Languages supported by the processing systems. ............................ 27
3.3 Regular Expressions used in the Grep Operation. ........................... 29
3.4 Wordcount input file. .............................................................. 31
3.5 Key/Value pairs. ................................................................. 31
3.6 Final output of the operation. ...................................................... 31
3.7 Description of Unicage Querying commands. ................................. 32

4.1 Single Machine Scenario - Experiment Set 1 - Hardware used (Machine #1). 37
4.2 Single Machine Scenario - Experiment Set 1 - Dataset List. .............. 37
4.3 Single Machine Scenario - Experiment Set 1 - Additional Dataset List. 38
4.4 Single Machine Scenario - Experiment Set 2 - Hardware used (Machine #2). 48
4.5 Single Machine Scenario - Experiment Set 2 - Dataset List. .............. 48
4.6 Batch Processing Results - Experiment Set 2 - 40 and 60 GB Datasets. 51
4.7 Unix sort Command. .............................................................. 55
4.8 Unicage msort Command. ....................................................... 55
4.9 Unicage - Average splitting costs (seconds) - 10 Samples. .............. 55

A.1 Maximum margin of error at multiple confidence levels and sample sizes. A.2
List of Figures

1.1 Relations between Information Systems. ................................................. 1
1.2 Software stack comparison between Hadoop and Unicage. ......................... 2

2.1 Unicage Cluster Sample Architecture. ..................................................... 7
2.2 Hadoop Architecture and Modules. .......................................................... 9
2.3 Hadoop HDFS Architecture. ................................................................. 10
2.4 Hive Data Model. ................................................................................. 11
2.5 Hive Architecture. ................................................................................. 12
2.6 Micro-Batching processing vs. Native Streaming. ....................................... 14

3.1 Composition of the Batch Processing module of LeanBench. ....................... 25
3.2 Composition of the Database Processing module of LeanBench. ................. 26

4.1 Batch Processing Operations - Experiment Set 1. ..................................... 38
4.2 Sort Operation - Experiment Set 1. .......................................................... 39
4.3 Average Grep Operations - Experiment Set 1. .......................................... 39
4.4 Wordcount Operation - Experiment Set 1. .............................................. 40
4.5 Unicage vs. Hadoop - Experiment Set 1. ................................................ 40
4.6 Batch Processing Operations - Experiment Set 1 - Additional Tests. ........... 41
4.7 Sort Operation - Experiment Set 1 - Additional Tests. .............................. 41
4.8 Average Grep Operations - Experiment Set 1 - Additional Tests. ............... 42
4.9 Wordcount Operation - Experiment Set 1 - Additional Tests. .................... 42
4.10 Unicage vs. Hadoop - Experiment Set 1 - Additional Tests. .................... 43
4.11 Data Querying Operations - Experiment Set 1 (Logarithmic Scale). .......... 43
4.12 Selection Query - Experiment Set 1. ..................................................... 44
4.13 Aggregation Query - Experiment Set 1. ................................................ 44
4.14 Join Query - Experiment Set 1. ............................................................. 45
4.15 Unicage vs. Hive - Experiment Set 1. .................................................... 45
4.16 Data Querying Operations - Experiment Set 1 - Additional Tests. .......... 46
4.17 Selection Query - Experiment Set 1 - Additional Tests. ........................ 46
4.18 Aggregation Query - Experiment Set 1 - Additional Tests. ..................... 47
4.19 Join Query - Experiment Set 1 - Additional Tests. ................................ 47
4.20 Unicage vs. Hive - Experiment Set 1 - Additional Tests. ............................ 48
4.21 Batch Processing Operations - Experiment Set 2. ................................. 49
4.22 Sort Operation - Experiment Set 2. .................................................. 49
4.23 Average Grep Operations - Experiment Set 2. ...................................... 50
4.24 Wordcount Operation - Experiment Set 2. ........................................ 50
4.25 Unicage vs. Hadoop - Experiment Set 2. .......................................... 51
4.26 Data Querying Operations - Experiment Set 2. .................................. 52
4.27 Selection Query - Experiment Set 2. .............................................. 52
4.28 Aggregation Query - Experiment Set 2. ........................................... 53
4.29 Join Query - Experiment Set 2. ...................................................... 53
4.30 Unicage vs. Hive - Experiment Set 2. ............................................. 54

5.1 Best processing system according to different processing tasks and specific requirements. 58
Chapter 1

Introduction

Traditionally, data in companies is processed by different kinds of information systems [1], such as Transaction Processing Systems (TPS) and Intelligent Support Systems (ISS). Management Information Systems (MIS) handle data provided by these systems and serves the purpose of providing information useful for monitoring and controlling a business. Figure 1.1 presents the relations between the various information systems.

Figure 1.1: Relations between Information Systems.

Data production continues to increase with the progression of technology. Companies produce large amounts of data, not only in volume but also in a wide variety of types. At the same time, in order to make the best business decisions in a timely manner, companies need to increase the velocity at which data is accessed and processed. Nowadays, these developments are designated by the term Big Data¹.

In addition to companies, the Internet of Things will also further increase the amount of data produced [2]. This is due to the fact that most of the data will start to be gathered automatically by devices and their sensors, and at the same time, transmitted directly from machine-to-machine in the network. These sensors are usually connected to small computers, such as a Raspberry Pi or an Intel Galileo.

¹Big Data: The term Big Data is used for large volumes of data in a wide variety of formats that require to be processed as quickly as possible, something that is not supported by traditional data processing tools.
Batch processing plays an important role by taking data periodically from transactional systems and processing it to be used by information management systems. Data querying facilities provide information about business transactions. Batch processing features are provided by systems like Apache Hadoop\(^2\), and querying facilities are provided by systems like Apache Hive\(^3\).

ISS are a type of systems that include a broad set of more specific support systems, such as *Decision Support Systems* (DSS). These systems rely on sophisticated techniques such as machine learning and graph analytics to provide insight about data. Apache Spark\(^4\) and Apache Flink\(^5\) are examples of systems that support these techniques.

However, all of these systems rely on external complex software stacks. As a consequence, it is very difficult to run these applications directly on smaller devices such as the ones mentioned previously. One possible solution to this problem would be to remove most of these dependencies and produce tools that provide the same functionality but rely on simpler software stacks.

One example of a tool like this is the Unicage system\(^6\). Unicage removes dependencies on complex software by using individual Unix shell commands that provide a single processing function. These commands are configured in a shell script which run as Unix processes, use system calls directly and use pipes for communicating and transmitting information. Pipes are the basis of the Unix inter-process communication model \cite{3} and enable to connect the standard output of one process to the standard input of another.

As observed in Figure 1.2, the Unicage stack is significantly smaller and simpler when compared to the stack of Hadoop.

![Hadoop Software Stack](image)

<table>
<thead>
<tr>
<th>Hive</th>
<th>MapReduce</th>
<th>YARN</th>
<th>HDFS</th>
<th>JVM</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database management system</td>
<td>Distributed processing framework</td>
<td>Scheduler and resource manager</td>
<td>Distributed file storage system</td>
<td>Java Virtual Machine</td>
<td>Operating System</td>
</tr>
</tbody>
</table>

![Unicage Software Stack](image)

<table>
<thead>
<tr>
<th>Unix Shell Script</th>
<th>Commands</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>User specified script</td>
<td>Unicage toolset commands</td>
<td>Operating System</td>
</tr>
</tbody>
</table>

Figure 1.2: Software stack comparison between Hadoop and Unicage.

\(^2\)http://hadoop.apache.org/
\(^3\)https://hive.apache.org/
\(^4\)http://spark.apache.org/
\(^5\)https://flink.apache.org/
\(^6\)http://en.usp-lab.com/
1.1 Objectives

There are many different types of systems with different processing goals, and due to the challenge of data sizes increasing in the near future, there is a need to determine what are the most efficient tools to perform the desired processing for each intended goal.

Unicage, a processing tool that directly uses the underlying operating system mechanisms such as Unix Pipes will be compared with Hadoop and Hive, tools that rely on additional abstraction layers such as virtual machines and middleware layers.

The main objective of this work is to assess how Unicage, a system that uses the operating system process execution and inter-process communication mechanisms directly, compares to Big Data processing tools that rely on complex software stacks.

1.2 Main Contributions

This work presents three main contributions: i) LeanBench, a benchmarking tool that allows to perform a comparison between Unicage and Hadoop, and Unicage and Hive.

ii) Performance measurements between the two kinds of tools. The main evaluation consisted on running and benchmarking the tools on a single machine scenario. The single machine scenario experiments carried out in this work have served the purpose to identify the limitations of a single machine, but also to identify if each system supports the capabilities necessary to process Big Data.

iii) Guidelines that enable users to choose the best processing system according to the type of processing tasks required.

1.3 Dissertation Outline

This document is organised into three different parts. The first part, composed by Chapter 2, analyses data processing systems and benchmarking tools.

The second part is composed by Chapters 3 and 4, where Chapter 3 describes the LeanBench benchmark for analysing the performance of the various processing systems and Chapter 4 describes the experimental validation and methodology that was used for the benchmark.

In the third and last part, composed by Chapter 5, the conclusions that we obtained by the benchmark results are presented and in addition, we present a discussion for future work possibilities.
Chapter 2

Background

This Chapter describes the most relevant paradigms used for processing Big Data as well as some of the most popular tools that support these paradigms. Section 2.1 presents Command Processing, a simple data processing paradigm, used by tools such as Unicage and Dryad. Section 2.2 explains MapReduce, a programming model used for handling data processing in a distributed way. Additionally, an analysis of Apache Hadoop, a framework that implements the MapReduce programming model is also presented. Section 2.3 describes how database functionalities are provided for Big Data, making an analysis of the Apache Hive framework and the Apache Pig tool. In Section 2.4, stream processing and micro-batching paradigms are explained and compared, in particular, Apache Spark and Apache Flink are analysed and compared. Section 2.5 presents a brief comparison of all the Big Data processing systems discussed and Section 2.6 presents and makes a comparison between multiple benchmarking tools, as well as describing metrics used when benchmarking Big Data processing systems.

At the end, Section 2.7 summarises the multiple systems and benchmarking tools discussed.

2.1 Command Processing

Command Processing is a data processing paradigm where the processing is accomplished by multiple commands, composed to achieve the processing goal. Each individual command usually provides a single function. In order to process a dataset, multiple commands are arranged in a script which is constructed according to the type of processing required.

2.1.1 Unicage & Unicage Cluster

Unicage is a toolset that provides Unix shell commands which enable users to compose data processing systems. In order to make maintenance easy, Unicage promotes a distributed development approach, where data, programs and hardware are all separated from each other [4].

The Unicage toolset is composed by a large number of individual commands\(^1\) which provide individual functionalities needed to process data. These include string formatting functions, system functions

\(^1\)http://en.usp-lab.com/command.html
for input/output of files, common operations used in relational queries (such as joins, selections), mathematical and statistical functions. Each individual command is designed to be simple to use and only provide a single function. Some examples of commands and their functionalities are presented below in Table 2.1.

Table 2.1: Sample Unicage toolset commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count</code></td>
<td>Counts entries with same key</td>
</tr>
<tr>
<td><code>cent</code></td>
<td>Counts columns</td>
</tr>
<tr>
<td><code>mdate</code></td>
<td>Performs date calculations</td>
</tr>
<tr>
<td><code>msort</code></td>
<td>Perform sort in-memory</td>
</tr>
<tr>
<td><code>transpose</code></td>
<td>Transposes rows and columns</td>
</tr>
<tr>
<td><code>tocsv</code></td>
<td>Converts a file delimited by spaces to a file with entries separated by commas</td>
</tr>
</tbody>
</table>

The Unicage development method consists on writing a script using a standard Unix shell programming language. This script uses the default Unix commands, such as `cp`, `find` and `sort` in conjunction with commands of the Unicage toolset.

A Unicage script can be represented with a processes and pipes abstraction, where a single task process receives an input and produces an output which is piped to another process. The first command listed in the script runs on the original input data and produces a result, which is then transmitted through an Unix Pipe to the next listed command. The various commands in the script communicate in this way until the last command. Usually the end result is piped to a command such as `tocsv`, that saves the data in a specific user-readable format. The toolset commands can be combined in a script to perform many different types of data processing, according to the specific requirements of each system.

**Unicage performance optimisation**

Each command of the Unicage toolset is written directly in the C programming language in order to take advantage of the input/output buffers and memory manipulation techniques of the language. In addition to this, Unicage programs use the Unix shell, which internally uses kernel functions directly, avoiding other middleware software or any other unnecessary processing that could possibly slow down the processing done by the command.

Other optimisations include sorting the data in advance to avoid an unnecessary sort during the processing of data. For sorting the entries, Unicage also provides and uses a custom sorting command based on the Merge Sort algorithm [5].

**Unicage and Big Data Processing**

Unicage Cluster BOA is a Big Data oriented architecture entirely dedicated to the processing of Big Data applying the Unicage development method, using Unix shell scripting and Unicage toolset commands.
A Unicage Cluster consists of a main Master server which runs the BOA cluster software and the user-defined main Unix shell script. The Master server communicates with the Slave servers through a high-speed network. The actual processing is distributed and performed by the multiple Slave servers. This is where Unicage becomes comparable with Hadoop and other systems that will be discussed later in this Chapter. Figure 2.1 presents a Unicage Cluster sample architecture configuration.

The Unicage Cluster can run under any Unix-based operating system. Similarly to Apache Hadoop, it is possible to scale out the processing by adding more commodity machines, which increases the number of processing nodes.

When it comes to fault tolerance, Unicage Cluster does not include any fault tolerance mechanisms by default. All fault tolerance capabilities and mechanisms have to be directly implemented in the user-defined processing scripts.

### 2.1.2 Microsoft Dryad

Microsoft Dryad [6] is a discontinued distributed execution engine system for processing data. Dryad uses multiple computational vertices that communicate to form a datalflow graph. A datalflow graph of Dryad is considered as a job, where each vertex of the graph is a program, and the edges of the graph are data.

Dryad provides a C++ interface for developers to write their own programs. This interface is the base for each vertex (a program) of the datalflow graph, which will be executed by the Dryad runtime library. The runtime library is responsible for setting up the distributed computation, assigning each vertex to the available resources. Dryad was discontinued as Microsoft shifted focus to have a distributed processing data system based in Apache Hadoop for the Windows Server and Windows Azure platforms.

---

2.2 MapReduce

MapReduce is a programming model designed to process and manage large datasets. The MapReduce programming model idea was firstly introduced at Google [7]. Developers observed that most of their data processing was composed by two types of operations, one for inputting data records and computing a set of intermediate key/value pairs, and a second operation to combine all of the data generated by the first operation. MapReduce applies this technique and uses two distinct functions: the map function processes a pair of key/values of data and generates an additional intermediate pair of key/values of data; the reduce function merges all the intermediate values generated by the map function. Both of these functions are written by the developer according to the type of the data and also to the type of processing that the data needs. Using this programming model, solutions to many real world problems, such as distributed grep and URL access frequency count, can be easily solved and implemented.

Programs written using this model are automatically parallelized as map and reduce tasks which can be assigned to multiple machines. Moreover, by adding more commodity machines, the cluster is expanded, taking the scale-out\(^3\) approach to increase the processing power. Each machine in the cluster is assigned as a worker for map and reduce tasks, executing the processing operations specified by the user-defined map and reduce functions.

In addition, a Master server monitors the machines assigned as workers and a periodic check is performed to ensure that every single machine is working. If a worker machine is detected to be faulty, the task can be reassigned to another available machine, giving MapReduce the ability to re-execute operations if needed, as certain machines in the cluster can fail in the middle of processing. This behaviour allows MapReduce to have basic fault tolerance capabilities that are compatible with the programming model.

2.2.1 Apache Hadoop

Apache Hadoop is an open-source framework designed for processing large datasets across clusters of computers. Hadoop uses the MapReduce programming model, which enables it to process the data in a distributed computing approach.

The framework consists on three basic modules:

- **Hadoop Yet Another Resource Negotiator (YARN)**: A job scheduling framework and cluster resource management module;

- **Hadoop MapReduce**: A YARN-based system for parallel processing of large datasets, using an implementation of the MapReduce programming model;

- **Hadoop Distributed File System (HDFS)**: A distributed file system module that stores data on the cluster machines.

\(^{3}\)Scale-out vs Scale-up: Scale-out refers to scaling a system horizontally by adding more nodes to increase processing power. Scale-up refers to scaling a system vertically, by improving the resources of a single machine to increase the processing power.
Figure 2.2 presents the architecture and modules of Hadoop.

![Hadoop Architecture and Modules](image)

Hadoop allows to store, manage and process large datasets of both structured and unstructured data\(^4\) in a reliable way because the data processing is based on a cluster of computers. Unlike traditional relational database management systems, Hadoop can process both semi-structured and unstructured data because the processing is not performed automatically but instead programmed and specified by the user.

**Hadoop YARN**

Hadoop started as a simple open-source implementation of MapReduce and it mainly focused on having a platform to process and store large volumes of data. Over time, it became the main tool used for this purpose. However, as its user base grew, it led to wrong uses, mainly due to users extending the MapReduce programming model with resource management and scheduling features, capabilities that MapReduce was not designed to have from the first place.

To overcome this problem, there was a community effort to produce a resource management module, called *Yet Another Resource Negotiator* (YARN). This job scheduling and cluster resource management module was designed and developed with efficiency in mind. It provided scheduling functions, and an infrastructure for resource management which included extended fault tolerance capabilities.

YARN was integrated into Hadoop, therefore Hadoop became split in two main logical blocks: \(i\) YARN the resource management block, providing scheduling functions for the various jobs, and \(ii\) MapReduce, the processing block running on top of YARN, providing the data processing capabilities [8]. With the inclusion of YARN, resource management functions were effectively separated from the programming model. In addition, this integration allows to easily scale Hadoop and distribute work along multiple processing nodes.

---

\(^4\)**Structured vs Unstructured Data**: Structured data refers to data that is organised according to a data model called a schema. Unstructured data refers to data that is not fully organised or lacks a data schema.
Hadoop Distributed File System (HDFS)

HDFS [9] is the distributed file system of Hadoop. It allows for distributed storage in a cluster of many commodity machines, making it easy to scale-out, enabling the storage of large volumes of data across multiple machines. HDFS consists of a single Master server and multiple Slave servers. The Master server is called a NameNode, while the Slave servers are called DataNodes. There is usually at least one DataNode per machine in the cluster. The NameNode is responsible for answering file system requests, such as creating, opening and closing files, managing directories and renaming operations. The DataNodes are responsible for storing the files in various blocks. They also provide read and write operations, giving access to the data. When a user requests some data, the location of the data is given by the NameNode.

After receiving this location, the user can directly access the data. The NameNode and DataNodes work in this way to ensure that the data never has to flow through the NameNode, as that would defeat the purpose of having a distributed file system architecture without having a concentration of workloads in a small set of machines.

Figure 2.3 presents a sample architecture of HDFS.

![Hadoop HDFS Architecture](image)

HDFS was designed to be fault-tolerant to hardware failure of machines in the cluster. If a machine fails, the HDFS system is able to quickly detect the fault and automatically recover from the failure. This is possible due to the data replication mechanisms that HDFS uses. HDFS stores files in a sequence of blocks. These blocks are then replicated and distributed in the various DataNodes of the cluster.

There are many types of faults that can occur to data stored in HDFS. A DataNode could simply fail, losing access to the entire data, but there is also a chance that blocks of files can get corrupted. To maintain data integrity, HDFS verifies the checksum of blocks to ensure that the data is not corrupted.

2.3 Data Warehousing

The MapReduce programming model enables tools such as Hadoop to process large datasets distributed through several machines in a cluster, containing both structured and unstructured data in a reliable and fault tolerant way. However, there are many data warehousing functionalities that the MapReduce programming model is not able to provide.
One of the biggest limitations is that tools using the MapReduce programming model cannot directly execute queries over the data using a querying language, such as SQL. In order to query the data using MapReduce, users have to manually write their own `map` and `reduce` functions in a lower level language, which is time consuming and difficult to maintain when compared to SQL-like querying languages. In addition to this, when the user defines the `map` and `reduce` functions for a certain dataset, the functions are usually optimised according to the characteristics of that dataset in particular. When a different dataset needs to be processed, most of the code from these functions cannot be reutilised, and thus the code needs to be rewritten. Apache Hive [10], aims at offering features to overcome these problems while providing the benefits of the MapReduce programming model, such as the distributed processing and fault tolerance capabilities.

2.3.1 Apache Hive

Apache Hive is a Data Warehousing system built on top of the Apache Hadoop framework. Hive enables users to query large datasets that reside in a distributed storage form, such as the Hadoop HDFS system, without relying on defining complex low level `map` and `reduce` functions.

Hive provides *HiveQL*, an SQL-like interface that enables the user to create queries in a language similar to SQL. Internally, the queries written in *HiveQL* are compiled and directly translated to MapReduce jobs, which are executed by the Hadoop framework. Additionally, Hive supports the compilation of *HiveQL* queries to other platforms, such as Apache Spark.

Hive Data Model

Hive manages and organises the stored data in three different containers, as shown in Figure 2.4.

- **Tables**: Similar to the tables in relational database systems, allowing filter, projection, join and union operations. These tables are stored in HDFS;
• **Partitions**: The tables may be composed of one or more partitions. Partition keys allow to identify the location of the data in the file system;

• **Buckets**: Inside each partition the data can be divided further into buckets. Each bucket is stored as a file in the partition.

**Architecture and Components**

Apache Hive has two main logical blocks: Hive and Hadoop. The Hive block is mainly responsible for taking the user input and turning it into something that is compatible with the Hadoop framework. This block has the following components:

• **UI**: User interface component for submitting queries;

• **Driver**: Driver component to handle sessions. Provides Java and C++ APIs (JDBC and ODBC interfaces) for fetching data and executing queries;

• **Compiler**: Component that creates execution plans by parsing *HiveQL* queries;

• **Metastore**: Component used for storing all the metadata on the various tables stored in a database, such as columns and data types. This component also provides serializers and deserializers to the Execution Engine, which are needed for read/write operations from/to the Hadoop Distributed File System;

• **Execution Engine**: Component used for executing the execution plan previously created by the compiler.

Figure 2.5 presents the architecture of Apache Hive.

![Figure 2.5: Hive Architecture.](image)

The *Metastore* component provides two additional useful features: data abstraction and data discovery. Without data abstraction, in addition to the query itself, users would need to provide more
information, such as the format of the data they are querying. With data abstraction, all the needed information is stored in the Metastore when a table is created. When a table is referenced this information is simply provided by the Metastore. The data discovery feature enables users to browse through the data in a database.

The Compiler component produces an execution plan, which consists of several stages. Each stage is a map or a reduce job. This execution plan is sent to the Driver, and then forwarded to the Execution Engine. The Execution Engine communicates directly with the Hadoop logical block. This block is responsible for executing the map and reduce tasks that it receives. After the processing is performed, the results are sent back to the Execution Engine on the Hive logical block. Finally, the results are then forwarded to the Driver and delivered to the user.

Hive Execution Engine Performance and Optimisations

Hive is extensively used when large volumes of data are handled, for example, in social networks and other online services. As the number of users increases the volumes of data become larger, requiring additional time to process the data. In order to provide faster processing times, the storage and query execution functionalities need to be improved.

The Execution Engine component in Hive Architecture (as shown in Figure 2.5) relies on using lazy deserialization to reduce the amount of data being deserialized. However, this mechanism introduces virtualised calls to the deserialization functions, which can slow down the execution process. Many possible optimisations for the Execution Engine of Hive have been suggested, these optimisations take into account the modern architectures and characteristics of CPUs to improve performance. Another suggested optimisation is to update the query planning component and extend the pre-processing analysis in order to attempt to reduce unnecessary operations in query plans of complex queries.

A new file format called Optimised Record Columnar File (ORC) that focuses on improving the efficiency of the storage functionalities of Hive has also been suggested. This new file format relies on compression and indexing techniques to improve data access speed. All of the optimisations suggested have been properly tested and benchmarked, showing a significant increase in performance.

2.3.2 Apache Pig

Apache Pig is a tool that provides a high-level platform for creating programs that run on the Apache Hadoop system. Pig provides a SQL-like language called PigLatin. Programs written in the PigLatin language are compiled to Map/Reduce tasks, a process similar to HiveQL of Apache Hive.

Lazy Deserialization: A method used by Hive which reduces deserialization by only deserializing objects when they are accessed.

https://pig.apache.org/
2.4 Data Stream Processing

The previous Sections have described how data can be processed in large batches. However, certain applications require processing an incoming stream of data and produce results as fast as possible. This type of data processing is called *data stream processing* and can be achieved in two main ways: *i*) *Micro-Batching* and *ii*) *Native stream processing*.

2.4.1 Micro-Batching and Native Stream Processing

*Micro-batching* is a technique for data stream processing that treats a data stream as a sequence of small batches of data. These small batches are created from the incoming data stream and contain a collection of events that were received during a small period of time, called the batch period. In *Native stream processing* data is not bundled into batches and is instead processed as it arrives in the stream, producing an immediate result.

Figure 2.6 represents the differences between micro-batching and native streaming.

![Diagram of Micro-Batching processing vs. Native Streaming](image)

Apache Spark and Apache Flink both support stream processing modes. Apache Spark uses the *Micro-batching* technique for stream processing while Apache Flink supports stream processing natively. Both tools provide libraries that support machine learning and graph analysis, i.e. features that are necessary for processing data streams using more sophisticated functions.

2.4.2 Apache Spark

Apache Spark is an open-source framework designed to provide a fast processing engine for large scale data while providing distributed computing and fault tolerance capabilities. When it comes to data processing, Spark emulates the data stream processing model by using the *Micro-batching* technique.
Spark was designed and developed to overcome the limitations of the MapReduce programming model. MapReduce forces users to implement their map and reduce functions and also causes programs to follow a linear data flow structure in distributed programs, i.e. the map functions read data from physical drives and map this data, while reduce functions reduce the results of the map and then write the output to disk.

Spark introduces the ability to iterate over the data, which was not possible using the MapReduce programming model. Having iterative capabilities allows for new types of processing analysis such as machine learning and graph processing, features made available by the MLlib and GraphX libraries of Spark.

Additionally, the main feature of Spark is the introduction of an abstraction called the Resilient Distributed Dataset which is explained in what follows.

Resilient Distributed Dataset

Spark uses a data structure abstraction called Resilient Distributed Dataset (RDD), which allows distributed programs to use a restricted form of DSMs. RDDs are immutable partitioned collections that can improve the way that iterative algorithms are implemented, for example when accessing data multiple times in a loop. In comparison to a MapReduce implementation, the RDD abstraction can improve processing speeds.

When in comparison to regular DSMs, RDDs also present many benefits [13]. For example RDDs can only be created when coarse-grained transformations occur, such as the transformations done by the map and reduce functions. This restricts RDDs to perform bulk writes but allows for a more efficient fault tolerance when compared to DSMs. The immutable nature of RDDs allows to run backup tasks on slower nodes. This task would be difficult to implement using DSMs as there would be a possibility that two different nodes could access the same memory location and modify the same data simultaneously.

The performance of RDDs is reduced significantly when the data size is too large to fit in memory. However, parts of the data that cannot fit in memory can be stored on disk while still providing almost identical performance to other distributed processing systems such as MapReduce.

MLlib

Spark includes an open-source distributed machine learning library that facilitates iterative machine learning tasks, called MLlib. This library provides distributed implementations of various machine learning algorithms, such as Naive Bayes, Decision Trees, K-means clustering, and more. The library also includes many lower level functions for linear algebra and statistical analysis which are optimised for distributed computing. MLlib includes many optimisations to allow for better performance, some of these include reduced JVM garbage collection overhead and usage of efficient C++ libraries for linear algebra operations at the worker level.

---

*DSM*: Distributed shared memory is a resource management technique used in Distributed Systems that allows to map physically separated memory address spaces in a single logical shared memory address space.

*Immutable*: Refers to objects that cannot be changed or modified once created.

*JVM*: Java Virtual Machine
MLlib also provides a Pipeline API and direct integration with Spark. The API allows users to simplify the development and tuning of multi-stage learning pipelines as it includes a set of functions that allows users to swap learning functions easily. Integration with Spark allows MLlib to use and benefit from the other components that Spark includes, such as the execution engine to transform data, and GraphX to support the processing of large-scale graphs [14].

GraphX

For graph processing, Spark uses GraphX, an open-source API for graph processing which allows to process graphs in a distributed form. This API takes advantage of Spark RDDs to create a new form of graph called the Resilient Distributed Graph abstraction (RDGs).

This new abstraction allows simplified file loading, construction, transformation and computations of graphs while improving the performance compared to other types of graphs. In addition to this, it also facilitates the implementation of graph-parallel abstractions, such as PowerGrap and Pregel [15].

GraphX includes many graph algorithms by default that are useful for data processing in graphs\(^\text{10}\), including PageRank, Label propagation and Strongly Connected Components (SCCs).

2.4.3 Apache Flink

Apache Flink [16] is an open-source framework designed for distributed Big Data computation and analysis through native streaming processing. The main goal of Apache Flink is to fill the gap that exists between MapReduce frameworks such as Hadoop and Hive, and RDD micro-batch oriented systems like Spark.

The main difference between Spark and Flink is that the latter natively supports in-memory stream processing, while Spark only emulates in-memory streaming processing through the RDD abstraction with micro-batches.

Flink can also work in a batch processing mode just like Spark, in addition to the streaming processing mode that it natively supports. Flink also supports integration with Hadoop modules, such as YARN and HDFS. For the processing features, Flink has available two different data processing APIs\(^\text{11}\): DataStream API which is responsible for handling unbounded data streams and DataSet API which handles static batches of data. These two APIs are supported by runtime engine of Flink. The two APIs have an additional Table API that provides a SQL-like language that enables users to query the data directly and interactively. The stream processing capabilities of Flink allow users to iterate over data natively, which Spark also supports, although only in micro-batching mode.

Flink programs can be written in programming languages like Java and Python. Internally, these programs are compiled by the runtime engine, which converts the programs to a Directed Acyclic Graph (DAG) that contains operators and data streams. This is called the Dataflow Graph.

Apache Flink provides fault tolerance capabilities, using a mechanism called Asynchronous Barrier Snapshotting (ABS) [17]. This mechanism consists of taking snapshots of the Dataflow Graph at fixed

\(^{10}\)http://spark.apache.org/graphx/

\(^{11}\)https://ci.apache.org/projects/flink/flink-docs-release-1.1/#stack
intervals. Each snapshot contains all the information contained in the graph at the moment the snapshot was taken, and thus, the snapshots can be used to recover from failures.

**FlinkCEP**

One of the main challenges in real-time processing is the detection of patterns in data streams. Flink handles this problem with the FlinkCEP (Complex Event Processing) library.

This library enables to detect complex patterns in a stream of incoming data, thus allowing users to extract the information they need. For detecting patterns in data, FlinkCEP provides a pattern API, where users can quickly define conditions to match or filter the patterns they are interested in.

These patterns are recognised in a series of multiple states. To go from one state to another the data must match the exact conditions that the user has specified. In addition to this, the native streaming support of Apache Flink permits the FlinkCEP library to recognise the data as it is being streamed.

**FlinkML**

FlinkML is the machine learning library of Flink. Just like MLlib of Spark, it supports many different algorithms, some of these include Multiple linear regression, K-nearest neighbours join, Min/Max scaler, Alternating least squares and Distance metrics.

**Gelly**

Gelly is the Graph API of Flink. The API contains a vast set of tools to simplify the development of graph analysis applications using Flink, similar to the GraphX library of Spark. It provides many algorithms and tools to load, construct, transform and compute graphs. Additionally, users can define other graph algorithms by extending the GraphAlgorithm interface of the Gelly API.

### 2.5 Comparison of Big Data Processing Systems

This Section makes a brief review and comparison of the relevant Big Data processing systems that have been discussed.

- **Unicage** is a toolset used for building information systems with Big Data processing capabilities without relying on complex middleware;

- **Apache Hadoop** is a batch processing framework for processing Big data using the MapReduce programming model;

- **Apache Hive** extends Hadoop by providing a querying language and Data Warehouse functionalities;

- **Apache Spark** and **Apache Flink** are systems that provide streaming processing capabilities, appropriate for tasks that require real-time responses. Both systems include support for machine learning and graph processing.
Table 2.2 summarises these processing systems and their respective major features.

<table>
<thead>
<tr>
<th></th>
<th>Unicage</th>
<th>Apache Hadoop</th>
<th>Apache Hive</th>
<th>Apache Spark</th>
<th>Apache Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Paradigm</strong></td>
<td>Batch processing through individual commands</td>
<td>Batch processing through MapReduce</td>
<td>Data Warehouse with HiveQL querying</td>
<td>Micro-Batch processing in memory using RDD abstraction</td>
<td>Native stream processing in memory</td>
</tr>
<tr>
<td><strong>Resource management</strong></td>
<td>Manually configured in script</td>
<td>YARN</td>
<td>YARN</td>
<td>YARN</td>
<td>YARN</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>Varies *</td>
<td>Varies *</td>
<td>Varies *</td>
<td>Seconds</td>
<td>Milliseconds</td>
</tr>
<tr>
<td><strong>Language written in</strong></td>
<td>C</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td><strong>Languages supported</strong></td>
<td>Unix Shell, Script, Java, HiveQL</td>
<td>Scala, Java, Python</td>
<td>Scala, Java, Python</td>
<td>Scala, Java, Python</td>
<td>Scala, Java, Python</td>
</tr>
<tr>
<td><strong>Distinguishing features</strong></td>
<td>Direct use of Unix kernel functions for higher performance</td>
<td>Stable resource management and file system libraries (YARN and HDFS)</td>
<td>The HiveQL queries are compiled and optimised to MapReduce jobs</td>
<td>RDD abstraction ease of use, support for machine learning and graphs</td>
<td>Native streaming processing in-memory, support for machine learning and graphs</td>
</tr>
</tbody>
</table>

* Latency in these systems varies depending on the volume of data being processed. Unlike Spark and Flink, these systems only produce results after the entire volume of data has been processed.
2.6 Benchmarking and Metrics

This Section describes benchmarks and relevant metrics used when evaluating the performance of data processing systems. Benchmarks provide the means to measure and compare the performance of data processing systems. Multiple benchmarking tools will be presented in this Section.

2.6.1 Metrics

The metrics used in data processing benchmarking are mostly based on individual operation execution times. However, these metrics disregard other factors that are important when dealing with Big Data.

The time to load or import data may be a significant factor to take into account as an additional metric, depending on the volume and types of data being loaded. Additionally, in systems like Hive, HiveQL queries must be compiled to MapReduce jobs. Some queries may be more complex than others, requiring more time to be fully processed before being executed. It can be an additional factor to consider as a metric.

The term Big Data also covers many different types of data, and as such, queries for processing image data can be several times more complex when compared to queries for processing text data. The type of data being processed can be a significant factor to take into account as a metric when benchmarking data processing systems.

2.6.2 Benchmarking Tools

There are many different benchmarking libraries and tools available to evaluate the performance of Big Data processing systems. However the benchmarking method used needs to be picked accordingly to the type of data analysis tool that is being used. This is due to the different methods and paradigms that Big Data processing systems use. For instance, Hadoop and Hive are batch processing systems that mainly rely on HDFS for storage and MapReduce for data processing. Other processing systems, like Spark and Flink, are based on processing data streams. Even though there are many different benchmarking tools available, proposals for having an industry standard benchmark for Big Data processing systems exist.

BigBench

BigBench [18] is a proposal for a Big Data benchmarking tool. It is an attempt to create an industry standard benchmark for Big Data. BigBench was designed with MapReduce systems in mind, such as Hadoop and Hive. The BigBench proposal suggests having a workload that uses a set of many different queries in order to evaluate the performance of the system, attempting to cover the three Big Data characteristics: i) Volume of data, ii) Variety of data and iii) Velocity of data processing.

For evaluating the Volume of data, BigBench proposes having a Big Data generator, which can scale and be configured accordingly to a scaling factor. When it comes to the variety of data, the proposal suggests that the Big Data generator should be able provide both structured and unstructured data.
Finally, for evaluating the Velocity, it proposes having a continuous input of data into the storage system. The continuous input of data varies in volume, depending on the queries being run and also on the defined scale factor.

The workload proposed contains a vast set of operations designed accordingly to the various different types of processing. This proposed workload in BigBench allows to have many different metrics according to what is being evaluated, but it also makes possible to create custom composite metrics that evaluate more than two relevant aspects at once.

**Hadoop Benchmarking Tools**

The Apache Hadoop distribution not only includes the tool itself but also tools designed to test and benchmark various aspects of the Hadoop framework, such as the storage component (HDFS) and the processing component (MapReduce). These tools can be used for evaluating the execution of tasks in Apache Hive as well, due to the fact that Hive mainly relies on the Hadoop MapReduce and HDFS systems for the processing and storage capabilities respectively. The Apache Hadoop distribution\(^{12}\) includes the following testing and benchmarking tools:

- **TestDFSIO** - A general distributed file system read and write test for HDFS, to test the scalability of the distributed file storage system. It can also be used to detect bottlenecks in the cluster network by identifying slower nodes;

- **NNbench** - An additional testing tool for stress testing the name nodes of HDFS, generally used for testing the cluster hardware and configuration;

- **TeraSort** [19] - A benchmark tool to evaluate the performance of batch systems. This tool is composed by the following components:
  - **TeraGen** - A data generation tool for the test, able to produce large volumes (1TB+) of data;
  - **TeraSort** - A program that sorts the data generated, runs as a MapReduce job;
  - **TeraValidate** - A program that validates the sorted data by TeraSort, runs as a MapReduce job.

- **MRBench** [20] - A tool designed to benchmark systems that use MapReduce, such as Hadoop. While other tools such as TeraSort focus on evaluating the performance by simply sorting large amounts of data, MRBench focuses on evaluating the performance of Map and Reduce tasks by executing highly complex processing on large volumes of relational data. MRBench also allows users to configure many parameters of the benchmark, such as data size, and also the number of Map and Reduce tasks being executed.

In addition to the tools and tests that the Apache Hadoop distribution includes, there are also other tools which have been specifically designed to benchmark and extensively test systems that handle and process Big Data.

These standalone benchmarking tools aim at supporting many different Big Data processing systems while covering many extensive workloads. The standalone benchmarking applications usually support many Big Data processing systems in order to be able to apply the same workloads uniformly. This guarantees that all the workloads are applied equally to all the processing systems being tested. Without this, it would not be possible to evaluate the results rigorously or to extract any relevant information regarding the performance of processing systems when compared to each other.

Some of the most used standalone tools are AMP Benchmark, BigDataBench and HiBench.

**AMP Big Data Benchmark**

The U.C. Berkeley AMP Big Data Benchmark\(^{13}\) is a standalone open-source tool for evaluating the performance of Data Warehousing systems based on MapReduce like Apache Hive. In AMP Benchmark, the main metric used for evaluation is the query execution time. When it comes to datasets, the AMP Benchmark includes support for both structured and unstructured datasets.

The AMP Benchmark workload is based on an existing proposed workload [21]. Some operations of the workload include simple scans, aggregations, joins and also other more complex user-defined functions. In total there are four different types of queries, the first two are regular SQL queries. These queries have three variants, varying the size of the datasets in order to evaluate how well the systems being tested handle scaling. The third query consists of a join query with varying sizes of data. The last query is a bulk user defined function query which calculates a simplified version of PageRank.

**BigDataBench**

BigDataBench [22] is an open-source implementation and extension of the BigBench proposal, including additional different types of data sets with more extensive workloads. It also supports multiple Big Data processing systems.

BigDataBench uses six different datasets. The data in these sets can be split in three different types: 

- **i)** unstructured,
- **ii)** semi-structured and
- **iii)** structured. The six different datasets are representative of real world applications, such as offline analytics, real-time analytics and online services. BigDataBench uses real world datasets instead of using randomly generated data. However, to allow scalability of dataset sizes, BigDataBench introduced the *Big Data Generator Suite* (BDGS). This tool is capable of producing very large sets of synthetic data based on real datasets while keeping the three Big Data characteristics: Volume, Variety and Velocity. In relation to the workloads, BigDataBench performs various different types of processing:

- **Unstructured data processing** - Text and graph processing, some of the workloads include: Sorting, Word count, Indexing, PageRank, K-Means and Connected components;

- **Semi-structured data processing** - Text processing, workloads include: Basic read/write/scan operations, Collaborative filtering and Naive Bayes;

\(^{13}\)https://amplab.cs.berkeley.edu/benchmark/
Structured data processing - Relational table processing, workloads include relational queries for selection, aggregation and joining.

To evaluate the performance of the processing systems, BigDataBench takes into account two categories of metrics: i) user-perceivable metrics and ii) architectural metrics. User-perceivable metrics are considered metrics that the user is easily aware of. For this category, BigDataBench considers the number of processed requests per second, the number of operations per second and also the amount of data processed per second.

When it comes to the architectural metrics, BigDataBench takes into account the million instructions executed per second (MIPS) and misses per kilo-instruction (MPKI), an utility metric useful when evaluating CPU cache partitioning [23]. The architectural metrics are focused on measuring the performance of workloads when compared to other workloads from different categories, while the user-perceivable metrics are mainly focused on evaluating the performance of workloads from the same categories.

HiBench

HiBench is an open-source Big Data benchmarking tool for MapReduce-based systems such as Hadoop. HiBench was firstly presented in 2011 [24] as a Hadoop benchmark suite. It includes many workload categories, composed by a wide set of Hadoop MapReduce jobs for many smaller benchmarking tasks, as well as larger benchmarking tasks such as real world applications and tasks.

HiBench evaluates the performance of Hadoop in three aspects: i) MapReduce job execution time, ii) Usage of system resources and iii) HDFS bandwidth and throughput.

To evaluate job execution times, throughput and usage of system resources, HiBench mainly uses smaller benchmarking tasks, such as sorting and word counting. For larger workloads HiBench uses tasks that are normally used in the real world such as PageRank. In addition, the HiBench benchmark suite also includes machine learning tasks.

For testing the HDFS component of Hadoop, HiBench provides a revised DFSIO test, which was modified to evaluate the aggregated bandwidth of HDFS. After the introduction of HiBench in 2010, it has been updated in 2012 [25]. Improvements include workloads for Data Warehousing functionalities, such as Join and Aggregation queries, functionalities that are essential to evaluate systems such as Hive.

Some of the most recent features of HiBench include workloads specifically designed to evaluate the streaming processing capabilities of tools such as Spark.

2.6.3 Comparison of Benchmarking Tools

This Section makes a brief review and comparison of the relevant benchmarking tools for Big Data processing that have been discussed so far.

- The Hadoop Benchmarking Tools and AMP Benchmark focus on benchmarking Hadoop and Hive, both based on the MapReduce programming model;

14https://github.com/intel-hadoop/HiBench
• **BigDataBench** and **HiBench** cover a more ample range of Big Data processing tools and paradigms and include support for both batch processing and stream processing.

Table 2.3 summarises these tools and their features.

**Table 2.3: Benchmarking tool comparison.**

<table>
<thead>
<tr>
<th>Authors</th>
<th>Apache Hadoop Distribution</th>
<th>U.C. Berkeley</th>
<th>ICT Chinese Academy of Sciences</th>
<th>Intel Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems supported</td>
<td>Hadoop and Hive</td>
<td>Hive</td>
<td>Hadoop, Hive, Spark and Flink</td>
<td>Hadoop, Spark and Flink</td>
</tr>
<tr>
<td>Datasets</td>
<td>Unstructured</td>
<td>Structured</td>
<td>Unstructured and Structured</td>
<td>Unstructured and Structured</td>
</tr>
<tr>
<td>Workload operations</td>
<td>Stability tests and Batch processing</td>
<td>Batch processing</td>
<td>Batch and Stream processing</td>
<td>Batch and Stream processing</td>
</tr>
<tr>
<td>Metrics</td>
<td>Execution time and usage of system resources</td>
<td>Execution time</td>
<td>Execution time, usage of system resources, MIPS and MPKI</td>
<td>Execution time and usage of system resources</td>
</tr>
</tbody>
</table>

* **TestDFSIO** - September 2016, **NNbench** - March 2016 **TeraSort and MRBench** - April 2016

### 2.7 Summary

This Chapter has discussed various paradigms for processing Big Data, as well as many data processing tools. Unicage provides a set of Unix shell commands that allows users to compose data processing systems. Hadoop enables users to process data in a distributed way using the MapReduce processing model. Hive extends Hadoop by supporting Data Warehousing facilities. Spark and Flink are two systems that provide support for processing streams of data, machine learning and graph processing.

Multiple benchmarking tools for each of the systems have been discussed. Hadoop includes a set of tools designed for testing and benchmarking HDFS and MapReduce. BigDataBench implements and extends the BigBench proposal, providing multiple workloads and a tool for generating structured and unstructured data. The AMP benchmark provides workloads for Data Warehousing systems such as Hive. HiBench covers a wide range of processing systems by supporting workloads for both batch and stream processing.
Chapter 3

LeanBench

This Chapter presents LeanBench, a benchmark proposal for analysing the performance of various Big Data processing systems. Section 3.1 makes a brief overview of the benchmark. Section 3.2 makes a detailed description of the datasets used, Section 3.3 details the operations and their implementations in each system and Section 3.4 presents the benchmark metrics. To conclude, Section 3.5 makes a brief summary of the LeanBench benchmark.

3.1 Leanbench Overview

The Big Data processing systems analysed in Chapter 2 use different paradigms. Unicage provides commands that focus both on batch processing and database querying operations. Hadoop is focused on batch processing while Hive is focused on providing data querying facilities that are not supported by Hadoop. Spark and Flink focus on stream processing.

Since some systems focus on batch processing and others on database querying operations, the LeanBench benchmark will be split into two different benchmarking modules: the first module focuses on batch processing operations and the second module focuses on database querying operations.

The first module of LeanBench - Batch Processing - is used to compare Unicage to Hadoop. Even though Spark and Flink support batch processing they have been designed specifically for stream processing and will not be considered. Figure 3.1 presents the Batch Processing module of LeanBench, in particular, the operations supported and the scripts available.

<table>
<thead>
<tr>
<th>Batch Processing Benchmark</th>
<th>Scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td></td>
</tr>
<tr>
<td>Sort Operation</td>
<td>Run Benchmark Script</td>
</tr>
<tr>
<td>Grep Operation</td>
<td>Verify Results Script</td>
</tr>
<tr>
<td>Wordcount Operation</td>
<td>Statistics Script</td>
</tr>
</tbody>
</table>

Figure 3.1: Composition of the Batch Processing module of LeanBench.
The Sort, Grep and Wordcount operations are implemented in Unicage and Hadoop. These operations serve to measure the performance of batch processing in each system. The scripts serve to run the benchmark operations in these systems multiple times, verify if the operation results are equal, and finally, to produce a file containing the benchmark results and statistics.

The second module of LeanBench - Database Processing - is used to compare Unicage to Hive. Figure 3.2 summarises this module.

![Database Processing Benchmark](image)

**Figure 3.2: Composition of the Database Processing module of LeanBench.**

The querying operations (i.e. Selection, Aggregation and Join) are used to measure and assess the performance of data querying processing in Unicage and Hive. The scripts serve to execute the benchmark with Unicage and Hive multiple times, to verify the querying results and to produce detailed statistics as a result of the benchmark.

The datasets used by the benchmark are produced by BDGS [22], a configurable tool introduced by BigDataBench that generates large amounts of data based on real datasets. The structured datasets are composed by two relational tables, OS.Order and OS.Order.Item. Further information about the benchmark datasets will be presented in Section 3.2.

Table 3.1 shows the benchmark operations that will run in each different system.

![Table 3.1: Benchmark operations for each system.](image)

To compare the scale-out capabilities of the systems, LeanBench has been designed with support for two different scenarios: i) a single machine scenario, and ii) a cluster scenario, with multiple machines working as processing nodes. The first scenario can be used to test the limits of a single machine, and is expected to provide insight of the processing capabilities of a single device. The second scenario enables to verify how each big data processing tool uses the available resources to scale-out the processing. In particular, by dealing with multiple sizes of datasets, it will enable the observation of how
each system uses the cluster resources to scale-out the processing as the amount of data to process increases. The scale-out capabilities of each system can be further tested by running the benchmark operations multiple times, and increasing the number of machines of the cluster in each iteration.

Bottlenecks will also be assessed in each scenario. Three possible bottlenecks are considered: i) the processing can be CPU bound, which means that CPU time is the bottleneck, ii) memory bound, the processing is bottlenecked by the memory capacity and response time, and iii) I/O bound, processing is bottlenecked by disk and network speeds.

### 3.2 Datasets

LeanBench uses two different categories of datasets: i) unstructured and ii) structured.

The unstructured datasets are used on the first part of the benchmark which focuses on processing operations in batch processing. The structured datasets are used on the second part of the benchmark, which is focused on data querying operations.

All of the datasets for the proposed benchmark are provided by the Big Data Generator Suite (BDGS) [22], a scalable data generation tool introduced by BigDataBench. This tool enables the generation of large amounts of data based on real datasets. For generating unstructured data, the tool uses two real datasets: i) English Wikipedia entries and ii) Amazon movie review entries.

For generating structured data, the tool uses an additional real dataset, which is based on anonymous E-commerce transaction data. The structured dataset is composed by the two relational tables described below. The primary key of each table is represented with an underline.

<table>
<thead>
<tr>
<th>Table 3.2: Languages supported by the processing systems.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
</tr>
<tr>
<td>Unix Shell Script</td>
</tr>
</tbody>
</table>

Due to the different languages supported by each system, the operations will be implemented in Unix Shell Script for Unicage, in Java for Hadoop, and in HiveQL for Hive.
The Java Grep and Wordcount operation source code is based on the source code included with the Hadoop Package. The HiveQL queries are based on the queries used in BigDataBench. Section 3.3.1 explains each batch processing operation of the benchmark and presents its implementation, including a listing with the most important blocks of code.

### 3.3.1 Batch Processing Operations

**Sort:** This benchmark operation consists on sorting the words of an unstructured dataset alphabetically.

#### Listing 3.1: Sort - Unicage (Unix Shell Script)

```
#!/home/TOOL/ush

sort -d ../input/dataset.dat > results/Sort_result
```

For Unicage, sorting is performed entirely by a single command, as shown in Listing 3.1, line 3. The `-d` option specifies the sorting criteria to be in alphabetical order. Line 1 specifies the interpreter to be used, which is `ush`, the Unicage command shell script interpreter. All Unicage shell scripts use this interpreter.

As shown in Listing 3.2 for Hadoop, at a first glance, the Sort operation code for Hadoop seems to not serve its functionality, since it does not include any kind of sorting code. There is no code specified for the `map` or `reduce` functions, besides the default predefined Mapper and Reducer classes of Hadoop in lines 3 and 4. In the Hadoop implementation of MapReduce, there is an intermediate Sort phase that sorts the output of the Mapper tasks automatically.

In line 5 we set the input format as a key value input text. As there is no code specified for the `map` or `reduce`, no processing is performed in these phases. In the intermediate Sort phase, Hadoop parses the output of the `map` phase and sorts it alphabetically, which is the default behaviour of the Sort function in Hadoop. In lines 6, 7 and 8 we specify the output format as text, in order to produce an output text file.

#### Listing 3.2: Sort - Apache Hadoop (Java)

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "Sort");
job.setMapperClass(Mapper.class);
job.setReducerClass(Reducer.class);
job.setInputFormatClass(KeyValueTextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```
**Grep:** The grep operation consists on searching and counting the number of words in an unstructured dataset that match multiple regular expressions. There are six different regular expressions used in this operation. These regular expressions have different complexities, and are composed by a prefix matching string, a suffix matching string, or both.

All of the regular expressions used in this operation have different processing complexities and have been chosen for this benchmark operation based on Arturs & Piotr [26]. The different complexities enable to determine how the complexity of a regular expression affects the processing performance. Table 3.3 lists and describes the six regular expressions used in the Grep operation of the benchmark.

<table>
<thead>
<tr>
<th>Description</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match words ending in &quot;es&quot;</td>
<td>\b\w*es\b</td>
</tr>
<tr>
<td>Match words ending in &quot;ies&quot;</td>
<td>\b\w*ies\b</td>
</tr>
<tr>
<td>Match words starting with a vowel</td>
<td>\b([aeiou][AEIOU])\w*\b</td>
</tr>
<tr>
<td>Match words starting with a vowel followed by a consonant</td>
<td>\b([aeiou][AEIOU])[^aeiou]\w*\b</td>
</tr>
<tr>
<td>Match words starting with a vowel followed by another vowel</td>
<td>\b([aeiou][AEIOU])[aeiou]\w*\b</td>
</tr>
<tr>
<td>Match words starting with a vowel, word ends with a vowel</td>
<td>\b([aeiou][AEIOU])[aeiou]\w*[aeiou]\b</td>
</tr>
</tbody>
</table>

Listing 3.3: Grep - Unicage (Unix Shell Script)

```
# ! /home/TOOL/ush
ugrep -E $1 ..//input/dataset.dat |
sort -d |
count 1 1 > results/Grep.result
```

As shown in Listing 3.3, the Grep operation is implemented in Unicage using a sequence of commands: *ugrep, sort* and *count.* The *ugrep* is a command for finding words in a dataset that match a regular expression given as argument. The \(-E\) option enables the usage of extended regular expression patterns, such as \(\w\) (match any word character) and \(\b\) (match a word break character, such as a space or new line).

The *sort* command sorts the words found by the *ugrep* command. Finally, the *count* command counts the words found that match the regular expression specified. The words found by *ugrep* command need to be sorted since the *count* command requires the input to be already sorted.
The Grep operation code for Hadoop consists of two main MapReduce jobs, exemplified in Listing 3.4. The first job searches and counts words that match the regular expression using the `RegexMapper` class specified in line 5. Lines 6 and 7 specify the Combiner and Reducer classes in order to calculate the total count of each word. The second job defined in line 12 sorts the words matched and counted by the previous job. The regular expression to be matched on the words is given as an argument when running the operation. This argument is saved in the configuration for the jobs, as shown in line 1.

**Wordcount:** The Wordcount operation consists on counting the number of occurrences of each word in the unstructured dataset.

In Unicage, the Wordcount operation is implemented using a sequence of two commands, `sort` and `count`, as shown in Listing 3.5. Just like in the Grep operation, the `sort` command sorts the words of the input dataset and the `count` command counts the individual words. Similarly to the Grep operation, the `sort` command is used to sort the input dataset since the `count` command requires the input to be already sorted.
In Hadoop, the Wordcount operation is implemented using two `map` and `reduce` functions, as exemplified in Listing 3.6. The `map` function reads every word from the input dataset, and for each word found, an intermediate `<key,value>` pair is written, consisting of the word found (the key), and the value, which corresponds to the integer value 1. Table 3.4 shows an example input file. This file is processed by the `map` function and produces the `<key,value>` pairs presented in Table 3.5.

Table 3.4: Wordcount input file.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boar</td>
<td></td>
</tr>
<tr>
<td>cheetah</td>
<td></td>
</tr>
<tr>
<td>cheetah</td>
<td></td>
</tr>
<tr>
<td>eagle</td>
<td></td>
</tr>
<tr>
<td>eagle</td>
<td></td>
</tr>
<tr>
<td>eagle</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Key/Value pairs.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boar</td>
<td>1</td>
</tr>
<tr>
<td>cheetah</td>
<td>1</td>
</tr>
<tr>
<td>cheetah</td>
<td>1</td>
</tr>
<tr>
<td>eagle</td>
<td>1</td>
</tr>
<tr>
<td>eagle</td>
<td>1</td>
</tr>
<tr>
<td>eagle</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.6: Final output of the operation.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boar</td>
<td>1</td>
</tr>
<tr>
<td>cheetah</td>
<td>2</td>
</tr>
<tr>
<td>eagle</td>
<td>3</td>
</tr>
</tbody>
</table>

After the `map` task finishes, the `reduce` task goes through all the intermediate `<key,value>` pairs and reduces them: the values are summed for all the words that share the same key, and are then written. Since there is no more data processing to be carried out, the results produced by the `reduce` task will be written as output. Table 3.6 shows the final output after the `reduce` task finishes execution.
3.3.2 Database Querying Operations

Queries in Apache Hive are implemented using a SQL-like querying language called HiveQL. These queries are directly compiled into MapReduce jobs.

In Unicage, the queries are implemented using multiple Unicage commands. Table 3.7 describes the function of all the Unicage commands used in the queries implemented.

Table 3.7: Description of Unicage Querying commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uawk</td>
<td>Select fields (columns) that match a condition</td>
</tr>
<tr>
<td>self</td>
<td>Select fields (columns)</td>
</tr>
<tr>
<td>sm2</td>
<td>Sum values by key</td>
</tr>
<tr>
<td>join1</td>
<td>Join two files using a matching key</td>
</tr>
</tbody>
</table>

In addition to the commands above, the Unix commands `cat` and `tr` are also used in conjunction to modify the dataset columns to be separated by a space instead of being separated by the character “|”. This extra processing step has to performed because Unicage commands expect the dataset columns to be separated by spaces.

Selection Query: The Selection Query selects the Goods Price and Goods Amount columns from the OS_Order_Item table, for entries that have the Goods Amount field greater than 250000. Listings 3.7 and 3.8 show the implementation of this query for Unicage and Hive respectively.

Listing 3.7: Selection - Apache Hive (HiveQL)

```sql
1 DROP TABLE tempSelect;
2 CREATE TABLE tempSelect AS SELECT GOODS_PRICE, GOODS_AMOUNT
3 FROM item WHERE GOODS_AMOUNT > 250000;
```

Listing 3.8: Selection - Unicage (Unix Shell Script)

```bash
1 #!/home/TOOL/ush
2 cat input/OS_ORDER_ITEM.txt | tr '|' ' ' | uawk '${6 > 250000}' | self 5 6 > results/Selection_result
```

Aggregation Query: The Aggregation Query calculates the sum of each Goods Number for each individual Goods ID using an aggregation function as exemplified in Listings 3.9 and 3.10.

Listing 3.9: Aggregation - Apache Hive (HiveQL)

```sql
1 DROP TABLE tempAggregation;
2 CREATE TABLE tempAggregation AS SELECT GOODS_ID, sum(GOODS_NUMBER)
3 FROM item GROUP BY GOODS_ID;
```
Listing 3.10: Aggregation - Unicage (Unix Shell Script)

```
#!/home/TOOL/ush
cat input/OSORDERITEM.txt |
tr ' ' ' ' |
self 3 4 |
sm2 1 1 2 2 > results/Aggregation_result
```

Join Query: The tables OS_Order and OS_Order_Item are joined on the Order ID field. After this, the sum of Goods Amount for each Buyer ID is selected, as shown in Listings and 3.11 and 3.12.

Listing 3.11: Join - Apache Hive (HiveQL)

```
DROP TABLE tempJoin;
CREATE TABLE tempJoin AS SELECT order_item.buyer_id, sum(item.goods_amount) AS total
FROM item JOIN order_item ON item.order_id = order_item.order_id
GROUP BY order_item.buyer_id;
```

Listing 3.12: Join - Unicage (Unix Shell Script)

```
#!/home/TOOL/ush
cat input/OSORDERITEM.txt |
tr ' ' ' ' > input/ORDERITEMFORMATTED.txt
self input/OSORDER.txt |
tr ' ' ' ' > input/ORDERFORMATTED.txt
join1 key=2 input/ORDERFORMATTED.txt input/ORDERITEMFORMATTED.txt |
self 3 8 |
sm2 1 1 2 2 > results/Join_result
rm input/ORDERFORMATTED.txt input/ORDERITEMFORMATTED.txt
```

3.4 Benchmark Metrics

The LeanBench benchmark considers three metrics: the first metric is the execution time of an operation. This metric is used to measure the performance of each operation between the different systems. This metric will also be used for comparing other characteristics of the systems. For example, it can be used to compare the processing times as the dataset sizes increase.

The second metric is the Data Processed per Second (DPS), obtained by dividing the input dataset size by the total processing time in seconds. This metric is useful in the cluster scenario to evaluate the scaling-out processing capacity of the various systems. By adding extra processing nodes in the cluster it will be possible to observe the DPS increase.

The third metric is the usage of system resources. This metric enables to make a comparison of how each system uses the available resources, such as CPU and memory usage for each operation, and also allows to identify processing performance bottlenecks as discussed previously.
Additional factors that are capable of influencing the metrics, such as additional data loading/importing times, start-up and pre-processing times will also be taken into account.

3.5 LeanBench Summary

The LeanBench benchmark is composed by two different categories of operations: i) batch processing operations and ii) data querying operations. The batch processing operations are performed on unstructured datasets, and the data querying operations are performed on structured datasets.

The batch processing operations have been chosen accordingly to the standards of other existing Big Data benchmarking tools, such as BigDataBench and HiBench. As shown previously in Chapter 2, it is common practice in these benchmarks to use operations such as batch sorting and word counting operations applied to large volumes of data in order to benchmark batch processing systems.

The database querying operations have also been chosen to match the standards of other Big Data benchmarking tools for Data Warehousing. For example, the AMP Big Data Benchmark, BigDataBench and HiBench evaluate the performance of a system by running queries based on operations such as Selection, Aggregation, and Join queries.

The metrics chosen for the benchmark have been picked according to what is being evaluated. The first metric is equivalent to metrics in other benchmarks such as BigDataBench and HiBench, which corresponds to the execution time of an operation. The first metric takes into account the execution time of each individual operation, enabling the comparison between two different systems. The second metric, Data Processed per Second (DPS), was chosen to enable a comparison of the scale-out capabilities of each system deployed in a cluster setup. The third metric, usage of system resources, was chosen to enable a comparison of different data processing systems, and also to help identifying bottlenecks.
Chapter 4

Experimental Validation

This Chapter describes the experimental validation of the LeanBench benchmark. The single machine and cluster scenarios supported by the benchmark are described in Section 4.1. Cluster scenario experiments have been designed, but not performed and are explained in the same Section. Single machine scenario experiments have been performed and are presented in Section 4.2, including the setup used and results obtained. Additional notes regarding the evaluation of Unicage are described in Section 4.3. Finally, an overview of all the experiments performed is presented in Section 4.4.

4.1 Evaluation Scenarios

The benchmark is designed to be performed in two different evaluation scenarios: (i) a single machine scenario and (ii) a cluster scenario, with multiple machines working as processing nodes.

4.1.1 Single Machine Scenario

The single machine scenario consists in performing all of the batch processing operations (Sort, Grep and Wordcount) in Unicage and Hadoop and performing all data querying operations (Selection, Aggregation and Join queries) in Unicage and Hive, as described in Table 3.1.

Unicage, Hadoop and Hive are deployed and configured in a standalone operation mode in order to work entirely from a single machine, not requiring any additional hardware.

This first scenario is mainly used to make a comparison between Unicage and Hadoop for the batch processing operations and Unicage and Hive for the data querying operations using a single machine. The scenario can be used to identify the processing capabilities of a single machine. In addition to this, the scenario allows to identify bottlenecks, such as CPU bound, memory bound and I/O bound bottlenecks. Bottlenecks in this particular scenario might be common since a single machine may not have enough resources for processing large data sets entirely at once. For example, the data set size can exceed the machine memory resources, which may cause a memory bound bottleneck in certain operations such as sort.
4.1.2 Cluster Scenario

In the cluster scenario, the benchmark operations would be performed in a cluster composed by multiple machines that support distributed processing. Like in the first scenario, all of the benchmark operations would be performed on the various systems: batch processing operations performed in Unicage and Hadoop and data querying operations performed in Unicage and Hive. However, unlike the first scenario, Unicage and Hadoop would be deployed in a cluster mode, configured to work in a fully-distributed way, using the machines of the cluster as processing nodes.

The cluster scenario would be used to assess how each different data processing system uses the available cluster resources to scale-out processing. The scaling-out capabilities of each system would be evaluated according to the second metric of LeanBench, \textit{data processed per second}. Bottlenecks for each of the operations would also be identified in this scenario similarly to the first scenario. However, in this scenario I/O bound bottlenecks would be more likely to occur because the multiple machines are connected by a network. The speed of this network would affect the processing speeds if the data is transferred slower than the machines can process it.

In order for Unicage to be configured in a cluster setup, each processing node requires a license of \textit{uspTukubai}. In addition, a license of \textit{Unicage BOA}, the Unicage Big Data oriented architecture cluster software, is required to deploy Unicage in a cluster configuration. The cluster scenario evaluation was not carried out due to time constraints and issues regarding the Unicage individual licenses required.

4.2 Single Machine Experiments

This section will present the two experiment sets performed in this work. The first set served the purpose of identifying the processing limitations of a machine with low hardware specifications. A second set of experiments has been performed on a different machine with higher hardware specifications, allowing to process datasets of larger volume. Both experiment sets have been carried out according to the single machine scenario description in Section 4.1.1. Unicage \textit{uspTukubai}, Hadoop 2.7.3 and Hive 2.1.1 have been used for both sets of experiments. Each experiment set will describe the setup used, including the hardware configurations and dataset sizes. The datasets used in both experiments have been generated by BDGS, as described previously in Section 3.2. The unstructured dataset sizes are described in Megabytes (MB) or Gigabytes (GB). The structured dataset sizes are described as the total number of rows. Each structured dataset row takes up approximately 85 Bytes in size.

The results of the benchmark take into account the first metric of the benchmark (execution time) and will be presented in the following way: a bar graph describes the average execution times for all operations in each system, three line graphs show the average operation execution time in each system for each operation. The operation execution times presented are the total execution time of the operations, considering the time each system took to process, from start to finish. A final line graph presents the relative performance between two systems, in percentage, and is obtained according to the formula presented in equation 4.1. For example, a value of 10 would represent a 10% performance increase.
Relative Performance = \left( \frac{\text{Average operation execution time of the slowest system}}{\text{Average operation execution time of the fastest system}} \times 100 \right) - 100 \quad (4.1)

In Hadoop, each operation is a job that has to be read and submitted in the system. All of the samples obtained in this evaluation had a job submission time\(^1\) of less than one second. The job submission times for Hadoop are included in values presented in the graphs but are not identified as they take a very small fraction of the total execution time. The same applies for Hive, as it uses Hadoop for processing. In Unicage the processing starts immediately as the script is executed. The values of the Average Grep operation presented in each graph represent the average of all individual Grep operations executed using the six different regular expressions described in Table 3.3.

### 4.2.1 Experiment Set 1

The first single machine benchmark experiment set has been performed on a machine with low system specifications. Table 4.1 describes the hardware specifications of this machine.

<table>
<thead>
<tr>
<th>Machine Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS: Linux 4.4.0-66-generic # 87-Ubuntu SMP x86_64 GNU/Linux</td>
</tr>
<tr>
<td>CPU: Intel Core i3-2350M @ 2.30GHz</td>
</tr>
<tr>
<td>RAM: 2048MB DDR3-1333</td>
</tr>
<tr>
<td>HDD: 60GB 5400rpm SATA HDD</td>
</tr>
</tbody>
</table>

Three unstructured datasets have been used to perform the Sort, Grep and Wordcount operations. The first and third have been generated based on the Amazon movie review entries dataset and the second has been generated based on the English Wikipedia entries dataset. Each one of the datasets has five different volumes of data, adding up to a total of 15.

For performing the structured data querying benchmark operations, multiple structured datasets have been used, with dataset sizes ranging from 100K rows to 10M rows. These structured datasets have been generated based on a real anonymous E-commerce dataset. Table 4.2 summarises all of the datasets used in this experiment.

<table>
<thead>
<tr>
<th>Dataset Description</th>
<th>Volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstructured Dataset #1 (AmazonMR1)</td>
<td>15 MB, 50 MB, 100 MB, 150 MB and 300 MB</td>
</tr>
<tr>
<td>Unstructured Dataset #2 (Wikipedia)</td>
<td>15 MB, 50 MB, 100 MB, 150 MB and 300 MB</td>
</tr>
<tr>
<td>Unstructured Dataset #3 (AmazonMR2)</td>
<td>15 MB, 50 MB, 100 MB, 150 MB and 300 MB</td>
</tr>
<tr>
<td>Structured Datasets (E-commerce)</td>
<td>100K, 400K, 800K, 1.6M and 3.2M Rows</td>
</tr>
</tbody>
</table>

\(^1\)Time that Hadoop takes to read and submit the jar file containing the processing operation. The submission time for each Job can be obtained from the Hadoop job logs.
Each one of the batch processing operations of LeanBench was performed using each unstructured dataset described above, in both Unicage and Hadoop. All of the operations have been performed 10 times for the 15, 50 and 100 MB volume variants of the datasets. For the 150 and 300 MB volume variants of the datasets the operations have been performed 5 times. The database querying operations of LeanBench have been performed 10 times for each structured dataset.

Additional tests have been performed after the initial tests described above. These additional tests have been performed by the same machine mentioned in Table 4.1, but used larger datasets, described in Table 4.3.

### Additional Dataset List

<table>
<thead>
<tr>
<th>Dataset Description</th>
<th>Volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstructured Datasets (AmazonMR1)</td>
<td>2000 MB and 5000 MB</td>
</tr>
<tr>
<td>Structured Datasets (E-commerce)</td>
<td>10M, 25M, 50M and 100M Rows</td>
</tr>
</tbody>
</table>

The additional benchmarking tests with larger datasets have been performed in order to identify how each processing system would respond to dataset sizes that exceed the machine hardware specifications. Due to the long processing times, the batch processing operations have been performed twice for the 2000 MB dataset, and once for the 5000 MB dataset. The database querying operations have been performed 5 times each.

### Batch Processing Results

Figure 4.1 summarises the batch processing results obtained in the first experiment set. The results presented describe the average execution times for each operation performed in Hadoop and Unicage.
Figures 4.2, 4.3 and 4.4 describe the average execution times in Hadoop and Unicage for the Sort, Average Grep and Wordcount operations respectively. As expected, the average execution times of an operation increase in both systems as the amount of data to process grows.
Figure 4.4: Wordcount Operation - Experiment Set 1.

Figure 4.5 presents the performance of Unicage relative to Hadoop in percentage per operation.

Unicage is faster than Hadoop except for the Wordcount operation with datasets above 100 MB, as observed in Figures 4.4 and 4.5. The Wordcount operation in Unicage, as described in Section 3.3.1, starts by sorting the entire input before the entries are counted, while Hadoop sorts the entries after they have been processed in the map phase, which results in a smaller volume of data to sort.
Batch Processing Results - Additional Tests

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DS2GB</td>
<td>1772.145</td>
<td>880.04</td>
<td>279.789</td>
<td>250.493</td>
<td>660.455</td>
<td>781.41</td>
</tr>
<tr>
<td>DS5GB</td>
<td>4234.78</td>
<td>2255.07</td>
<td>682.32</td>
<td>625.3</td>
<td>1656.85</td>
<td>2384.96</td>
</tr>
</tbody>
</table>

Figure 4.6: Batch Processing Operations - Experiment Set 1 - Additional Tests.

Figure 4.6 summarises the results obtained in the first experiment set with the larger datasets. The average execution times for each operation in Hadoop and Unicage are presented.

Figures 4.7, 4.8 and 4.9 show the average execution times in Hadoop and Unicage for the Sort, Average Grep and Wordcount operations respectively, for processing the 2000 MB and 5000 MB unstructured datasets.
Unicage continues to be slower in the Wordcount operation when compared to Hadoop. The Sort and Grep operations remain faster in Unicage, which confirms the results obtained previously with smaller datasets.

Figure 4.10 presents the percentual performance of Unicage relative to Hadoop when processing the additional larger datasets of 2000 MB and 5000 MB.
The processing times continue to increase significantly in both systems. Unicage is still faster than Hadoop for the Sort and Grep operations, however, there is a decrease in performance as the volume of data to process increases. For the Wordcount operation, the Unicage performance is lower and also continues to decline when compared to Hadoop, as evidenced in Figure 4.10.

**Data Querying Results**

![Data Querying Operations - Experiment Set 1 (Logarithmic Scale).](image)

Figure 4.11: Data Querying Operations - Experiment Set 1 (Logarithmic Scale).
Figure 4.11 summarises the average Selection, Aggregation and Join execution times of Hive and Unicage obtained in the first experiment set. This graph uses a logarithmic scale due to the large difference of operation execution times between the two systems.

Figures 4.12, 4.13 and 4.14 describe the average execution times in Hive and Unicage for the various querying operations of the benchmark.
Figure 4.14: Join Query - Experiment Set 1.

Figure 4.15 presents the percentual performance of Unicage relative to Hive.

Figure 4.15: Unicage vs. Hive - Experiment Set 1.

Unicage is faster than Hive in all of the querying operations, for all the dataset volumes used. However, while the Unicage performance maintains to be roughly 10 times as faster than Hive for the largest dataset processed, there is a substantial decrease in performance as the dataset sizes increase, as clearly described in Figure 4.15.
Data Querying Results - Additional Tests

Figure 4.16 summarises the average Selection, Aggregation and Join execution times of Hive and Unicage obtained in the first experiment set by using larger datasets of 25, 50 and 100 million rows.

Figures 4.17, 4.18 and 4.19 present the average execution times of Selection, Aggregation and Join queries respectively, using the additional larger datasets. The average execution times of each system continue to increase as larger datasets are used.

Figure 4.17: Selection Query - Experiment Set 1 - Additional Tests.

Figure 4.18: Aggregation Query - Experiment Set 1 - Additional Tests.

Figure 4.19: Join Query - Experiment Set 1 - Additional Tests.
Figure 4.18: Aggregation Query - Experiment Set 1 - Additional Tests.

Figure 4.19: Join Query - Experiment Set 1 - Additional Tests.

Figure 4.20 shows the percentual performance of Unicage relative to Hive.

Unicage continues to be faster than Hive when processing the larger datasets of 10, 25, 50 and 100 million rows, in all of the benchmark queries. The results obtained confirm the results obtained previously when using smaller datasets, the performance of Unicage relative to Hive remains high, but continues to decrease as larger datasets are used, as observed in Figure 4.20.
4.2.2 Experiment Set 2

After reaching the limitations of the machine used in the first experiment set, a second machine was used, with higher hardware specifications. The hardware specifications of the machine used in this second experiment are described in Table 4.4.

Table 4.4: Single Machine Scenario - Experiment Set 2 - Hardware used (Machine #2).

<table>
<thead>
<tr>
<th>Machine Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS:</strong> Linux 3.13.0-117-generic #164-Ubuntu SMP x86_64 GNU/Linux</td>
</tr>
<tr>
<td><strong>CPU:</strong> Intel Xeon E5506 @ 2.13GHz (Using two cores)</td>
</tr>
<tr>
<td><strong>RAM:</strong> 10GB DDR3-1066</td>
</tr>
<tr>
<td><strong>HDD:</strong> 100GB 7200rpm SATA HDD</td>
</tr>
</tbody>
</table>

The higher hardware specifications of this machine allowed to use larger dataset sizes for performing the benchmark operations. Table 4.5 lists all of the datasets used in this experiment.

Table 4.5: Single Machine Scenario - Experiment Set 2 - Dataset List.

<table>
<thead>
<tr>
<th>Dataset Description</th>
<th>Volumes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstructured Datasets (AmazonMR1)</td>
<td>5GB, 10GB, 20GB, 40GB and 60GB</td>
</tr>
<tr>
<td>Structured Datasets (E-commerce)</td>
<td>200M and 400M Rows</td>
</tr>
</tbody>
</table>

The batch processing operations have been performed 5 times for the 5 GB dataset, twice for the 10 and 20 GB datasets, and once for the 40 and 60 GB datasets. The database querying operations have been performed 10 times for each structured dataset.
**Batch Processing Results**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DS5GB</td>
<td>6194.4975</td>
<td>2977.418</td>
<td>2788.93</td>
<td>1049.503</td>
<td>1948.055</td>
<td>3022.976</td>
</tr>
<tr>
<td>DS10GB</td>
<td>13073.565</td>
<td>6065.295</td>
<td>1546.541</td>
<td>1049.503</td>
<td>3918.08</td>
<td>6105.88</td>
</tr>
<tr>
<td>DS20GB</td>
<td>17328.625</td>
<td>12779.62</td>
<td>3045.147</td>
<td>2144.1</td>
<td>7771.7</td>
<td>12475.28</td>
</tr>
</tbody>
</table>

Figure 4.21: Batch Processing Operations - Experiment Set 2.

Figure 4.21 summarises the average operation execution times of Hadoop and Unicage obtained in the second experiment set. Figures 4.22, 4.23 and 4.24 present the average execution times for the Sort, Average Grep and Wordcount operations respectively. Similar to all the results obtained previously, the average execution times continue to increase with larger datasets.

Figure 4.22: Sort Operation - Experiment Set 2.
As observed in Figure 4.24, Unicage continues to be slower than Hadoop when performing the Wordcount operation, confirming the results obtained previously. Furthermore, as the amount of data to process increases, the Wordcount average execution times increase more in Unicage than in Hadoop, as evidenced by the purple line of Figure 4.24.
Figure 4.25 shows the percentual performance of Unicage relative to Hadoop.

![Figure 4.25: Unicage vs. Hadoop - Experiment Set 2.](image)

Unicage remains faster than Hadoop for the Sort and Grep operations, using the 5, 10 and 20 GB datasets, but there is a significant drop in performance as the dataset sizes increase, similar to previous results. For the Wordcount operation, Unicage is slower and maintains a decline in performance which confirms the previous tests.

Additional tests using 40 and 60 GB datasets have been performed with the Grep and Wordcount operations. Table 4.6 presents the average operation execution times obtained in these tests.

<table>
<thead>
<tr>
<th>System</th>
<th>40 GB Dataset</th>
<th>60 GB Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicage Avg. Grep</td>
<td>4458.262 seconds</td>
<td>Failed execution</td>
</tr>
<tr>
<td>Hadoop Avg. Grep</td>
<td>6359.157 seconds</td>
<td>10669.153 seconds</td>
</tr>
<tr>
<td>Unicage Wordcount</td>
<td>23284.28 seconds</td>
<td>Failed execution</td>
</tr>
<tr>
<td>Hadoop Wordcount</td>
<td>15840 seconds</td>
<td>25108.92 seconds</td>
</tr>
</tbody>
</table>

For the 40 GB dataset, Unicage is faster than Hadoop for the Grep operations, and slower for the Wordcount operation. However, when testing the same operations with a 60 GB dataset, Unicage terminated execution in the middle of processing, failing to produce a valid output for the Grep operations with (Prefix 1 and Prefix 3), and also for the Wordcount operation.

While the Prefix 1 and Prefix 3 regular expressions described in Section 3.3.1 are not the most complex to match, the expressions match a large portion of the entries in the Dataset, resulting in a large volume of entries matched to count, which may explain why Unicage failed to process the data, similar to the Wordcount operation.
Data Querying Results

Figure 4.26 summarises the average query execution times obtained in the second experiment set, with Hive and Unicage when using the 200 and 400 million rows datasets.

![Data Querying Operations - Experiment Set 2](image)

Figure 4.26: Data Querying Operations - Experiment Set 2.

Figures 4.27, 4.28 and 4.29 show the Selection, Aggregation and Join queries average execution times obtained in this evaluation. As evidenced in the lines of the following Figures, the execution times increase with the number of rows, but Hive execution times increase at a higher rate than Unicage.

![Selection Query - Experiment Set 2](image)

Figure 4.27: Selection Query - Experiment Set 2.
Figure 4.28: Aggregation Query - Experiment Set 2.

Figure 4.29: Join Query - Experiment Set 2.

Figure 4.30 describes the percentual performance of Unicage relative to Hive.

Similar to the previous querying evaluations, Unicage is faster than Hive when processing the datasets of 200 and 400 million rows, in all the benchmark queries. The Unicage performance relative to Hive remains high, increasing slightly for the 400 million rows dataset in Aggregation and Join queries.
4.3 Additional Evaluations

This Section describes additional evaluations performed in this work, including an evaluation of alternate sorting commands for Unicage and an evaluation of costs for splitting datasets in Unicage.

4.3.1 Alternative Sort Implementations

Unicage provides a custom sorting command (msort) based on the merge sort algorithm, as described previously in Section 2.1.1. A separate evaluation of this command has been performed. This evaluation proved that while the command performs faster than the Unix sort command for small datasets, it lacks support for external sorting\(^2\). In addition, the command documentation refers that the msort command uses up to a maximum of three times the size of the dataset being sorted. However, the command consumes an higher amount of memory than what is described, which can lead to the command exhausting all of the system memory resources. For example, according to the msort documentation, a dataset of 100MB should only require up to a maximum of 300MB of memory, however, the command actually consumes a much higher amount of memory.

Without any option to limit memory usage or support for external sorting, the command uses the entire amount of memory and swap space of the machine, leaving it in an unresponsive state for multiple hours, producing no output. This can occur for dataset sizes as low as 100 MB in a machine with 2 Gigabytes of memory. Tables 4.7 and 4.8 describe the execution times and memory usages for the sort command of Unix and the msort of Unicage, allowing to make a comparison of both sorting commands.

\(^2\)External Sorting: A technique that allows sorting algorithms to sort very large datasets (that do not fit in memory) by using external storage such as the hard-drive to store intermediate processing information.
This test was performed on the machine used in the second set of experiments, described previously in Table 4.4.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Execution Time (seconds)</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 MB</td>
<td>92.82</td>
<td>1347 MB</td>
</tr>
<tr>
<td>200 MB</td>
<td>194.97</td>
<td>2694 MB</td>
</tr>
<tr>
<td>400 MB</td>
<td>411.78</td>
<td>5385 MB</td>
</tr>
<tr>
<td>800 MB</td>
<td>857.06</td>
<td>7499 MB</td>
</tr>
</tbody>
</table>

Table 4.7: Unix `sort` Command.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Execution Time (seconds)</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 MB</td>
<td>16.53</td>
<td>2282 MB</td>
</tr>
<tr>
<td>200 MB</td>
<td>34.51</td>
<td>4565 MB</td>
</tr>
<tr>
<td>400 MB</td>
<td>71.18</td>
<td>10482 MB</td>
</tr>
<tr>
<td>800 MB</td>
<td>Did not terminate</td>
<td>13161 MB</td>
</tr>
</tbody>
</table>

Table 4.8: Unicage `msort` Command.

The lack of external sorting support in addition to a very high memory consumption makes the command unfeasible to use when processing large datasets in a single machine. For the reasons above, it was decided that the batch processing operations of LeanBench implemented in Unicage would use the Unix `sort` command, as it supports external sorting, and therefore, allows to perform tests with large datasets without being limited by the system memory resources. If the Unicage operations used the `msort` operation instead, it would not be possible to perform most of the experiments presented in this work.

4.3.2 Dataset Splitting

Unicage also provides other commands that allow to split datasets in multiple parts (`ocat` and `ocut`). There are additional costs for processing datasets that are split: not only for splitting the dataset into multiple parts but also in the joining process after the multiple parts have been processed. Even though the multiple evaluations performed by LeanBench in this work focused on processing datasets composed by a single file, a separate test was performed in order to assess the costs of splitting datasets. The tests used two different datasets and three different sizes of splitted parts. Table 4.9 summarises the results obtained.

Table 4.9: Unicage - Average splitting costs (seconds) - 10 Samples.

<table>
<thead>
<tr>
<th>Operation</th>
<th>200MB Splits</th>
<th>500MB Splits</th>
<th>1000MB Splits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split - 20GB Dataset</td>
<td>572.677</td>
<td>567.335</td>
<td>569.065</td>
</tr>
<tr>
<td>Split - 40GB Dataset</td>
<td>1182.247</td>
<td>1176.253</td>
<td>1182.171</td>
</tr>
</tbody>
</table>

The minimum average cost of joining back the 20 GB dataset (based on 10 samples) was 628.333 seconds, and 1200.103 seconds for joining a 40 GB dataset. The amount of splits to join did not affect the joining costs. It is also important to note that the join costs presented take only into account the cost
of joining the splitted parts back into one file. In most operations, such as Sort and Wordcount, there are additional joining costs: for example, in the Sort operation there is the cost of merging the splitted parts back together without losing the sorted entries, and for the Wordcount operation there is an additional cost of calculating the total values (sum) of the entries that share the same key. This shows that splitting datasets introduces significant costs that have to be considered as part of processing times.

4.4 Experiments Overview

The batch processing operations of LeanBench performed in both experiments show that Unicage performs better than Hadoop in all of the operations for datasets sized below 100MB. When processing datasets larger than 100MB, Unicage performs always worse than Hadoop in the Wordcount operation. For the remaining batch processing operations, Sort and Grep, Unicage is faster than Hadoop, but the performance also declines as the dataset sizes increase. This could indicate that there is a point where Unicage starts performing worse than Hadoop. In particular, for the largest dataset processed in the second experiment (60 GB), Unicage failed to produce a valid output in both Grep and Wordcount operations. In general, the performance declines as the dataset sizes increase, in all of the benchmark operations performed in Unicage.

However, for the data querying operations of LeanBench, Unicage always performed better than Hive in all of the operations. The performance of Unicage when compared to Hive also declines as the dataset sizes increase, but there are cases where there is a performance increase. For example: in the second set of experiments, Unicage performance in the Aggregation and Joining queries for the 400 million row dataset slightly increased when compared to the same queries performed in the 200 million row dataset.

When it comes to bottlenecks, in exception of the memory bound bottlenecks identified when using the Unicage msort command, presented in Section 4.3, system resource usage logs show that all operations of LeanBench are mainly limited by the CPU time in both systems. The memory usage never exceeds the total system memory, even when the dataset sizes are larger than the system memory. This can be explained due to the fact that Hadoop, and Unicage (when using the Unix sort command) support external sorting techniques, as described in Section 4.3. Having additional memory would allow to process larger datasets without having to rely on external sorting, which uses secondary slower memory such as the hard-drive, and could therefore introduce a bottleneck.

However, in a single machine setup, having support for external sorting capabilities is essential in a processing system, as the dataset sizes can easily surpass the system memory capacity. In addition, introducing additional system memory is not always an option in a single machine setup as there are physical and hardware limitations of how much memory can be added to a system.
Chapter 5

Conclusion

This thesis proposes LeanBench, a benchmark tool for evaluating the performance of Big Data processing systems. LeanBench was constructed with benchmarking operations and metrics chosen accordingly to the standards of other existing Big Data benchmarking tools, such as BigDataBench and HiBench. LeanBench is composed by two benchmarking modules: i) Batch Processing and ii) Database Processing. These two modules allow to benchmark two different categories of systems: batch processing systems such as Hadoop, and data warehousing systems that include data querying functionalities, such as Hive. In addition, LeanBench includes support for benchmarking Unicage, a tool composed by multiple individual commands that, in conjunction, are able to perform both batch processing and database querying tasks.

This Chapter presents the conclusions obtained as result of the experiments performed using LeanBench, presented in Chapter 4. Section 5.1 summarises the results obtained by benchmarking the processing systems. Section 5.2 presents multiple guidelines to help choosing a system according to specific processing tasks. Future Work regarding the cluster scenario evaluation is described in Section 5.3. To conclude, Section 5.4 presents a brief discussion on the cost and value of abstractions.

5.1 Results Overview

The experiments performed with the LeanBench benchmarking modules allowed us to make a performance comparison of the different processing systems. When it comes to database querying tasks, Unicage has an advantage over Hive. Unicage is able to produce results significantly faster than Hive in all the queries of the benchmark. These results can be explained due to the small software stack of Unicage. Unicage is composed by individual commands that provide a single functionality, while Hive relies on the entire Hadoop stack for performing the processing. In Unicage, the only processing that is performed consists of the individual processing tasks carried out by each command listed in the querying script of the task. In comparison with Hive, the HiveQL queries are first compiled to a Hadoop MapReduce job. Hadoop is responsible for processing the data and returning the results to Hive. The querying language of Hive presents an advantage with respect to the Unicage scripts as users can directly write
queries in *HiveQL*, a process very similar to writing SQL queries. In Unicage, it is not possible to write queries directly in an SQL-like language, and queries must instead be written using a combination of the individual low-level processing commands of Unicage.

When it comes to batch processing tasks, while Unicage performs better than Hadoop when processing volumes of data smaller than 100MB, the lack of essential features such as support for external sorting makes Unicage to perform worse than Hadoop when processing larger volumes of data. As presented in Chapter 4, Unicage provides commands to split datasets into multiple individual parts to facilitate processing large volumes of data. Nevertheless, splitting datasets introduces additional costs as well as the joining operations do, required after the main processing is performed. In addition, dataset splitting and joining is not automatic and must be manually programmed in the user script.

In conclusion, in a single machine scenario, the small software stack of Unicage presents advantages and disadvantages. The small software stack of Unicage allows to process some tasks faster than other systems by avoiding unnecessary processing steps. However, for tasks that require sorting large volumes of data, Unicage presents a disadvantage when compared to the other systems presented due to the lack of support for external sorting techniques, an essential feature when processing large datasets in single machine, as the volume of data can easily exceed the system memory resources.

### 5.2 Guidelines

As concluded in this thesis, the best processing system to use depends largely on the processing task. In general, for batch processing, if the processing task requires sorting large volumes of data (larger than 100MB), Hadoop is the best choice. If the processing task does not involve sorting large volumes of data then Unicage is a faster alternative to Hadoop. Unicage is the best choice for processing tasks that involve querying structured datasets. If a querying language similar to SQL is necessary then Hive is the best choice as it supports *HiveQL*, at a cost of performing significantly slower than Unicage.

Figure 5.1 presents a flowchart with the best system choice for some of the most common processing tasks and requirements.

![Flowchart](image.png)

**Figure 5.1:** Best processing system according to different processing tasks and specific requirements.
5.3 Future Work

The experiments performed and repeated in Section 4.2 allow us to make a single machine comparison between Unicage and Hadoop for batch processing tasks, and a comparison between Unicage and Hive for database processing tasks. Nonetheless, the single machine experiments performed do not allow to make an assessment of how the different systems perform in a cluster configuration, where the processing is distributed. Using a single machine not only restricts the volume of data that can be analysed in experiments, but it forces processing systems to work in a single machine, when they have been designed to work in a cluster setup, which is the case with Hadoop. Due to time restrictions and license limitations, it was not possible to perform a cluster evaluation in this work.

When dealing with Big Data, a cluster setup is the most commonly used. The multiple processing nodes of a cluster allow to scale-out the processing by distributing the work along nodes, allowing to process larger amounts of data significantly faster when compared to a single machine setup. In addition, the cluster setup can always be expanded by adding extra processing nodes. This is not possible with a single machine configuration.

To assess how the processing systems compare in a cluster scenario, a dedicated cluster evaluation has to be carried out. Cluster benchmarking experiments require the different systems to be deployed in a cluster, with multiple machines working as processing nodes. Hadoop supports cluster configurations natively, but Unicage is designed to work in a single machine, and requires additional software to be able to be deployed in a cluster. When it comes to the benchmarking tool, LeanBench has been designed to work in both single machine and cluster scenarios.

In conclusion, the single machine experiments performed in this thesis have allowed to make a performance comparison of the systems when deployed in a single machine configuration, but a future cluster evaluation is necessary to make a performance assessment of each processing system when deployed in a cluster scenario.

5.4 The Cost and Value of Abstractions

Multiple processing systems with two distinct software stack sizes have been analysed and benchmarked in this work. Unicage provides faster processing performance in multiple tasks when compared to Hadoop and Hive, two systems that rely on large and complex software stacks. However, the additional software layers of Hadoop and Hive provide many useful abstractions and functionalities that are lacking in Unicage.

Hadoop includes a dedicated distributed file system, allowing to scale-out the storage across multiple machines automatically, which avoids having to split data manually, an issue present in Unicage. In addition, the distributed file system of Hadoop also provides data replication mechanisms that allow the system to recover from hardware failures.

Hive allows users to write queries in a high level querying language. Queries in Unicage have to be constructed resorting to multiple low-level commands that perform individual processing tasks,
optimised to the dataset being processed. Processing a different dataset requires constructing a new querying script optimised to that dataset in specific, similar to what was discussed previously in Section 2.3. In order to query structured datasets using Hadoop before the existence of Hive, users had to write their own MapReduce tasks, a time consuming task. Hive solved this issue by providing HiveQL, an abstraction layer on top of Hadoop that allows users to construct queries executed using the MapReduce framework without having to write complex MapReduce tasks.

In conclusion, there are multiple systems designed for each kind of processing task. The guidelines presented in this work have been constructed with the goal of identifying the type of processing task and the tool that is optimised for that specific task. There is not a single system that is best for every existing task, as each processing system presents distinct advantages and disadvantages. This work has presented a quantification of the cost in processing performance by having large software stacks with additional layers. Unicage could benefit by having additional abstraction layers that would allow it to handle the processing of large datasets. For example, Unicage could provide a compiler that translates SQL queries to Unicage scripts. The additional software layers of Hadoop and Hive provide ease of use for developers by providing multiple features that are characteristic of Big Data processing systems, allowing to reduce development and maintenance times, at a cost of slower processing performance: the value provided by the additional abstraction layers can outweigh the costs, in short, there is a tradeoff between development and processing complexities.
Bibliography


databench: A big data benchmark suite from internet services. In 2014 IEEE 20th International

[23] X. Lin and R. Balasubramonian. Refining the utility metric for utility-based cache partitioning. In
Proc. Workshop on Duplicating, Deconstructing, and Debunking (WDDD), 2011.

of the mapreduce-based data analysis. In New Frontiers in Information and Software as Services,


[26] A. Backurs and P. Indyk. Which regular expression patterns are hard to match? In 2016 IEEE 57th
Annual Symposium on Foundations of Computer Science (FOCS), pages 457–466, Oct 2016. doi:
10.1109/FOCS.2016.56.

Appendix A

Validation of Results

This Section describes the statistical functions and techniques that can be used to validate all of the benchmark experiments. Douglas & George [27] can provide a more comprehensive reference about the theory used in this section.

A.1 Population Mean estimation based on Samples

The LeanBench benchmark includes two different scenarios with multiple benchmark operations, each of them being performed in at least two different systems. Because of this, the number of experiments is vast and it may not be feasible to obtain a large number of samples for all of the experiments. However, by using statistical functions, it is possible to use a smaller number of samples to estimate the population behaviour for each of the experiments.

By repeating an experiment \( n \) times, it is possible to obtain the sample mean \( \bar{x} \) and the standard deviation \( \sigma \).

Equation A.1 shows how to estimate the population behaviour for a specific Experiment A. \( \mu \) is the estimated population mean of Experiment A with a confidence interval \( CI \), using a confidence level of \( \alpha \).

\[
\text{Experiment A} = \mu \pm CI\text{ seconds with }\alpha\text{ of confidence (A.1)}
\]

If all the samples are independent from each other and all come from the same population, using the sample mean \( \bar{x} \) as an estimation of the population mean \( \mu \), and standard deviation \( \sigma \) then the Central Limit Theorem (CLT) can be used to calculate a confidence interval \( CI \) around the population mean \( \mu \).

The confidence interval calculation depends on the number of samples used for each experiment. Since each individual experiment in this work never exceeded thirty samples, the confidence interval \( CI \), according to CLT, can be calculated using the t-Distribution as described in equation A.2:

\[
CI = \pm t_{\alpha/2, n-1} \frac{s}{\sqrt{n}}
\]  

(A.2)
A.2 Choice of Sample Size

The margin of error in the population mean estimation depends on two factors: i) the chosen confidence level and ii) the total number of experiment samples performed. If the standard deviation $\sigma$ of the population is known and if the sample mean $\bar{x}$ is used as an estimate of $\mu$ then we can be $100(1 - \alpha)\%$ confident that the error $|\bar{x} - \mu|$ does not exceed a margin of error $E$ for a sample of size $n$, as shown in Equation A.2.

$$n = \left( \frac{z_{\alpha/2}}{E} \right)^2 (0.25)$$  \hspace{1cm} (A.3)

Using equation A.3, it is possible to calculate the maximum margin of error for a sample size, using the typical confidence levels. Table A.1 describes the maximum margin of error for a certain sample size and confidence level.

<table>
<thead>
<tr>
<th>Samples</th>
<th>Confidence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90%</td>
</tr>
<tr>
<td>5</td>
<td>36.783%</td>
</tr>
<tr>
<td>10</td>
<td>25.010%</td>
</tr>
<tr>
<td>15</td>
<td>21.237%</td>
</tr>
<tr>
<td>20</td>
<td>18.392%</td>
</tr>
<tr>
<td>25</td>
<td>16.450%</td>
</tr>
<tr>
<td>30</td>
<td>15.017%</td>
</tr>
<tr>
<td>40</td>
<td>13.005%</td>
</tr>
<tr>
<td>45</td>
<td>12.261%</td>
</tr>
<tr>
<td>50</td>
<td>11.632%</td>
</tr>
<tr>
<td>55</td>
<td>11.091%</td>
</tr>
<tr>
<td>60</td>
<td>10.618%</td>
</tr>
<tr>
<td>65</td>
<td>10.202%</td>
</tr>
<tr>
<td>70</td>
<td>9.831%</td>
</tr>
<tr>
<td>75</td>
<td>9.497%</td>
</tr>
<tr>
<td>80</td>
<td>9.196%</td>
</tr>
<tr>
<td>85</td>
<td>8.921%</td>
</tr>
<tr>
<td>90</td>
<td>8.670%</td>
</tr>
<tr>
<td>95</td>
<td>8.439%</td>
</tr>
<tr>
<td>100</td>
<td>8.225%</td>
</tr>
</tbody>
</table>

As observed in table A.1, the margin of error is the highest for lower sample sizes, and the highest difference is between five and ten samples.

The margin of error values obtained from equation A.3 and presented in table A.1 are the highest possible, and not the actual margin of error of each single experiment. However, we can be assured that the actual margin of error never exceeds the values presented.