A MICROARCHITECTURAL ANALYSIS OF
MACHINE LEARNING ALGORITHMS ON SPARK

by

José Antonio Muñoz Cepillo

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2017 by José Antonio Muñoz Cepillo
Abstract

A Microarchitectural Analysis of Machine Learning Algorithms on Spark

José Antonio Muñoz Cepillo
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

Analysis of large data sets utilizing Machine Learning Algorithms (MLAs) is an important research and application area. This thesis presents a comprehensive performance analysis of MLAs on the Apache™ Spark platform, identifying bottlenecks at the microarchitectural level, and associating them to higher-level code constructs. In addition, we propose solutions to address the discovered performance issues.

Our results indicate that: 1) The majority of MLAs are backend-bound, in particular memory-bound. 2) The dot product is the main bottleneck in a considerable number of MLAs. 3) Garbage collection and high-level abstractions are a major source of overhead in Spark. 4) Enabling Hyper-Threading is a recommended feature that improves performance up to 52%. 5) The improved data representation of the Dataset API results in performance improvements of up to 64% over the RDD API. 6) Enabling software caching in the right place improves the performance between 6%-23%.
Acknowledgements

First and foremost, I would like to acknowledge the guidance and contributions of my supervisor, Prof. Andreas Moshovos. Special thanks go to Prof. Nick Koudas, for all his help and support.

I would like to thank all of the people in our Computer Architecture group and the people in Pratt 477 for their support, interesting discussions, and camaraderie. I am going to miss you all, the ones that are still here and the ones that already left.

I would like to especially thank Patrick Judd, Mario Badr, Emilianna Machado, Yu Ting (Joy) Chen, Ernesto Díaz, Jorge Albericio and Alberto Delmás for sharing their knowledge, and supporting me both academically and personally. Without your support I would have already come back to Spain.

I am sincerely thankful to all my friends, my mother, and my sister for their love and belief in me. Lastly, I would like to thank Laura for her unlimited love and confidence in me.
# Table of Contents

Acknowledgements  
Table of Contents  
List of Tables  
List of Figures

## 1 Introduction

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

## 2 Background

2.1 Big Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.1 Data Mining . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.2 Apache Spark™ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.1 Programming Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.2 Spark Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Out-of-Order Core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3.1 Performance Counters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3.2 Call Stack Traces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.4 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4.1 Performance characterization of Big Data Systems . . . . . . . . . . . . . . 16
2.4.2 Performance Characterization of Apache Spark™ . . . . . . . . . . . . . . 16
2.4.3 Profiling Tools and Methodologies . . . . . . . . . . . . . . . . . . . . . . . . 17

## 3 Methodology

3.1 Microarchitectural Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Application Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 A Top-Down Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.4 System Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
List of Tables

3.1 System Details: Our single node has two CPUs, one per socket, each CPU has 8 cores. ................................................................. 23

3.2 Algorithms and their corresponding workloads. For Spark 1.6.1 and 2.0.0 with only RDDs, 16 partitions have been used. For RDDs vs. Datasets, only 1 partition is used due to the limitation of Datasets to only 1 partition. ....................... 24

5.1 Execution time for the training stage (milliseconds) ......................... 40

6.1 Execution time for the training (milliseconds) ................................. 45

7.1 First level of Top-Down metrics including the average CPU utilization, CPI and speedup for ML API with three different JVMs, taking OpenJDK as a baseline in ML API in Spark 2.0.0. ........................................... 57

A.1 Spark 1.6.1 and 2.0.0 non-default configuration parameters. ................ 64
List of Figures

2.1 Apache Spark™ organization. The user can program in any of the shown languages, and use the different APIs (sometimes combining them, e.g., Streaming + MLlib, or Spark SQL + GraphX), or the Core API directly. The aforementioned APIs rely on the Core API. ............................................................... 5

2.2 MLlib API: As of Spark 2.0.0, the RDD-based API in the spark.mllib package is in maintenance mode. The primary Machine Learning API for Spark is now the DataFrame-based API in the spark.ml package. ............................................. 6

2.3 Frontend API, Backend computations and Data representation of RDDs (left) and Datasets (right). On the left side an RDD is created from an Array of Point objects. This RDD internally is represented as Java objects, with their corresponding metadata and pointers. On the right side, a Dataset is created from the Array, but it is stored off-heap of the JVM, and accessed using encoders/decoders using dynamically generated code from the Tungsten project. Both APIs provide the same functionality in this context, but have a different internal representation of the data. ................................................................. 10

2.4 Spark architecture. In addition to the shown edges, the Executors also communicate among them and with the SparkContext (not shown in the diagram for simplicity). ................................................................. 11

2.5 Intel® Ivy Bridge µarch simplified block diagram of a single core. The frontend fetches instructions in-order, whereas the backend executes them out-of-order. The µop queue acts as a buffer for µops that are ready to be delivered to the backend. Therefore, the star denotes where a µop arrives at issue point. .... 14

3.1 Top-Down first level breakdown flowchart. ........................................ 20

3.2 Top-Down analysis hierarchy for Naïve Bayes. .................................... 22

4.1 Spark 1.6.1 MLlib Algorithms Performance: First two levels of the Top-Down methodology. ......................................................... 26

4.2 Memory-bound breakdown for Regression Decision Trees .................... 29
4.3 Clock ticks per Core (left vertical axis, bars) and CPU utilization × CPU frequency ratio (right vertical axis, line) for Regression Decision Trees varying $maxBin$ and $maxDepth$ parameters. ............................................ 35

4.4 Top-Down first and second level breakdowns for regression decision trees showing the percentage of pipeline slots falling into each category at issue point with fixed maximum depth in the tree ($maxDepth = 5$) and variable maximum number of bins ($maxBin =$ from 32 up to 512). ............................................ 36

4.5 Top-Down first and second level breakdowns for regression decision trees showing the percentage of pipeline slots falling into each category at issue point with fixed maximum number of bins ($maxBin = 32$) and variable maximum depth ($maxDepth =$ from 5 to 30). ............................................ 37

5.1 Top-Down first and second levels breakdown per algorithm showing the percentage of pipeline slots falling into each category at issue point. Values for Spark 2.0.0 with Scala 2.10 and 2.11 have been normalized with respect to the total number of clock ticks per core in Spark 1.6.1 with Scala 2.10. ......................... 43

6.1 Top-Down second level analysis for Logistic Regression. The graph shows clock ticks per core normalized to MLlib and the corresponding breakdown of the clock ticks per category. MLlib (C) corresponds to the MLlib implementation being forcing software caching after the transformation from RDD to DataFrame. .... 46

6.2 Top-Down second level analysis for Naïve Bayes. ............................. 48

6.3 Top-Down second level analysis for Decision Trees. ..................... 49

6.4 Top-Down analysis for function $labeledPointToTreePoint$. The results for ML have been normalized with respect of the total amount of clock ticks used by this function in the MLlib implementation. .......................... 50

6.5 Top-Down second level analysis for K-means. ML corresponds to the regular implementation of ML, whereas ML (C) corresponds to the modified version from Listing 6.3. ............................................. 50

6.6 Top-Down second level analysis for Gaussian Mixture. ................. 52

6.7 Top-Down memory-bound analysis for $BinaryRegistry$::$apply$ in Gaussian Mixture. .......................................................... 52

6.8 Top-Down second level analysis for Word2Vec. ML corresponds to the regular implementation of ML, whereas ML (C) corresponds to the modified version that caches the transformation in the right place. ......................... 53

7.1 JVM speedup comparisons, normalized to OpenJDK’s performance. .... 57

7.2 SMT vs. ST speedup, normalized to 2 sockets × 4 cores/socket × 1 thread/core execution time. ......................................................... 58

7.3 Average CPU Utilization for ML API. ........................................ 59
7.4 HT Effectiveness. Values greater than or equal to 1 are desirable, meaning that HT is achieving 30% of what could be achieved by doubling the number of cores.
Chapter 1

Introduction

1.1 Motivation

Analysis of large data sets and databases utilizing machine learning and statistical algorithms is an important research area and application focus. The evolution of data centers along with the exponential growth of data, led naturally to the development of powerful platforms for such analysis. New software layers were developed, such as Hadoop [2], MapReduce [17], and Spark [65]. Such frameworks have become popular because they enable developers to utilize commodity machines for rapid application building and analysis, without requiring them to be highly skilled into the inner workings of complex systems; such analysis can take place without worrying about the complexities of parallelism and warehousing details such as data placement, partitioning and failures.

This thesis focuses on understanding and analyzing the performance of Machine Learning Algorithms (MLAs) on the Spark platform. We focus on MLAs due to their prevalence for analytical tasks in Big Data (more details in Chapter 2). We target the machine learning library (MLlib) on Spark, a widely supported library of algorithms which are considered the most prevalent and applicable. Spark has emerged as a leading platform for large scale data processing, enjoying wide industrial and user support.

MLAs and applications operate on massive data volumes [45], hence the interest by the Data Management community recently [30, 34, 45]. Understanding the performance of MLAs on data analytics platforms such as Spark is paramount as they are the building blocks of complex analytical work flows. Performance improvements on one part of such work flows may largely impact the resulting analytics pipeline and considerably improve the performance of the entire system. Recent works [6, 54] report on the importance of machine learning work flows, their optimizations and associated data management challenges in large industrial settings. Understanding the performance of such work flows and the parameters affecting them is a first step towards addressing those challenges. We focus on the training phase of algorithms targeting continuously training platforms [6]. Machine learning inference performance is also important, primarily in deep learning scenarios (e.g., [41] and references therein) but is not a focus of our
Our study aims to understand the root causes and associated performance bottlenecks when popular learning algorithms are executed on Spark over commodity servers. This is a challenging task due to the various software layers involved, both at the system level (Spark and associated run time) and the application level (Scala, Java, implementation of the algorithms, etc.). In order to address this challenge, we employ a methodology that highlights the impact of system parameters on performance and the associated reasons behind that. We adopt a microarchitectural (µarch) driven approach and use VTune [13] to study execution stack traces to guide our analysis.

This thesis focuses on single-workloads (one algorithm at a time) running on a single machine, rather than a set of algorithms in order to understand the root causes behind the bottlenecks. We consider this a required first step in order to gain enough understanding before extending the study on analytics pipelines, i.e., a sequence of algorithms applied in a pipeline, and multiple machines, as part of our future work in this area.

We begin our study by analyzing the performance of algorithms belonging to the MLib library on Spark 1.6.1. This is a required first step to establish a baseline and study the improvements brought on by Spark 2.0.0 [3, 65], explained in detail in Chapter 2.

1.2 Contributions

This thesis presents an extensive analysis of MLAs performance on Spark. At all parts of our discussion, after identifying and attributing performance bottlenecks, we explain how the performance could be potentially improved and/or which parameters should be optimized. In this work, we make the following contributions:

- We adopt a µarch approach, deploying VTune and analyzing call-stack traces in order to understand performance bottlenecks when executing MLAs on Spark. For all cases, we focus on individual functions, studying their performance and attributing the bottlenecks to specific code/functions/parameters as the case may be. We analyze the performance and bottlenecks of Spark 1.6.1 as a baseline, and then we provide a comprehensive treatment of the improvements and bottlenecks when utilizing Datasets vs. Resilient Distributed Datasets (RDDs) in Spark 2.0.0. In some cases, we demonstrate that simple code changes offer vast performance benefits.

- In Spark 1.6.1, we found that dot product is very inefficient, suffering from memory bandwidth contention caused by the overhead of Java/Scala constructs and the Non-Uniform Memory Access (NUMA) topology of our system. Garbage collection is also a problem, increasing memory bandwidth congestion. The performance of some algorithms could improve using Single-Instruction, Multiple Data (SIMD), such as vector extensions.
• We observed in Spark 2.0.0 that Datasets have a better data representation that is able to reduce memory stalls and to improve efficiency due to superior cache performance resulting from the lean data representation and the dynamically generated bytecode to access the data structures. These optimizations provide a speedup of up to 64% for Datasets over RDDs. We also show how caching a transformation of a Dataset/RDD in software before performing an action is recommended if the data is going to be reused, increasing the performance between 6%-23%.

• We present an evaluation of the impact of some system parameters, such as the choice of the Java Virtual Machine (JVM), or enabling Simultaneous Multi-Threading (SMT) for Spark 2.0.0 with Datasets and RDDs. In our experiments, Oracle's Hotspot JVM outperforms OpenJDK's and IBM's JVM by 4%-15%. In addition, SMT provides on average 13% and 18% speedup for the MLlib and the ML APIs, respectively, and up to 52% speedup.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 provides the necessary background on Big Data analytics, Apache Spark™ infrastructure, and the performance monitoring mechanisms available in the processor used in this study. Chapter 3 presents our methodology and in particular outlines the Top-Down [61] approach to performance analysis utilizing a μarch approach. Chapter 4 presents the performance evaluation of MLAs (all implemented utilizing RDDs) on Spark 1.6.1, setting a baseline to study how their performance evolved in Spark 2.0.0. Chapter 5 evaluates the performance differences between Spark 1.6.1 and Spark 2.0.0 to determine if the RDD API performance remains the same. Chapter 6 studies MLAs using Datasets in Spark 2.0.0, comparing and contrasting with their RDD implementation in Spark v2.0.0. It also details the performance advantages as well as limitations of these implementations. Chapter 7 analyzes MLAs performance while varying several system parameters of interest. Finally, Chapter 8 offers our concluding thoughts and a description of potential future work.
Chapter 2

Background

This chapter reviews the following background material: Section 2.1 discusses the general context of this research. Section 2.2 introduces Apache Spark\textsuperscript{TM} (Spark), the framework analyzed in this work, and its more relevant features. Section 2.3 reviews modern general purpose processing cores emphasizing out-of-order execution, and the more relevant techniques that are currently used to identify bottlenecks on them. Finally, Section 2.4 comments on related work.

2.1 Big Data

Big data [47, 52] refers to data sets that are so large or complex that challenge traditional data processing software. It involves any voluminous amount of structured, semi-structured and unstructured data that has the potential to be “mined” for information. Big data is often characterized by three ‘V’s: the extreme volume of data, the wide variety of data types, and the velocity at which the data must be processed.

The ability to extract and infer knowledge from these massive amounts of data is often referred to as “big data analytics”, or “data mining”, and it has a wide range of applications, for example strategic decisions in industry, health care, or government applications [11, 31, 32].

2.1.1 Data Mining

Data mining [24, 59], also known as Knowledge Discovery from Data (KDD), is the computing process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems, among others.

There are multiple tools available for data mining, e.g., Weka [23, 26], RapidMiner [28], Pentaho [8], MapReduce [17]. We focus on Spark [65] because it is recent, open-source, and a general engine for large-scale distributed processing that includes programming languages extensively used in data mining. Some of the aforementioned frameworks are too specific, or cannot handle large amounts of data in a fault-tolerant way. Therefore, Spark offers many optimization opportunities.
2.2 Apache Spark™

Spark [3, 27, 65] is an open source unified engine for large-scale distributed data processing. It provides a programming model similar to MapReduce [17] but extends it with a data-sharing abstraction, the Resilient Distributed Datasets [63] (RDDS, explained in detail in Section 2.2.1.1). With this extension, Spark can capture a wide range of processing workloads that previously needed separate engines, including SQL, machine learning, graph processing and streaming [4, 39, 60, 64], as shown in Figure 2.1. Currently, Spark provides APIs for the aforementioned processing workloads, and the programming languages supported are Java [22], Scala [42, 43], Python [58] and R [55]. These implementations use the same optimizations as specialized Big Data processing engines (such as column-oriented processing and incremental updates) but run as libraries over a common engine, the Core Application Programming Interface (API). This Core of Spark is mainly written in Scala, which runs on the top of the Java Virtual Machine [35] (JVM).

![Figure 2.1: Apache Spark™ organization. The user can program in any of the shown languages, and use the different APIs (sometimes combining them, e.g., Streaming + MLlib, or Spark SQL + GraphX), or the Core API directly. The aforementioned APIs rely on the Core API.](image)

This thesis focuses on the Machine Learning API from Figure 2.1, MLlib, which has two different implementations as shown in Figure 2.2. One is MLlib and is based on RDDS; whereas the other one is based on DataFrames [4] (also known as Datasets after Spark 2.0.0), and is commonly known as ML. This thesis studies MLlib package in all sections, because RDDs are extensively used in Spark; whereas ML package is used in addition to MLlib only in Sections 6 and 7 because Datasets/DataFrames and RDDs are compared.
MLlib API

| Spark.mllib (RDD) | Spark.ml (DataFrame) |

Figure 2.2: MLlib API: As of Spark 2.0.0, the RDD-based API in the spark.mllib package is in maintenance mode. The primary Machine Learning API for Spark is now the DataFrame-based API in the spark.ml package.

2.2.1 Programming Model

The programming model of Spark is based on the aforementioned abstraction, the RDD (Datasets offer the same abstractions). The interface with RDDs consists on high-order functions that execute user-defined functions in parallel. A detailed explanation about RDDs and these functions follow below.

2.2.1.1 Resilient Distributed Datasets

RDDs are fault-tolerant collections of objects partitioned across a cluster that can be manipulated in parallel. High-order functions are divided into two operations: transformations and actions. Users create RDDs by applying transformations to their data, and compute a result by applying actions:

- **Creation:** In order to start using transformations and actions, first we need to create an RDD. SparkContext (explained in Section 2.2.2.2) offers some methods to create an RDD, e.g., from a collection of elements, or from an input file:
  - `parallelize(seq: Seq[T]) : SparkContext => RDD[T];` Distributes a local Scala collection to form an RDD.
  - `textFile(path: String) : SparkContext => RDD[String];` Reads a text file and returns it as an RDD of Strings.

- **Transformations:** Lazy operators that create new RDD(s) from existing RDD(s), e.g:
  - `map(function: T => U) : RDD[T] => RDD[U];` returns a new distributed dataset of U formed by passing each element of type T from the source through a function.
  - `reduceByKey(function: (V, V) => V) : RDD[(K,V)] => RDD[(K,V)];` returns a new distributed dataset of (K,V) pairs where the values for each key are aggregated using a function provided by the user\(^1\).
  - There are other numerous predefined transformations, e.g., `filter`, `join`, `cartesian`, `union`, `intersection`, `distinct`.

\(^1\)The function should be commutative and associative so that it can be computed correctly in parallel.
Chapter 2. Background

- **Actions**: Issue a computation on the RDD and return a value to the *driver* program (explained in Section 2.2.2.1), or writes the result to storage, e.g.:
  - `reduce(function: (T, T) => T) : RDD[T] => T`: aggregates the RDD elements using the given function\(^1\).
  - `collect() : RDD[T] => Seq[T]`: returns all the RDD elements as a sequence at the driver.
  - `count() : RDD[T] => Long`: returns the number of RDD elements.
  - `saveAsTextFile(path: String) : RDD[T] => Unit\(^2\)`: writes the RDD elements as a text file in a given directory.
  - There are other predefined actions, for example, *first*, *take*, *countByKey*, *foreach*.

RDDs have the following properties:

- **Resilient**: RDDs are fault tolerant, if a node fails during the computation of an RDD, it is possible to follow the chain of transformations (i.e., its lineage) required to reconstruct the RDD.
- **Distributed**: Data resides in multiple nodes in a cluster.
- **Dataset**: Collection of partitioned data, e.g., tuples or other objects.
- **In-Memory**: The data inside an RDD is stored in memory (can be swapped to disk) in order to provide low access latency.
- **Immutable**, or Read-Only: RDDs do not change once created and can only be transformed using transformations that will create a new RDD.
- **Lazy evaluated**: The data inside an RDD is not available or transformed until an action is performed.
- **Cacheable**: The data can be cached in software, i.e., can be held in memory or disk to avoid recomputations.
- **Parallel**: The data is processed in parallel between multiple cores and nodes.
- **Typed**: RDD records have types, e.g., Long in RDD[Long].
- **Partitioned**: Records are partitioned, i.e., split into logical partitions and distributed across nodes.
- **Location-Stickiness**: RDDs can define placement preferences to compute partitions.

*User-defined* functions are those used inside *higher-order* functions, for example the aggregator function inside the `reduceByKey` transformation. A precise example follow below.

**An RDD Example**

In Listing 2.1 an example for RDD manipulation is shown. In line 1, a regular array of *Strings* is created, containing the words “one” once, “two” twice, and so on. The purpose of this code

\(^{1}\)In Scala, `Unit` is a subtype of `scala.AnyVal`. There is only one value of type `Unit`, `()`, and it is not represented by any object in the underlying runtime system. A method with return type `Unit` is analogous to a Java method which is declared `void`.

\(^{2}\)In Scala, `Unit` is a subtype of `scala.AnyVal`. There is only one value of type `Unit`, `()`, and it is not represented by any object in the underlying runtime system. A method with return type `Unit` is analogous to a Java method which is declared `void`.
is to count how many times each word appears in the array. Line 4 transforms this array into an RDD by calling the method `parallelize` from the `SparkContext` (sc) object. Once the `RDD[String]` is created, the `map` transformation is applied, generating a new RDD of tuples `(String, Int)` which will contain each word of the array and the number “1” per tuple. In this case the high-order function is `map`, and the user-defined function is the tuple constructor of a `(String, Int)`. Scala is able to automatically infer the types of the objects. In line 7, a second transformation is applied to the previous result. The function `reduceByKey` will apply the user-defined aggregator to the second element of the tuple, using the first element as a key. Up to this point, no computations have been triggered due to the lazy evaluation of RDDs. Finally, in line 10, the action `collect` is performed, which will trigger the computation of all the previous transformations and will give an array as a result. This array holds the final result that we were expecting, with the original words and a numerical value per word indicating how many times this word appeared. The particular transformations and the action marked in bold text in Listing 2.1 are:

- Line 4 (transformation):
  \[
  \text{map}(f: \text{String} \Rightarrow (\text{String}, \text{Int})) : \text{RDD[String]} \Rightarrow \text{RDD[(String, Int)]}
  \]
- Line 7 (transformation):
  \[
  \text{reduceByKey}(f: (\text{Int}, \text{Int}) \Rightarrow \text{Int}) : \text{RDD[(String, Int)]} \Rightarrow \text{RDD[(String, Int)]}
  \]
- Line 10 (action):
  \[
  \text{collect}() : \text{RDD[(String, Int)]} \Rightarrow \text{Array[(String, Int)]}
  \]

```
val words = Array("one", "two", "two", "three", "three", "three")

val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
wordPairsRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[1]

val wordCountsWithReduce = wordPairsRDD.reduceByKey(_ + _)
wordCountsWithReduce: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3]

val wordCountsWithReduce.collect()
res1: Array[(String, Int)] = Array((two,2), (one,1), (three,3))
```

Listing 2.1: Spark interactive console example for RDD manipulation.

2.2.1.2 DataFrames and Datasets

In this Section, we assume the reader possesses a basic understanding of relational database concepts. Spark SQL [4] introduced the `DataFrame` tabular data abstraction. Like an RDD, a DataFrame is a distributed collection of data. Unlike an RDD, this data is organized as named columns. Conceptually, it is equivalent to a relational table in a database or a data frame in R/Python. Once created, a DataFrame provides a data abstraction or Domain-Specific Language (DSL) for working with structured and semi-structured data, i.e., datasets with a
schema (the description of the structure of the data). Therefore, a DataFrame is a distributed collection of rows\(^3\) with a schema that is a result of the structured query it describes.

With a structured data representation, it is possible to enable additional store and computation optimizations that were not possible with regular RDDs. Behind the DataFrame API are two optimization elements: the Catalyst optimizer and the Tungsten execution engine.

The Catalyst optimizer contains a general library for representing expression trees (e.g., algebraic or boolean expressions) and applying rules to manipulate them. On the top of this framework, there are extra libraries specific to relational query processing [33] and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries into Java bytecode.

The Tungsten project focuses on improving the efficiency of memory and CPU. To do so, it focuses on three points:

1. Memory Management and Binary Processing. Leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection using encoders for (de)serialization, among other techniques. An example is shown in Figure 2.3.
2. Cache-Aware computation. Improving the libraries and data structures to exploit memory hierarchy.

Datasets are strongly-typed data structures similar to DataFrames, but more restrictive, in the sense that DataFrames are untyped. Starting from Spark 2.0.0, Dataset API and DataFrame API merged into the Dataset API, and a DataFrame is just an alias for a Dataset[Row].

We are interested in understanding the precise performance implications of using the RDD API versus the Dataset/DataFrame API in Spark version 2.0.0. Adopting the terminology from the Scala packages shown in Figure 2.2, we will refer to the RDD API as the MLlib package (MLlib), and to the Dataset/DataFrame API as the ML package (ML) in what follows.

### 2.2.2 Spark Architecture

Spark uses a master/slave(s) architecture. There is a driver that communicates with a single coordinator, the master, that manages workers in which executors run tasks. The driver and the executors run in their own Java processes. It is possible to run them all on the same (horizontal cluster) or separate machines (vertical cluster), or in a mixed machine configuration. Physical machines are referred to as hosts or nodes.

In this study, we use only one node in order to analyze the performance in the CPU and memory from a µarch perspective. Including more machines would move the bottlenecks from the CPU and the memory to the network and disk I/O, which is not our objective. Therefore,\(^3\) These Row objects are constructed on the fly and do not necessarily represent the internal storage format of the data, which is typically columnar.
Figure 2.3: Frontend API, Backend computations and Data representation of RDDs (left) and Datasets (right). On the left side an RDD is created from an `Array` of `Point` objects. This RDD internally is represented as Java objects, with their corresponding metadata and pointers. On the right side, a Dataset is created from the `Array`, but it is stored off-heap of the JVM, and accessed using encoders/decoders using dynamically generated code from the Tungsten project. Both APIs provide the same functionality in this context, but have a different internal representation of the data.

Running on a single node is a reasonable starting point for this analysis. Understanding individual node behavior will be a stepping stone for analyzing multi-node execution and may be applicable in that case as well. Adding more machines to the study is contemplated as future work in Chapter 8.1. Additional information regarding the Spark configuration for our study can be found in Appendix A.

### 2.2.2.1 Driver

The application driver process (driver) is a Java process that runs the `main` function and hosts the `SparkContext` for an application. It is the coordinator of jobs and tasks, using a `Directed Acyclic Graph Scheduler` (DAGScheduler) and a `TaskScheduler`, among other components. A Spark application is split by the driver into tasks that are scheduled to run on executors. It also coordinates workers and overall execution of tasks.
Figure 2.4: Spark architecture. In addition to the shown edges, the Executors also communicate among them and with the SparkContext (not shown in the diagram for simplicity).

DAGScheduler

* DAGScheduler* is the scheduling layer of Spark that implements stage-oriented scheduling. It transforms a logical execution plan, i.e., RDD lineage of dependencies built using RDD transformations, to a physical execution plan using stages. After an action has been performed, *SparkContext* hands over a logical plan to *DAGScheduler* that will translate to a set of stages that are submitted as *TaskSets* for execution.

2.2.2.2 Spark Context

*SparkContext* sets up internal services and establishes a connection with the deployment environment (local or clustered). Once a *SparkContext* is created, it is possible to create distributed entities such as RDDs, accumulators, or broadcast variables. It is also possible to access Spark services (e.g., TaskScheduler, BlockManager) and run jobs.

2.2.2.3 Master

The machine were the cluster manager runs is the master node. The resources of the cluster are allocated by the master, using the workers running throughout the cluster for the creation of the executors for the driver. After that, the driver runs tasks con the executors.

2.2.2.4 Worker

A worker is essentially any node that can run application code in the cluster. It contains one or more executors that will execute tasks. They are the compute nodes in Spark. A worker receives a serialized task that will run in a thread pool. They also host a local *BlockManager*
that serves blocks to other workers in a Spark cluster, and communicate among themselves using these BlockManager instances.

### 2.2.2.5 Executor

An executor is a process launched for an application on a worker node. It is a distributed agent that runs tasks and keeps data in memory or disk storage across the tasks. Each application has its own executors that usually run for the entire lifetime of a Spark application. Executors report heartbeat and partial metrics for active tasks.

### 2.2.2.6 Task

A task (also referred to as command) is the minimum unit of work that will be sent to one executor to compute an RDD partition. There are two particular implementations of a Task: ShuffleMapTask, which executes a task and divides the task’s output into multiple buckets; and the ResultTask, which executes a task and sends the task’s output back to the driver application.

### 2.2.2.7 Job

A job is a parallel computation consisting of multiple tasks that is submitted to the DAGScheduler in response to a Spark action (e.g., save, collect). Computing a job is equivalent to computing the partitions of the RDD the action has been executed upon. A job starts with a single target RDD, but it can include other RDDs that are all part of the target RDD’s lineage.

### 2.2.2.8 Stage

Each job gets divided into smaller sets of parallel tasks, the stages, which depend on each other. Each stage is a physical unit of execution, i.e., a step in a physical execution plan. In other words, a Spark job is a computation sliced into stages. A stage can only work on the partitions of a single RDD, but can be associated with many other dependent parent stages, with the boundary of a stage marked by shuffle dependencies.

### 2.3 Out-of-Order Core

For this Section, we assume the reader is familiar with out-of-order cores and their characteristics. Modern high-performance processors integrate multiple cores in the same chip [18]. These cores implement several techniques in order to keep their execution pipelines busy. Some of these techniques include a large out-of-order execution window, speculative execution, or hardware prefetching. In these processors is difficult to identify the true performance limiters due to the amount of nonsequential issues they can tolerate. Various techniques interact in multiple non-obvious ways, moreover, as many operations proceed in parallel, the bottlenecks may be due to multiple overlapping causes.
Figure 2.5 shows a simplified block diagram of the out-of-order pipeline present in the processor used in this study. This pipeline is divided into two main sections: the frontend and the backend. The frontend fetches instructions from memory and translates them into micro-operations (µops), i.e., simple low-level instructions that together conform the Complex Instruction Set Computing (CISC) of the Intel®64 architecture (the majority of commodity computers used nowadays for Big Data processing are based in this instruction set). These µops are fed to the backend in program order (in-order). The backend schedules, executes and completes the µops in an order different than that specified by the program (i.e., out-of-order), and finally commits (retires) them in program order (i.e., in-order).

The following sections review some of the mechanisms that aid in identifying performance bottlenecks in out-of-order processors, namely the use of performance counters and call stack traces.

2.3.1 Performance Counters

In order to identify performance issues in out-of-order processors, different parts of the processor are instrumented with performance counters, also known as the Performance Monitoring Unit (PMU). They consist on a set of measurable events that may be relevant to identify performance issues, such as the number of cache misses; and a set of counters to store how many times a particular event happened.

Usually the PMU consists on two types of Model-Specific Registers (MSRs): the performance event select registers and the Performance Monitoring Counters (PMCs). Measuring a performance event requires programming the performance event select registers, and the performance events are counted in the PMCs. Some PMCs are for general purpose, and can be used to measure different events, whereas a few counters can measure only a specific event. Furthermore, some architectures provide global control registers which can be used to control all, or a group of control registers or counters.

In our processor (refer to Section 3.4 for more details) there are two types of performance measurement: counting, for obtaining aggregate counts of occurrences of specific events; and Event Based Sampling (EBS), for determining the frequencies of event occurrences produced by program locations at the functions, basic block, or instruction levels.

Chapter 3, Methodology, explains in more detail how we used our processor’s PMU to identify performance issues in the analyzed algorithms.

2.3.1.1 Counting

The two key challenges with this performance measurement infrastructure are: 1) if the number of events to be monitored are more than the total number of counters provided by the processor, and 2) if two different events to be monitored are measured by the same digital logic present in the processor.
A simple solution to address the aforementioned problems is multiplexing, although it also has its limitations.

In the first case, as the number of counters is less than the number of events to be monitored, a solution is time division multiplexing. In other words, one event does not get a dedicated event counter for the entire duration of the measurement. Instead, the events are measured in small periods multiple times during the entire measuring duration. At the end of the measurement duration the actual measurement period is also recorded and the aggregated event count is

![Diagram](image)

Figure 2.5: Intel® Ivy Bridge μarch simplified block diagram of a single core. The frontend fetches instructions in-order, whereas the backend executes them out-of-order. The μop queue acts as a buffer for μops that are ready to be delivered to the backend. Therefore, the star denotes where a μop arrives at issue point.
scaled to the complete measurement period.

In the second case, the same technique can be applied. The only difference is that instead of multiplexing a PMC, the digital logic unit is time multiplexed to measure different events.

Even though time division multiplexing solves some of the aforementioned issues, it has some limitations. The scaled values may not be accurate enough, e.g., it may happen that the event which was not being measured for a particular instance may have spiked or tanked during that instance and the scaled value will be misleading. A possible solution is to run the application multiple times, each time selecting a different event for the counter. If the variance across runs is small enough, the results can be considered representative.

2.3.1.2 Event Based Sampling

In EBS, the PMU’s counters are configured to overflow after a preset number of events, and when it overflows the information is recorded by capturing the data of the instruction pointer, and some registers by raising an interrupt. The limitations of this method are the sampling delay, and the speculative count.

Sampling delay occurs when there is some delay between the counter overflow and the time when the interrupt is raised. Combined with the long pipelines and superscalar datapaths of the core, the program counter data stored at the sampling time may be not the event that caused the counter to overflow.

Speculative count occurs because some instructions have executed speculatively and may not commit to the architectural state of the machine if, for example, the followed branch was wrong. These instructions may cause events and contribute to the event count even if they do not complete.

As in the counting method, there are also problems with the number of available resources to keep track of the events, and the applied solution is the same, time division multiplexing.

2.3.2 Call Stack Traces

Ultimately, we would like to associate bottlenecks with the program structures and behavior that caused them. Having only a set of performance events may be useful in some cases, however as the complexity of the software increases, most of the time it is insufficient. Call stacks are used to keep track of the point where each active subroutine should return control when it finishes execution. Therefore, a call stack trace is the sequence of subroutine calls performed during a program execution. Combined with performance counters, call stack traces provide information about the performance events caused per function, as opposed to having these events just for the entire program execution without an event breakdown per function.
2.4 Related Work

Big Data analytics is an active research area that exhibits very different behavior depending on three main different components: variety, velocity and volume [47, 48]. Big Data has the potential to provide insights that can transform every business, and it has generated a whole industry of supporting architectures such as MapReduce [17]. The infrastructure required for Big Data analytics is so big that multiple disciplines coexist in order to make it possible, e.g., analytics engines, warehouse infrastructure, security. In this study we characterized the performance of a particular general engine, Apache Spark™ [3, 27, 65].

2.4.1 Performance characterization of Big Data Systems

There are multiple performance characterization works related with Big Data systems. For example, Ailamaki et al. [1] characterized the performance bottlenecks of four commercial Database Management Systems (DBMS). Mishra and Koudas [40] designed a query monitoring system that enables profiling and performance analysis of data management workloads. Ghoting et al. [21] characterized the performance and memory access behavior of several data mining algorithms, determining that data mining algorithms are memory and compute intensive. Our study corroborates their findings, with additional insights. We found additional overhead in the data structures and garbage collection derived from the Java Virtual Machine. Furthermore, the numerous data abstractions and complex structures turn the applications more memory-bound rather than compute bound. Sirin et al. [50] used a microarchitectural approach to characterize database Online Transaction Processing (OLTP) performance. Extending such ideas on cloud-based platforms and popular analytical workloads is a natural step. Sridharan and Patel [51] presented an evaluation of workloads written on the popular R system on a contemporary processor [56] and cataloged where time goes. They found that processor stalls and garbage collection overheads are the main reasons behind performance inefficiencies. In their evaluation they use estimates [15] of the execution stages. In principle, their study could be repeated utilizing the trace-driven approach we adopt herein in order to contrast their results and findings. Finally, Kanev et al. [29] present a scale-out workloads profiling at the microarchitectural level utilizing 20,000 machines on live workloads collected over a year, revealing some opportunities for hardware and software optimizations.

2.4.2 Performance Characterization of Apache Spark™

Our study focuses on Apache Spark™, a general engine for distributed computation. Awan et al. [5] recently (2016) quantify the performance of various packages in Spark at the microarchitectural level, including K-means, Gaussian Mixture, Naïve Bayes, Support Vector Machines, and Logistic Regression from MLlib on Spark 1.5.0 with Java 7. Their work however does not utilize a trace driven approach to understand the specific reasons behind performance degradation or improvements. Our work dives deeper into the execution following the Top-Down
methodology, revealing for each algorithm the precise performance hurdles, code inefficiencies and impact of Spark design choices.

2.4.3 Profiling Tools and Methodologies

Performance characterization is a hard task that lacks of a default methodology. Big Data analytics in general, and Spark in particular bring additional challenges to traditional performance characterization. Our first approach was simulating an entire system relying on virtualization techniques using QTrace [57], from Tong et al., or PinOS [9], from Bungale et al., and collecting all kind of events from the operating system and applications. However, the size of our workloads and the instrumenting of all the relevant parts of the system would take an unfeasible amount of time, and the fact of virtualizing the entire system would incur extra overheads. Furthermore, Spark runs on the top of the Java Virtual Machine, therefore dynamic code generation and automatic garbage collection are two issues to keep in mind.

We explored alternatives that do not rely on virtualization and have low overhead, such as Linux Perf [44] or Intel® VTune [13], using the latter in our study. VTune utilizes the Top-Down methodology (detailed in Chapter 3) introduced by Yasin et al. [61], and is now part of the Intel®64 and IA-32 architectures optimization reference manual [14]. This tool has low overhead and uses the performance counters and call stack traces explained in detail previously in this chapter, allowing to identify the performance bottlenecks at the microarchitectural level, and the functions that originated them. This methodology has been used by Yasin et al. [62] to analyze the performance of CloudSuite [20], a CPU centric benchmark.

Finally, we explored the performance impact of Simultaneous Multi-Threading (SMT) in our analysis. Marr et al. [38] measured the impact of SMT on a Pentium 4 processor, showing speedups of up to 30% on a set of common server application benchmarks. In our study, using SMT brings up to 52% speedup, and on average the speedup is between 13% and 18%.
Chapter 3

Methodology

In this chapter, we present the methodology followed to identify the performance issues of the analyzed algorithms, as well as the machine and workloads used.

Building a Machine Learning (ML) model typically proceeds in three steps: firstly, loading the data, which is split between training and validation data. The validation data is not used during the training, but during the validation phase to test how accurate is the generated model. Secondly, a training phase utilizing the data read in memory. This phase generates a trained model that would be used to perform inference. Finally, a validation phase of the model, where the data that was not used during training with known outcomes is tested against the model. The focus of this study is on benchmarking the training phase of various Machine Learning Algorithms (MLAs). Depending on the ML model, the algorithms for training are fairly involved and time consuming. Any insights to optimize the algorithms’ training phase will yield vast benefits in iterative learning scenarios [6].

The analysis adopted in this paper consists of a two-fold process inspired by Yasin et al. [61, 62], consisting of microarchitectural and application levels. For example, the percentage of cache misses, or the data structures utilized in the algorithms, are sample issues at each level, respectively. The execution of a workload involves multiple software layers, namely the operating system, Java Virtual Machine (JVM) implementation, Spark framework, Scala version, and machine learning algorithms (the specific algorithms and the associated data structures utilized in the programming language they are implemented). The analysis process we adopt is customized towards identifying performance inefficiencies and attributing these inefficiencies to specific code constructs. Observations at the microarchitectural level are utilized to gain insights at the application level. In addition, we examine sensitivity to system parameters such as the JVM implementation, or single- versus multi-threaded execution, and how they impact algorithm performance.

The rest of this chapter is organized as follows. Section 3.1 describes the microarchitectural analysis level. Section 3.2 explains the application analysis level. Section 3.3 provides a practical example of the Top-Down methodology applied in this study. Finally, Section 3.4 shows the system configuration parameters, machine, and workloads used.
3.1 Microarchitectural Analysis

The microarchitectural (μarch) level further identifies and analyzes inefficiencies due to interactions among run-time, architecture, and μarch at the lowest level. Intel® VTune Amplifier XE 2016 [13] (VTune) is the main tool used for profiling the analyzed algorithms (configuration details are described in Appendix B). VTune performs hardware event based sampling as well as generating detailed call stack traces associated with the event samples; these can provide detailed understanding of the performance at all abstraction levels, enabling the Top-Down Method for Performance Analysis and Counters Architecture (TMAM or Top-Down) [61]. Top-Down enables a hierarchical approach for identifying critical bottlenecks in processors with non-trivial features, such as out of order execution, superscalar, and speculative execution. Top-Down establishes an iterative approach in which execution is divided into a number of high-level events which are progressively refined in a drill-down fashion to reveal other important performance events at a lower level. That way, a principled methodology is established that yields a trace-driven approach to explain performance.

For the experiments performed in this study, it is imperative to ensure that the particular runs are representative, namely that the events sampled represent true execution trends as reported by the Top-Down methodology. Thus, for each algorithm, the default sampling parameters from VTune’s configuration have been tested, to assure that the associated Top-Down categorization reflects the true algorithm behavior: for each algorithm we tuned the sampling interval for the data set used as input and compare the resulting Top-Down categorization with the results reported when using the default sampling intervals; if the percentage difference in each category when comparing the two Top-Down categorizations is within ±5%, the default sampling parameters are sufficient as further sampling yields small differences.

We identified that in most cases the default sampling parameters in VTune, when used for each algorithm along the size of datasets utilized as input, were sufficient to capture the events accurately. In the cases that this was not true (Decision Trees, SVD), we increase the dataset size, so that with the default sampling rate we can capture more events until the maximum measurement variance remains within ±5%. An alternative is increasing the sampling rate, but that increases the measurement overhead. For greater confidence, each experiment is run multiple times, and VTune is able to multiplex the performance counters among runs to get an accurate representation of the algorithm’s behavior, as explained in Section 2.3.1. In all cases, unless otherwise stated, each version of Spark utilized has the default configuration parameters and all the algorithms are executed with their default configuration parameters available in the MLlib/ML distributions.

Top-Down classifies μops. This classification is done at issue point, denoted in Figure 2.5 by a star, which is the point where the μops go from the in-order frontend to the out-of-order backend. At this point, the μops (which correspond to a pipeline slot) are classified into one of the four base categories that comprise the first level of Top-Down as shown in Figure 3.1.
The first and second level categories are:

1. **Frontend Bound.** Denotes the case when the frontend of the core is under-supplying the backend. The pipeline can issue up to 4 μops per cycle, if the frontend delivers less than 4 μops in a cycle, then these empty pipeline slots would count as frontend bound.
   
   1.1. **Fetch Latency**, e.g., an instruction cache miss, Instruction Translation Lookaside Buffer (ITLB) misses or fetch stalls after a branch misprediction.  
   1.2. **Fetch Bandwidth**, e.g., inefficiency in the instruction decoders, or code restrictions for caching in the Data Stream Buffer (DSB, or Decoded Instruction Cache).

2. **Bad speculation.** Pipeline slots wasted due to incorrect branch prediction.
   
   2.1. **Branch Mispredicts**, e.g., an instruction executed speculatively that is wrong.  
   2.2. **Machine Clears**, when the entire pipeline is flushed and restarted after the last retired instruction, e.g., memory ordering violations, self-modifying code, and certain loads to illegal address ranges.

3. **Retiring (or General Retirement).** μops that eventually get retired. This is ideally the category were all the clock ticks should go because it represents μops that were issued timely and executed useful work.
   
   3.1. **BASE**, regular retiring, excluding μops fetched by the Microcode Sequencer.  
   3.2. **Microcode Sequencer**, pipeline slots where the CPU was retiring μops fetched by the Microcode Sequencer (MS) ROM, e.g., μcode sequences such as Floating Point (FP) assists.

4. **Backend Bound.** No μops are being delivered at the issue pipeline because the backend of the core has no resources available to accept them.
   
   4.1. **Core Bound**, how many core non-memory issues were the bottleneck, e.g., sub-optimal execution ports utilization, or FP-chained long-latency arithmetic operations.  
   4.2. **Memory Bound**, how the memory subsystem affects performance, e.g., a load missing all the caches, or stores implying back-pressure on the pipeline.
3.2 Application Analysis

Application level analysis maps the microarchitectural performance data collected back to source code when possible. Deep software layers and the heavy use of third-party packages in modern platforms are a source of many inefficiencies which are hard to detect due to heavy software nesting. Various profilers utilize Event-Based Sampling (EBS) and call stack traces to associate performance counters back to code [10, 44], whereas Intel® VTune [13], in addition to the aforementioned techniques, uses the Top-Down methodology to identify critical bottlenecks.

3.3 A Top-Down Example

Figure 3.2 presents an example Top-Down’s methodology results (using Naïve Bayes’ algorithm) that addresses stall overlapping, when many units work in parallel; superscalar inaccuracies, because a core can issue, execute, and retire multiple instructions in a single cycle; speculative execution, when a core executes an incorrect control path; and variable miss penalties to name some of the key challenges that preceding methods (e.g., [20]) faced. The categories highlighted by VTune appear in bold text in Figure 3.2. Applying this classification in the training phase of Naïve Bayes on a specific data set will yield 23.1% of the pipeline slots at issue point classified as Frontend Bound. The algorithm’s performance is limited by instruction cache misses at the core’s frontend due to the code that iterates over the data and checks some conditions, not fitting in the L1 cache; it is memory bound in the core’s backend with 41.6% of the pipeline slots falling into this category, also due to extensive iteration over data without any expensive computation between them. Drilling down in this category, we can observe increased clock ticks in three subcategories shown in bold in Figure 3.2: L1 Bound (11.0%), L3 Bound (11.0%), and DRAM Bound or External Memory Bound (23.3%).

3.4 System Parameters

The system parameters in our study include different distributions of the JVM and two different versions of Spark. The default parameters for the JVM and Spark have been used unless stated otherwise, the complete set of configuration parameters for Spark can be found in Appendix A. Furthermore, the JVM implementation used for the algorithms is OpenJDK for all of them except the specific system analysis in Section 7, where other non-open source JVMs have been explored. This is interesting because the analyzed JVMs are used extensively, and even though two out of three compared versions are non-open source, they can be used for free. In addition, modern processors include various configurable features such as Simultaneous Multithreading (SMT), also called Hyper-Threading (HT), TurboBoost, or different data prefetchers that can be enabled or disabled. In this study unless stated otherwise, Hyper-Threading and Turbo Boost are disabled, and all prefetchers are enabled in order to isolate the behavior of the algorithm from other system processes. Disabling HT allows us to clearly identify when other processes are
running because the execution of these processes would never overlap within the same core. This allows us to judge whether they represent a significant source of interference. In addition some performance counters are shared between threads inside the same core, therefore one thread would interfere with the measurements of the other. TurboBoost would artificially inflate the speed of a single core and would hide possible performance issues that can be addressed by increasing the parallelism of the application. Leaving TurboBoost disabled will let us identify these cases easily. Moreover, it would influence the measurements as the processor’s frequency would fluctuate in response to thermal limits. It would be interesting to see how TurboBoost or disabling the data prefetchers influence the algorithm’s performance, but we decided to leave this analysis for future work.

<table>
<thead>
<tr>
<th>Frontend bound</th>
<th>23.1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad speculation</td>
<td>4.7%</td>
</tr>
<tr>
<td>Retiring</td>
<td>23.3%</td>
</tr>
<tr>
<td>Backend bound</td>
<td>48.9%</td>
</tr>
<tr>
<td>Fetch latency</td>
<td>15.4%</td>
</tr>
<tr>
<td>Fetch bandwidth</td>
<td>7.7%</td>
</tr>
<tr>
<td>Core bound</td>
<td>7.3%</td>
</tr>
<tr>
<td>Memory bound</td>
<td>41.6%</td>
</tr>
<tr>
<td>iTLB miss</td>
<td>8.5%</td>
</tr>
<tr>
<td>iCache miss</td>
<td>0.7%</td>
</tr>
<tr>
<td>Branch mispredictions</td>
<td>2.5%</td>
</tr>
<tr>
<td>Store Bound</td>
<td>11.0%</td>
</tr>
<tr>
<td>Load Bound</td>
<td>11.0%</td>
</tr>
<tr>
<td>Store Bound</td>
<td>9.0%</td>
</tr>
<tr>
<td>Load Bound</td>
<td>0.0%</td>
</tr>
<tr>
<td>Ext. Memory Bound</td>
<td>23.3%</td>
</tr>
<tr>
<td>MEM Bandwidth</td>
<td>78.2%</td>
</tr>
<tr>
<td>MEM Latency</td>
<td>21.8%</td>
</tr>
</tbody>
</table>

Figure 3.2: Top-Down analysis hierarchy for Naïve Bayes.
3.4.1 System Details

As stated in Section 2.2.2, we decided to use only one node in order to analyze the performance in the CPU and memory from a μarch perspective. Including more machines would translate the bottlenecks from the CPU and memory to the network and disk I/O, which is not our objective. Table 3.1 presents the node configuration that has been used in the experiments. The node is part of a Dell PowerEdge C6220 II rack server, which is a representative machine of a commodity cluster.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Name</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel® Xeon E5-2650 v2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ivy Bridge μarch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cores / Socket</td>
<td>8 @ 2.6 GHz (wo. Turbo)</td>
</tr>
<tr>
<td></td>
<td>Threads / Core</td>
<td>2 w. HT, 1 wo. HT</td>
</tr>
<tr>
<td></td>
<td>L1 Cache (pr.)</td>
<td>32KB Ins. + 32KB Data</td>
</tr>
<tr>
<td></td>
<td>L2 Cache (pr.)</td>
<td>256KB</td>
</tr>
<tr>
<td></td>
<td>L3 Cache (sh.)</td>
<td>20MB</td>
</tr>
<tr>
<td>Memory</td>
<td>6 × 8GB (48GB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 DDR3 channels @ 1,333 MHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Max BW 31.992 GB/s</td>
<td></td>
</tr>
<tr>
<td>HDD</td>
<td>ST1000NM0033-9ZM173</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1TB @ 7,200 rpm</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>CentOS 7.2.1511 (Core)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Linux Kernel 3.10.0</td>
<td></td>
</tr>
<tr>
<td>JVM</td>
<td>OpenJDK 64-Bit Server VM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(build 25.91-b14, mixed mode, gcc 5.4.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Oracle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Java HotSpot(TM) 64-Bit Server VM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(build 25.91-b14, mixed mode, gcc 4.3.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IBM J9 VM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(build 2.8, JRE 1.8.0 amd64-64, gcc 4.4.7)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scala</td>
<td>2.10.5 and 2.11.8</td>
</tr>
<tr>
<td></td>
<td>Apache Spark</td>
<td>1.6.1 and 2.0.0</td>
</tr>
</tbody>
</table>

Table 3.1: System Details: Our single node has two CPUs, one per socket, each CPU has 8 cores.
3.4.2 Workloads

The workloads used in order to train the algorithms have been generated using the same method as spark-perf [16], which generates synthetic datasets on the fly, except for Alternating Least Squares where we used a real dataset [25]. We modified spark-perf to make datasets persistent across runs by saving the data in disk. Since by choice we execute on a single node, we modify the scale factor so that the datasets fit in memory in order to avoid disk swapping. The workloads used for each algorithm and their respective sizes are shown in Table 3.2.

<table>
<thead>
<tr>
<th>Category</th>
<th>Algorithm</th>
<th>Size (text)</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification and Regression</td>
<td>Linear Support Vector Machines</td>
<td>21 GB</td>
<td>10M points × 100 features/point, 2 classes with 5M points per class</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>21 GB</td>
<td>10M points × 100 features/point, 2 classes with 5M points per class</td>
</tr>
<tr>
<td></td>
<td>Linear Regression</td>
<td>22 GB</td>
<td>10M points × 100 features/point</td>
</tr>
<tr>
<td></td>
<td>Naïve Bayes</td>
<td>21 GB</td>
<td>10M points × 100 features/point, 10 classes with 1M points per class</td>
</tr>
<tr>
<td></td>
<td>Decision Trees</td>
<td>21 GB</td>
<td>10M points × 100 features/point</td>
</tr>
<tr>
<td>Collaborative Filtering</td>
<td>Alternating Least Squares</td>
<td>509 MB</td>
<td>20,000,263 ratings, 465,564 tags, 27,278 movies from 138,493 users [25]</td>
</tr>
<tr>
<td>Clustering</td>
<td>K-means</td>
<td>1.4 GB</td>
<td>30M points (2 clusters, 2 dimensions)</td>
</tr>
<tr>
<td></td>
<td>Gaussian Mixture</td>
<td>451 MB</td>
<td>10M points (2 clusters, 2 dimensions)</td>
</tr>
<tr>
<td>Dimensionality Reduction</td>
<td>Singular Value Decomposition</td>
<td>19 GB</td>
<td>20M points (3 clusters, 50 dim.)</td>
</tr>
<tr>
<td></td>
<td>Principal Component Analysis</td>
<td>19 GB</td>
<td>20M points (3 clusters, 50 dim.)</td>
</tr>
<tr>
<td>Feature Extraction</td>
<td>Word2Vec</td>
<td>96 MB</td>
<td>17M words</td>
</tr>
<tr>
<td>Frequent Pattern Mining</td>
<td>FP-growth</td>
<td>662 MB</td>
<td>30M lines, dict. size 26, maximum 10 items/transaction, minimum support level 3</td>
</tr>
</tbody>
</table>

Table 3.2: Algorithms and their corresponding workloads. For Spark 1.6.1 and 2.0.0 with only RDDs, 16 partitions have been used. For RDDs vs. Datasets, only 1 partition is used due to the limitation of Datasets to only 1 partition.
Chapter 4

Analysis of MLlib performance on Spark 1.6.1

This chapter presents an application of the methodology followed in this thesis for the analyzed algorithms in Spark 1.6.1. Understanding their performance in Spark 1.6.1 establishes a baseline to study further advancements.

Figure 4.1 presents the results of the first (Figure 4.1a) and second (Figure 4.1b) levels of Top-Down for the analyzed MLlib algorithms. Each category classifies the percentage of overall execution clock ticks. The algorithms can be grouped in natural categories such as:

- Classification & Regression
  - Support Vector Machines (SVM)
  - Logistic Regression (LogRegr)
  - Linear Regression (LinRegr)
  - Naïve Bayes (NaivBay)
  - Decision Trees (DecTree)
- Collaborative Filtering
  - Alternating Least Squares (ALS)
- Clustering
  - K-means
  - Gaussian Mixture (GaussMix)
- Dimensionality Reduction
  - Singular Value Decomposition (SVD)
  - Principal Component Analysis (PCA)
- Feature Extraction
  - Word2Vec
- Frequent Pattern Mining
  - FP-growth

Figure 4.1a presents in the vertical right axis the Instructions Per Cycle (IPC) for each
algorithm (dashed line). Since each core in our machine is a 4-way superscalar, the maximum achievable IPC is 4.0.

The rest of this chapter is organized as follows. Sections 4.1 to 4.6 analyze where most of the clock ticks go for each of the categories aforementioned, providing the functions responsible for that and how they are limited from a \( \mu \)arch perspective. In addition, optimization hints are given. Section 4.7 shows an example of using Top-Down to measure the impact of varying some algorithm parameters in the \( \mu \)arch, and how they influence performance as they scale up. Finally, Section 4.8 summarizes the findings in the previous sections, and the possible optimizations.

Figure 4.1: Spark 1.6.1 MLlib Algorithms Performance: First two levels of the Top-Down methodology.

### 4.1 Classification and Regression

Classification algorithms are used mainly to identify group membership, whereas regression algorithms involve estimating or predicting a response. Some of these algorithms involve mathematical method to compute the derivatives (or gradients) in order to converge to a solu-
Chapter 4. Analysis of MLlib performance on Spark 1.6.1

An example is the *dot* product, used in SVM, Logistic Regression and Linear Regression algorithms for the gradient descent method. Other algorithms, such as Naïve Bayes, use faster operations that do not require the computation of gradients, like *AXPY*, a vector-vector operation that involves a scalar-vector product and a vector-vector addition ($y = a \cdot x + y$). Finally, decision trees do not require extensive mathematical operations. However, the complexity in this algorithm comes from building the tree itself, finding the correct bins to split the tree and classifying the points, so the computational cost comes from traversing and resizing the tree rather than performing mathematical operations.

For some algorithms such as Logistic Regression, it is possible to use alternative training methods like Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) instead of gradient descent based methods. SVM, LogRegr and LinRegr algorithms analyzed in this chapter use gradient descent, which is mainly dominated by the computation of *dot* products.

### 4.1.1 Support Vector Machines

Starting from Support Vector Machines (SVM), we can observe that the algorithm is backend-bound and, in particular, memory-bound. To understand why, an examination of the stack trace reveals that the function where most of the clock ticks are spent in this algorithm is the *dot* function (performing dot products). In particular, 57% of the total clock ticks and 40% of the total instructions retired are in this function, with a CPI of 1.85 (overall algorithm CPI is 1.295). For comparison, the second function in terms of clock ticks corresponds to 2.7% of the total. The *dot* function is 86% backend-bound. Out of this portion, 78% corresponds to memory-bound, especially DRAM bandwidth-bound. The entire behavior of the algorithm is explained by the fact that the *dot* function is the main bottleneck waiting for memory to supply data. Anything one can do to assist the performance of this function (e.g., improving data locality in order to reduce memory bandwidth consumption) would make the algorithm faster.

### 4.1.2 Logistic Regression

Logistic Regression (LogRegr) follows a similar pattern as SVM; it is backend-bound, but appears less memory-bound and slightly more core-bound compared to SVM. An examination of the stack trace reveals that the function *dot* accounts for 45% of the total clock ticks, dominating the algorithm execution time and contributing to the entire algorithm appearing as backend- and memory-bound. However, the second most costly function is *LogisticGradient* (computing gradients), which is core-bound and accounts for 17% of the total clock ticks, contributing to a higher overall percentage in the core-bound category for the entire algorithm. The computation of gradients for a logistic function is more expensive than the computation of gradients for a linear function.
4.1.3 Linear Regression

Linear Regression (LinRegr) follows a very similar pattern and is mostly backend- and memory-bound due to the dot function taking most of the clock ticks (50% of the total clock ticks). Overall, for SVM, LogRegr and LinRegr, the main bottleneck is the computation of inner products, which is highly memory bandwidth-bound and is using floating-point scalar (non-vectorized) operations. A better data layout that improves data locality would help to reduce the impact of memory bandwidth, and the use of Single-Instruction, Multiple Data (SIMD) instructions such as vector extensions would also help with the floating point scalar overhead in the execution units inside the cores.

4.1.4 Naïve Bayes

Naïve Bayes (NaivBay) appears mostly backend-bound, and in particular memory-bound. In addition, the algorithm has a higher frontend latency with respect to the previous regression algorithms. An analysis of the stack trace reveals that most of the time is spent in function prefixLength; this is a high-level Scala construct to iterate over array values and return the longest prefix of values that satisfies a condition (in this case checking for the feature values to be non-zero). This function is mostly memory-bound and DRAM-bound. This immediately reveals an implementation inefficiency. The function is called multiple times in the MLlib implementation and this is redundant, checking for non-negative values should be done only once. If we remove the unnecessary calls, the bottleneck becomes function changeValue from SizeTrackingAppendOnlyMap, which is called inside the function insertAll from the class ExternalSorter, which is a Scala function to insert values in an array after they are modified. This function is mostly memory-bound as well. Overall for this algorithm, since the computations involved are simple, such as the aforementioned AXPY method, most of the overhead comes from manipulating data structures that handle the data, derived from utilizing inefficient Scala language constructs.

4.1.5 Decision Trees

Decision Trees (DecTree) (the particular implementation is Regression Decision Trees) is mainly backend-bound, and specifically, memory-bound. However, this algorithm has a considerable amount of clock ticks falling under the bad speculation category. A closer inspection of the function with the highest number of clock ticks, findBin, reveals that it accounts for 30% of the total clock ticks (the function identifies which bin to split by examining information measurements on the sub tree at each node level during the decision tree construction). A closer examination of the memory-bound category (at level three of Top-Down), shown in Figure 4.2, reveals that the algorithm is heavily L3-bound. The reason is that the tree traversal takes place in parallel among the cores and the L3 cache is the closest memory that is shared among the cores. Thus, the main bottleneck here is L3 cache latency which is increased if the memory block resides in a different socket due to its Non-Uniform Memory Access (NUMA) topology.
4.2 Collaborative Filtering

Collaborative Filtering (CF) is a technique used by recommendation systems [46]. It is the process of filtering for information or patterns using techniques that involve collaboration among multiple agents, for example using a database of preferences for items by users to predict additional topics or products that a new user might like. The analyzed CF algorithm in this thesis is Alternating Least Squares (ALS), given that it was the only one implemented in the MLlib API for Spark at the time of this research.

4.2.1 Alternating Least Squares

Alternating Least Squares algorithm is mainly backend-bound (48%), and has a similar distribution between memory and core-bound (26% and 21%, respectively). This algorithm has to compute a set of latent factors that can be used to predict missing entries, and the main functions involved in this task are DSPR, which performs a rank-1 update of a double precision symmetric packed matrix \(A = \alpha \cdot xx^T + A\), DAXPY, which performs a double precision vector-vector operation that involves a scalar-vector product and a vector-vector addition \(y = a \cdot x + y\), and this particular implementation uses the TimSort [7] sorting algorithm for generic collections. The function with the highest number of clock ticks is DSPR (15% of the total clock ticks), and it has 87% of its clock ticks falling under the retirement category, with a CPI of 0.293. This is an indicator of good performance, however, this function is using scalar floating-point instructions, and the performance could be improved by vectorization. The function DAXPY has a similar behavior, even though it only accounts for 2% of the total clock ticks. The second function with the highest number of clock ticks is the apply method inside an anonymous Scala method in ml.recommendation.ALS class, with 11% of the total clock ticks. This function is memory-bound. The methods related with the TimSort algorithm are also memory-bound, and account for 9% of the total clock ticks. In order to improve the performance of TimSort,
the data should have a lean representation, without the characteristic overhead of Java/Scala
constructs, thus fitting more data in the same cache line and reducing the amount of DRAM
accesses.

4.3 Clustering

Clustering is an unsupervised learning problem that consists on grouping a set of elements in such
a way that elements in the same group (or cluster) are more similar to each other of those in other
clusters. The algorithms analyzed in this study are K-means and Gaussian Mixture (GaussMix).
Both work on an Euclidean space, and have to compute the distance between points. The main
mathematical operator in this case is the dot product (single- or double-precision), however,
there is a huge overhead accessing the data structures in both implementations.

4.3.1 K-means

K-means appears heavily backend-bound and, in particular, memory- (48%) and core-bound
(17%) . The function with the highest percentage of clock ticks is dot, applied on the normalized
input vectors to assess distance in an Euclidean space (with almost 14% of the total clock ticks);
it is heavily memory-bound (80% of its clock ticks), contributing to the memory-bound behavior
of the entire execution. There is a bottleneck in the function hasNext from Iterator, a Scala
implementation of generic data types that accounts for 7% of the total clock ticks, and is
memory-bound (and more precisely L1-bound, Store-bound and DRAM-bound). It returns a
boolean value based on the state of the iterator object, thus generating a large number of boolean
values and causing memory stalls due to memory bandwidth and the store queue saturation. There is also a noticeable overhead due to garbage collection, mainly caused by the function
copy_to_survivor_space from the PSPromotionManager class. This function accounts for 7%
of the total clock ticks and is also highly memory-bound, and is the responsible of copying the
objects from one survivor space to the other.

The efficiency of this algorithm could be improved with a better data layout that maximizes
data locality in order to reduce memory bandwidth consumption. In addition, using an iterator
that returns a boolean value seems to be a bad idea since it saturates the store queues in the
backend of the cores. Finally, garbage collection is a problem, especially the process of copying
objects between survivor spaces, a fine grain tuning of the Java Virtual Machine generations’
size could improve the performance in this cases.

4.3.2 Gaussian Mixture

Gaussian Mixture (GaussMix) appears heavily backend-bound but also memory and frontend
latency-bound. The function with the highest percentage of total clock ticks, 42%, is the apply
method from the BinaryRegistry class, an operator from the Breeze linear algebra package,
which is used for manipulating large sparse/dense vectors utilized heavily in the algorithm.
This function is principally memory and core-bound, with 33% and 24% of its own clock ticks falling into each category respectively. From the memory perspective, the function is DRAM and Store-bound. The second main bottleneck is the function `foldLeft`, a Scala construct to manipulate generic lists for user defined computations. It is mainly frontend-bound and frontend latency-bound due to branch resteers (delays as a consequence of pipeline flushes). The lists manipulated are generic and the branch predictor incurs numerous miss-predictions while accessing the elements.

The efficiency of this algorithm is compromised due to expensive and inefficient data structures and wrappers, such as the `BinaryRegistry` operator class and the traversal of structures that incur into multiple branch resteers. The algorithm should be limited by the `ddot` function (which only accounts for 0.8% of the total clock ticks), but the overhead from accessing data structures is taking most of the clock ticks.

### 4.4 Dimensionality Reduction

Dimensionality Reduction (or Dimension Reduction) is the process of reducing the number of random variables under consideration via obtaining a set of principal variables. It can be used to extract latent features from raw and noisy features, or compress data while maintaining the structure. The algorithms analyzed are Singular Value Decomposition (SVD) and Principal Component Analysis (PCA). They have a decent IPC (around 1.8) and their principal function is `DSPR` (explained in Section 4.2.1).

#### 4.4.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is mainly backend-bound (28%) after general retirement (59%), with a similar distribution between memory and core-bound (13% vs. 15%, respectively). The function with a higher amount of clock ticks is `DSPR` (performing a symmetric rank-1 operation); this function takes 45% of the overall clock ticks. A large fraction of clock ticks is due to general retirement, which points to high efficiency. This is noticeable as this algorithm exhibits a high IPC with respect to the previous analyzed algorithms. Performance could be improved even more with vectorization. Even though having a high percentage of clock ticks in general retirement is good, in this case 21% of the clock ticks are due to Floating Point (FP) scalar arithmetic. With the appropriate vectorization, core-bound would be increased, but the overall execution time would be less because less instructions will be executed with their corresponding overhead removed.

#### 4.4.2 Principal Component Analysis

Principal Component Analysis (PCA) exhibits a similar behavior to the one observed in SVD, with some exceptions. The algorithm is backend-bound (41%) after general retirement (46%), however, the difference between backend-bound and general retirement is less than for SVD.
This is because PCA utilizes the *Breeze* linear algebra library, which internally is using some vector iterators that are more backend-bound. These functions account for 24% of the clock ticks. As in SVD, it also uses the rank-1 *DSPR* function, which accounts for 24% of the total clock ticks and could be faster by exploiting vectorization.

### 4.5 Feature Extraction

Feature Extraction (or Feature Selection) starts from an initial set of measured data and builds derived values (or features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps. This paradigm is related to dimensionality reduction. For example, Word2Vec computes distributed vector representations of words. The main advantage of distributed representations is that similar words are close in the vector space, which makes generalization to novel patterns easier, and model estimation more robust. Distributed vector representation is used in many natural language processing applications such as machine translation, named entity recognition, disambiguation, parsing, or tagging.

#### 4.5.1 Word2Vec

Word2Vec has a high percentage of retirement (65%), and an IPC of 2.1. After this, the algorithm is backend-bound (28%), and core-bound in particular (20%). The function with the highest percentage of clock ticks is *SAXPY*, the single-precision implementation of *AXPY* mentioned in Section 4.1, with 43% of the total clock ticks. This function could be optimized by using vectorization, given the fact that 93% of its clock ticks fall into general retirement category, and it is performing lots of FP-scalar arithmetic operations. The next function with the highest amount of clock ticks after *SAXPY* is *SDOT*, the single-precision implementation of the function *DOT*. In this case the function is 41% backend-bound, and Core-bound in particular. It also has a high percentage of clock ticks under general retirement, but the performance could be improved also using vectorization because all the operations are FP-scalar. By default, this algorithm runs on a single core and a single data partition. More cores can be involved by creating more data partitions (each core would work on a single partition) but that reduces the accuracy/generalization of the entire algorithm.

### 4.6 Frequent Pattern Mining

Frequent Pattern Mining refers to itemsets, subsequences, or substructures that appear in a data set with frequency (the frequency is given as a threshold). For example, a set of items such as milk and bread, that appear frequently together in a transaction data set, is a frequent itemset. In this study FP-growth has been analyzed. Given a data set of transactions, the first step of FP-growth is to calculate item frequencies and identify frequent items. After that, it creates a suffix tree (FP-tree) structure to encode transactions without generating candidate
sets explicitly, which are expensive to generate. Finally, the frequent items can be extracted from the FP-tree.

4.6.1 FP-growth

FP-growth is mainly backend-bound and, in particular, memory-bound (38%). The main bottleneck is primarily due to garbage collection derived from the Java Virtual Machine (JVM), mainly in function `copy_to_survivor_space`, from the `PSPromotionManager` class, which is heavily memory-bound with 18% of the algorithm’s total clock ticks. The second main bottleneck is due to software abstractions and, in particular, in `changeValue` which is part of the Scala collection for manipulating generic map objects. This function is called repeatedly in `insertAll`, from the `ExternalSorter` class. The functions are core-bound and they involve repeated mappings, aggregations, reductions and filtering by keys to compute frequent items. A fine-grained tuning of the JVM garbage collector’s generations could improve the performance.

4.7 Varying Algorithm Parameters

All the algorithms in MLlib are associated with numerous configuration parameters that can affect their performance. For the experiments in this thesis, the default configuration parameters have been used. The methodology employed however can be applied to reveal performance trends related to the choice of these parameters as well. A complete parameter exploration for all the studied algorithms requires an impractical amount of time, therefore for purposes of illustration, only regression decision trees have been analyzed.

When the tree is constructed, it can grow horizontally and vertically. The features are classified inside a `bin` if they are between a specific interval. A higher number of `bins` increase fine-grained classification, making the tree wider, whereas a deeper tree will generate more levels, making the tree higher. There are two main configuration parameters:

1) `maxBin`, determining the maximum number of bins to use when continuous features are discretized. Increasing `maxBin` will expand the tree horizontally, allowing the algorithm to make more fine-grained decisions, but increasing computation.

2) `maxDepth`, which if increased allows the algorithm to create a deeper tree expanding vertically, being more expressive. However, it increases computation (the maximum value allowed for the implementation of this algorithm is 30 in Spark 1.6.1).

Consider Figure 4.4 which presents the first (Figure 4.4a) and second level (Figure 4.4b) categorization of Top-Down as we keep `maxDepth` fixed but modify the `maxBin` parameter. As we increase the number of bins the algorithm can utilize, the decision tree training increasingly requires more instructions. However, as shown in Figure 4.4, the algorithm becomes increasingly backend-bound, and particularly memory-bound. This is due to the function `findBin`, inside the `TreePoint` class, which is the function with the highest percentage of clock ticks among all the experiments as `maxBin` increases (between 31% and 42% of the total clock ticks). The CPI of
the function becomes also worse as we increase the parameter, from 1.3 for $\text{maxBin} = 32$, up to a CPI of 6.0 for $\text{maxBin} = 1024$. Consequently, as we increase the number of bins, the bottleneck of the algorithm at the µarch level becomes worse, contributing to an additional increase in execution time apart of the non-linear computational complexity of the algorithm.

If we conduct the same experiment, but instead of increasing $\text{maxBin}$, we increase $\text{maxDepth}$ while keeping $\text{maxBin}$ constant, the total number of clock ticks for training also increases, but the proportion of clock ticks corresponding to each category of Top-Down remains fairly constant, as shown in Figure 4.5. A deeper look into the functions reveals that the parameter $\text{maxDepth}$ has a direct impact in the number of clock ticks spent in the function $\text{predictNodeIndex}$, from the class $\text{DecisionTree}$ and its related functions. In particular, this function goes from 1% of the total clock ticks for $\text{maxDepth} = 5$, up to 11% for $\text{maxDepth} = 30$, causing $\text{findBin}$ to go down from 32% of the total clock ticks to 11%, respectively. As the distribution of the bottlenecks do not change by increasing $\text{maxDepth}$, the performance of the algorithm will not be limited by the microarchitecture. Consequently, the algorithm scales more gracefully with this parameter than with $\text{maxBin}$.

Figure 4.3 shows the distribution of clock ticks per Core (primary vertical axis) and the CPU utilization taking into account the CPU frequency ratio (secondary vertical axis) for both experiments. The clock ticks per Core are computed as the total amount of clock ticks of the algorithm divided by the product of CPU usage and the CPU frequency ratio. The CPU usage (or utilization) is a value given by VTune that goes from (0.0, 16.0] in this machine, and indicates the average number of Cores used during the execution. In this machine, HyperThreading is disabled, and all the Cores are enabled, so the maximum value is 16 (with HT it would be 32). The CPU frequency ratio is a value that goes from (0.0, 1.0]. For example a value of 1.0 indicates the maximum frequency (2.6 GHz in this case), and as TurboBoost is disabled, values greater that 1.0 are not allowed. For all the experiments the CPU frequency ratio is between 0.98 and 1.00, so it is not very relevant.

It is noticeable that CPU utilization is constant for variable $\text{maxBin}$, but it goes down from 11 to 8 for variable $\text{maxDepth}$ after a value of 10. A possible explanation is that the CPU is reaching its physical limit for coherence communication across the cores and sockets due to the numerous and continuous accesses to the Last Level Cache (LLC), hurting the parallelism of the application. In addition, due to the uniform distribution of the training data, around half of the data will reside in other socket, causing more off-chip accesses. This explains the “bump” in execution time in Figure 4.5a for values of $\text{maxDepth}$ greater than 10. In addition, we can observe that for high values of $\text{maxBin}$, that is, greater than or equal to 256, it could be better to increase $\text{maxDepth}$ instead of $\text{maxBin}$ to obtain better accuracy in the trained model if we are concerned about the execution time. In fact, the accuracy improves better if we increase $\text{maxDepth}$ rather than $\text{maxBin}$, for example with $\text{maxDepth} = 10$, the accuracy is already better than all the other experiments with variable $\text{maxBin}$ (which appears to be constant, that is, it does not improve). However, this observation is not data independent, and different workloads
could lead to different results.

Figure 4.3: Clock ticks per Core (left vertical axis, bars) and CPU utilization × CPU frequency ratio (right vertical axis, line) for Regression Decision Trees varying maxBin and maxDepth parameters.

4.8 Summary

In this chapter, we described the main bottlenecks for a wide variety of machine learning algorithms on Spark 1.6.1, pointing to the responsible functions and high-level programming constructs, and suggesting optimizations. In general, there are four main problems that hurt the performance of all algorithms: overhead in data representation, excessive wrappers and dynamic dispatch, garbage collection, and lack of vectorization. Creating data structures with less overhead would increase spatial locality, allowing a better utilization of caches and decreasing memory bandwidth. Furthermore, less overhead means also less objects and structures, which would help with garbage collection. Using generic code and traits is good for code reusability and fast programming, but requires extra steps to perform even the simpler tasks such as a vector multiplication. In addition, these complex code constructs prevent the compiler to identify vectorization opportunities.

Finally, we analyzed the impact of varying algorithm parameters for Regression Decision Trees, showing how the Top-Down approach can be used to see how gracefully an algorithm scales from the µarch perspective. In the case that both parameters impact the computational complexity in the same way, leading to similar results, the next thing to do would see which one has less impact in the µarch. For example we showed how maxBin will increase the memory-bound category, whereas maxDepth has no impact in the µarch apart of lowering the CPU utilization after a value, but then it keeps constant.
Figure 4.4: Top-Down first and second level breakdowns for regression decision trees showing the percentage of pipeline slots falling into each category at issue point with fixed maximum depth in the tree ($maxDepth = 5$) and variable maximum number of bins ($maxBin = from 32 up to 512$).
Figure 4.5: Top-Down first and second level breakdowns for regression decision trees showing the percentage of pipeline slots falling into each category at issue point with fixed maximum number of bins ($\text{maxBin} = 32$) and variable maximum depth ($\text{maxDepth} = \text{from 5 to 30}$).
Chapter 5

From Spark version 1.6.1 to Spark version 2.0.0

This chapter shows the performance differences between Spark 1.6.1 with Scala 2.10 and Spark 2.0.0 with both Scala 2.10 and 2.11, using the same algorithms as the ones analyzed in Chapter 4. We analyze the MLlib API, which internally relies on RDDs. The developers of Spark claimed a lot of changes and improvements in Spark 2.0 with respect to previous versions, mainly in the Dataset API. With this analysis we can determine if the RDD API performs in the same way as before. Spark 2.0.0 by default uses Scala 2.11, whereas Spark 1.6.1 uses Scala 2.10. Therefore, we have compiled Spark 2.0.0 with both implementations of Scala in order to isolate performance issues that may be due to the Scala implementation instead of that of Spark. We needed a way to focus on relatively important differences, so we set the performance difference threshold at 6% as this resulted in a good balance between the number of cases that have to be highlighted and the potential benefit from alleviating the respective performance degradation causes. Therefore, only Naïve Bayes, K-means, Principal Component Analysis and FP-growth are analyzed in more detail in this study. The Top-Down categorization is shown in Figure 5.1. In addition, the execution times and associated speedups of all the algorithms are shown in Table 5.1.

The rest of this chapter is organized as follows. Sections 5.1 to 5.4 report the differences between Spark 1.6.1 and 2.0.0 only for the four aforementioned algorithms. Finally, Section 5.5 summarizes the findings and problems found during the analysis.

The algorithms whose execution time with Spark 2.0.0 is within ±6% of that with Spark 1.6.1 are:

- Singular Value Decomposition
- Logistic Regression
- Linear Regression
- Decision Trees
- Alternating Least Squares
Algorithms in Spark 2.0.0 whose execution time is greater than ±6% of that in Spark 1.6.1:

- Naïve Bayes
- K-means
- Principal Component Analysis
- FP-growth

5.1 Naïve Bayes

Naïve Bayes is on average 0.89× slower in Spark 2.0.0 compared to Spark 1.6.1. In Spark 1.6.1 and 2.0.0 with Scala 2.10, the function with the highest percentage of clock ticks is `prefixLength` from `GenSeqLike` class, a Scala collection, with 19% of the total clock ticks. However, in Spark 2.0.0 with Scala 2.11, the main bottleneck is `changeValue`, from the class `SizeTrackingAppendOnlyMap` (already explained in more detail in Section 4.1.4).

While we have tested Spark 1.6.1 and 2.0.0 both with Scala 2.10, the function that accounts for the second most clock ticks in Spark 2.0.0 with Scala 2.10 is not the same as in 1.6.1. The function `forall::apply` from `IndexedSeqOptimized` class only appears in 1.6.1, and accounts for 8% of the total clock ticks. We compiled Spark 2.0.0 with Scala 2.10 and 2.11, so we expected this function to appear also in Spark 2.0.0 with Scala 2.10, however, it did not. This function, although accounting for 8% of the total clock ticks in 1.6.1, seems to improve the overall application performance. In all the cases, regardless of Spark and Scala version, all the traces point to the same bottleneck, which can be avoided as stated in Section 4.1.4. There is no reason to check for non-negative features twice (as it is done now in the code), and just adding a precondition this can be avoided totally.

5.2 K-means

K-means is 0.71× slower in Spark 2.0.0 with respect to Spark 1.6.1. The distribution of clock ticks with Spark 2.0.0 between Scala 2.10 and 2.11 is very similar, therefore the rest of this analysis focuses on Spark 1.6.1 with Scala 2.10 and Spark 2.0.0 with Scala 2.11. The function with more clock ticks in both versions of Spark is the `dot` function; it accounts for 14% and 9% of the total clock ticks in Spark 1.6.1 and 2.0.0, respectively. This function performs better in 2.0.0 with 25% less clock ticks and almost equal number of instructions retired with respect to 1.6.1, with a CPI decrease from 2.5 in Spark 1.6.1 down to 1.9 in Spark 2.0.0.

We observe a considerable increase in garbage collection in Spark 2.0.0. On average, 46% extra clock ticks are spent in garbage collection. This comprises 20% of the total clock ticks
and 8.5% of the total instructions retired respectively in Spark 1.6.1, and 31% of the total clock ticks and 21.7% of the total instructions retired in Spark 2.0.0. A deeper analysis in the JVM garbage collector would be interesting to determine which objects are being collected, and to identify how the code generates ans used them. However, with our current methodology this information is not available, and special techniques to debug the garbage collector are required.

There are no significant changes in the training phase implementation for K-means between versions, and in both versions of Spark extra overhead is incurred traversing the internal interface

<table>
<thead>
<tr>
<th>Classification and Regression</th>
<th>SVM</th>
<th>Execution Time (ms)</th>
<th>Relative Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark Ver.</td>
<td>Scala Ver.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6.1</td>
<td>2.10.5</td>
<td>65,495</td>
<td>+1.60%</td>
</tr>
<tr>
<td>2.0.0</td>
<td>2.11.8</td>
<td>65,467</td>
<td>+0.23%</td>
</tr>
<tr>
<td>LogRegr</td>
<td>1.6.1</td>
<td>77,733</td>
<td>+1.74%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>79,087</td>
<td>+3.52%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>80,467</td>
<td>+3.52%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>80,467</td>
<td>+3.52%</td>
</tr>
<tr>
<td>LinRegr</td>
<td>1.6.1</td>
<td>31,636</td>
<td>+5.25%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>33,297</td>
<td>+8.27%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>32,543</td>
<td>+8.27%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>32,543</td>
<td>+8.27%</td>
</tr>
<tr>
<td>NaivBay</td>
<td>1.6.1</td>
<td>4,041</td>
<td>+12.05%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>4,528</td>
<td>+18.97%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>4,808</td>
<td>+18.97%</td>
</tr>
<tr>
<td>DecTree</td>
<td>1.6.1</td>
<td>34,199</td>
<td>+2.05%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>35,104</td>
<td>+2.05%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>33,297</td>
<td>+2.05%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>35,138</td>
<td>+2.05%</td>
</tr>
<tr>
<td>Collaborative Filtering</td>
<td>ALS</td>
<td>1.6.1</td>
<td>+5.11%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>17,127</td>
<td>+5.11%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>18,071</td>
<td>+5.11%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>17,378</td>
<td>+5.11%</td>
</tr>
<tr>
<td>K-means</td>
<td>1.6.1</td>
<td>19,948</td>
<td>+41.36%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>28,198</td>
<td>+41.36%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>27,918</td>
<td>+41.36%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>27,918</td>
<td>+41.36%</td>
</tr>
<tr>
<td>GaussMix</td>
<td>1.6.1</td>
<td>188,354</td>
<td>-4.12%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>180,588</td>
<td>-4.12%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>182,160</td>
<td>-4.12%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>182,160</td>
<td>-4.12%</td>
</tr>
<tr>
<td>Dimensionality Reduction</td>
<td>SVD</td>
<td>1.6.1</td>
<td>+0.16%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>4,536</td>
<td>+0.16%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>4,544</td>
<td>+0.16%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>4,607</td>
<td>+0.16%</td>
</tr>
<tr>
<td>PCA</td>
<td>1.6.1</td>
<td>7,904</td>
<td>-32.49%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>5,335</td>
<td>-32.49%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>5,307</td>
<td>-32.49%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>5,307</td>
<td>-32.49%</td>
</tr>
<tr>
<td>Feature Extraction</td>
<td>Word2Vec</td>
<td>1.6.1</td>
<td>+1.02%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>557,600</td>
<td>+1.02%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>563,294</td>
<td>+1.02%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>563,282</td>
<td>+1.02%</td>
</tr>
<tr>
<td>Frequent Pattern Mining</td>
<td>FP-growth</td>
<td>1.6.1</td>
<td>+58.83%</td>
</tr>
<tr>
<td></td>
<td>2.10.5</td>
<td>12,378</td>
<td>+58.83%</td>
</tr>
<tr>
<td></td>
<td>2.0.0</td>
<td>19,659</td>
<td>+58.83%</td>
</tr>
<tr>
<td></td>
<td>2.11.8</td>
<td>20,990</td>
<td>+58.83%</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time for the training stage (milliseconds)
tables from the JVM to select the proper methods to execute, accounting for 7-9% of the total clock ticks in both versions.

We suggest two ways to improve the performance in this case. First, analyze the problem with the garbage collector in Spark 2.0.0 in more detail, or do a proper garbage collector tuning. The JVM is the same for all the experiments in this section, and no relevant modifications have been done to the MLlib API for K-means, therefore the problem should be in some change in the Spark core. Second, do a better use of interfaces in order to reduce the excessive overhead of traversing the internal interface tables from the JVM in order to select the proper methods to execute.

5.3 Principal Component Analysis

For Principal Component Analysis (PCA), the main bottleneck in both Spark versions is function \textit{dspr} from the BLAS library. This function accounts for 30% of the total clock ticks in Spark 1.6.1, and 53% in Spark 2.0.0. However, the number of instructions retired is the same in both implementations, therefore this implies that the extra overhead in Spark 1.6.1 has been reduced in Spark 2.0.0.

The significant performance improvement in the algorithm is due to the changes in the code inside the class \textit{RowMatrix}, in the function \textit{computeColumnSummaryStatistics}. In particular, the improvements are attributed to two methods (\textit{add} and \textit{merge}) present in Spark 2.0.0. The major improvements in these methods are “in place merging”, i.e., no allocation for a result object is required, but the caller object is modified instead; and the other improvement is the “skipping of zero-values”.

Due to these changes, the overall IPC goes from 1.86 in Spark 1.6.1 up to 2.06 in Spark 2.0.0 and the performance increases by 1.49 in Spark 2.0.0, with 30% less clock ticks per core as shown in Figure 5.1.

5.4 FP-growth

FP-growth performs worse in Spark 2.0.0 regardless of the Scala version, with 0.63 slowdown for Spark 2.0.0 with Scala 2.10, and 0.59 slowdown for Spark 2.0.0 with Scala 2.11. Garbage Collection (GC) takes 47.8% of the total clock ticks in Spark 1.6.1, 63.5% in Spark 2.0.0 with Scala 2.10, and 73% in Spark 2.0.0 with Scala 2.11. In absolute terms, this is $1.73 \times$ and $2.87 \times$ extra clock ticks in Spark 2.0.0 with Scala 2.10 and 2.11 with respect to 1.6.1, respectively.

After GC, the main bottleneck is common for all versions of Spark and Scala, the function \textit{insertAll} from the class \textit{ExternalSorter}, a Scala utility for generic collections of objects that takes 20-24% of the total clock ticks.

As in K-means, we suggest analyzing the problem with the garbage collector in Spark 2.0.0 in more detail, or do a proper garbage collector tuning. The JVM is the same for all the
experiments in this section, and no relevant modifications have been done to the MLlib API for FP-growth, therefore the problem should be in some change in the Spark core. Due to the limitations in our profiling methodology related with the JVM, we cannot track the source of the inefficiency in this study.

5.5 Summary

In this chapter, we described the performance differences between Spark 1.6.1 with Scala 2.10, and Spark 2.0.0 with Scala 2.10 and 2.11. For Naïve Bayes, even though performance is worse in Spark 2.0.0, the main bottleneck is due to an unnecessary condition check. The algorithm checks twice if the features of all data points are non-negative. This should be done only once, or avoided altogether by just adding a precondition. The rest of the algorithm exhibits no clear bottlenecks.

For K-means and FP-growth, the main change in performance is due to an increase in garbage collection. However, with our methodology it is not possible to identify the source of the inefficiency (which are the objects that trigger the garbage collector), just the consequence (increase in overhead due to garbage collection) in this case. Further effort is required to identify the specific objects and trace their creation and use in the code. We leave a possible analysis and tuning of the garbage collector for future work, as it is out of the scope of our methodology in this study.

Finally, PCA performs considerably better in Spark 2.0.0, regardless of the Scala version. This is due to a change in the source code implementation in the algorithm in Spark 2.0.0 that uses techniques such as “in place merging” and “skipping of zero-values”.

For the remaining algorithms, there are no relevant changes between Spark 1.6.1 and Spark 2.0.0, therefore we assume the same bottlenecks dominate the algorithms across versions.
Figure 5.1: Top-Down first and second levels breakdown per algorithm showing the percentage of pipeline slots falling into each category at issue point. Values for Spark 2.0.0 with Scala 2.10 and 2.11 have been normalized with respect to the total number of clock ticks per core in Spark 1.6.1 with Scala 2.10.
Chapter 6

Datasets vs. RDDs

An RDD is a read-only, partitioned collection of records [63]. Records are stored in a distributed fashion in the RDD among the cluster and can be operated in parallel with a low-level API that provides transformations and actions. The main problem with the performance of RDDs in our context is the overhead associated with their unstructured data representation in conjunction with the overheads of the JVM. These overheads (among other factors) resulted in the development of the DataFrame and Dataset APIs; since Spark 2.0, both have been merged into the Dataset API in which a DataFrame is just an alias for a Dataset[Row]. These implementations provide a named column representation of the data, imposing structure onto the distributed data where further improvements take place. Some of them are:

- Static-typing and runtime type-safety.
- High-level abstraction and custom view into structured and semi-structured data.
- Ease-of-use of APIs with structure.
- Performance and optimization, such as the use of encoders to translate between JVM’s objects and Spark’s internal representation, or the use of Catalyst to generate optimized code, and query plans.

For more information about DataFrames and Datasets, please refer to Section 2.2.1.2. We are interested in understanding the precise performance implications of using the RDD API versus the Dataset API in Spark version 2. Adopting the terminology from the Scala packages in Spark, we will refer to the RDD API as the MLlib package (MLlib), and to the Dataset API as the ML package (ML) in what follows.

We compared the performance differences at the µarch and application level for RDD and Dataset APIs in Spark 2.0 for the algorithms shown in Table 6.1. Notice that the algorithms compared are a subset of those in Chapter 4 because there is no implementation for all the algorithms in the Dataset API for Spark 2.0 (as of the time of this writing) and we wanted to focus on what is publicly available. To demonstrate the performance of the algorithms, we will use the count of clock ticks per core for ML implementations normalized to the clock ticks per core for MLlib implementation. Clock ticks per core are the total number of clock ticks
across all cores divided by the average CPU utilization. The average CPU utilization is the number of cores used on average. In our system the maximum is 16 with HT disabled, or 32 with HT enabled. This metric is proportional to the execution time across the algorithms. In each graph, we present a category breakdown, for example in Figure 6.1 in ML, among all the clock ticks required in the execution, 40.8% were categorized as memory-bound, 25.6% were core-bound, and so on. In some algorithms there is only one underlying implementation of the training algorithm regardless of the API utilized (MLlib or ML). In order to run the algorithms, mappings over the input data are required; for example from a Dataset into an RDD in the case that the training implementation requires an RDD as input. Such mappings incur a soft copy (copies only references). We will point out if/when this occurs and analyze associated overheads.

The rest of this Chapter is organized as follows. Sections 6.1 to 6.3 compares Datasets and RDDs for the presented categories in Table 6.1 and algorithms from a µarch and application perspective, applying Top-Down. Finally, Section 6.4 summarizes the most relevant findings, and suggests more optimizations.

### Table 6.1: Execution time for the training (milliseconds)

<table>
<thead>
<tr>
<th>Category</th>
<th>Algorithm</th>
<th>Package</th>
<th>Execution Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification and Regression</td>
<td>LogRegr</td>
<td>MLlib</td>
<td>51,822</td>
<td>1.64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML</td>
<td>31,557</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NaivBay</td>
<td>MLlib</td>
<td>4,553</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML</td>
<td>7,220</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DecTree</td>
<td>MLlib</td>
<td>36,576</td>
<td>1.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML</td>
<td>28,355</td>
<td></td>
</tr>
<tr>
<td>Clustering</td>
<td>K-means</td>
<td>MLlib</td>
<td>27,998</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML</td>
<td>31,906</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GaussMix</td>
<td>MLlib</td>
<td>188,158</td>
<td>1.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML</td>
<td>142,184</td>
<td></td>
</tr>
<tr>
<td>Feature Extraction</td>
<td>Word2Vec</td>
<td>MLlib</td>
<td>562,153</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML</td>
<td>476,686</td>
<td></td>
</tr>
</tbody>
</table>

### 6.1 Classification and Regression

Logistic Regression and Regression Decision Trees perform better with the Dataset API, primarily due to the improve data layout and representation. For Naïve Bayes, the implementation used underneath is MLlib. In this case, the performance of ML is worse primarily due to the overhead of extra transformations.
6.1.1 Logistic Regression

Logistic Regression executes in both packages (MLlib and ML) the training implementation from the ML package, which accepts a Dataset[_] as input. However, the optimizer used in this implementation is not Stochastic Gradient Descent (SGD) as in Section 4.1.2 because such implementation was not available for ML in Spark 2.0. Therefore, we have used the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) optimizer in both MLlib and ML APIs in this Chapter. The input from MLlib is an RDD[LabeledPoint], consequently a transformation is required prior to training in ML as shown in Listing 6.1, line 5, where the input RDD is mapped into an ML representation and transformed into a DataFrame. The transformation process and the posterior data access in the training incur the overhead in clock ticks observed in Figure 6.1 for MLlib. For the case of MLlib package, the data transformation and access overhead accounts for more than 41% of the total clock ticks and more than 46% of the total instructions retired\(^1\). In ML it accounts for around 1% of the total clock ticks and 2% of the total instructions retired. The main reason for this difference is that the MLlib package loads the data from a file into an RDD, and there is a chain of transformations until the data is finally accessed (13 intermediate transformations in total). Notice that such transformations take place every time the data is accessed in order to perform an action. In ML, however, the data is loaded from a file using directly an internal row representation and uses the optimizations from Tungsten such as encoders and off-heap memory management, therefore fewer and faster transformations are required to access the data (7 intermediate transformations). In the end, the Dataset API is \(1.64 \times\) faster than the RDD API.

We detected an implementation problem that hurts performance in our experiments. In Listing 6.1, line 7, there is a condition that checks if the input RDD is already cached in

\(^1\)The functions related with data access and transformations with higher clock ticks and instructions retired are \texttt{jint\_disjoint\_arraycopy} and \texttt{apply}, from the dynamically generated class \texttt{SpecificSafeProjection}.\n
memory. If that is the case, the DataFrame generated to run the training will not be cached. In our experiments we always cache the input, regardless of its type, therefore the condition in line 7 will be evaluated to false. Each time the training phase access the input DataFrame, it will perform the transformation from line 5. We have modified the code in order to make condition in line 7 evaluate always to true, forcing in this way the caching in memory of the resulting DataFrame. After this modification, the functions related with the last transformation went from 45.5% of the total clock ticks down to 2.7%. The results are shown in Figure 6.1 as MLlib (C). ML is still 46% faster in execution time than the modified MLlib version, but by caching there is a speedup of 12% in execution time with respect to the version that does not cache the DataFrame.

Regardless of the API used, in all cases the algorithm is limited by the function \textit{add} from the class \textit{LogisticAggregator}. This function does not use any acceleration library to perform the mathematical operations, and is heavily core-bound due to the amount of divisions required. It is also memory-bound due to the multiple elements required to compute each output that reside in other socket, incurring numerous NUMA accesses.

```scala
private def run(input: RDD[LabeledPoint], initialWeights: Vector, userSuppliedWeights: Boolean): LogisticRegressionModel = {
  def runWithMlLogisticRegression(elasticNetParam: Double) = {
    val spark = SparkSession.builder().sparkContext(input.context).getOrCreate()
    // Convert our input into a DataFrame
    val df = spark.createDataFrame(input.map(_.asML))
    // Determine if we should cache the DF
    val handlePersistence = input.getStorageLevel == StorageLevel.NONE
    // Train our model
    val mlLogisticRegressionModel = lr.train(df, handlePersistence)
    ...
  }
}
```

Listing 6.1: mllib.classification.LogisticRegression.scala

### 6.1.2 Naïve Bayes

Naïve Bayes performs worse for ML than MLlib, as shown in Table 6.1. Although the implementation of the ML package is utilizing the one from MLlib, there are significant differences in the functions that appear most costly (in terms of clock ticks) among implementations. In the ML package, even though data is loaded as a Dataset[Row] (i.e., a DataFrame), the DataFrame is transformed before training into an RDD[LabeledPoint] for compatibility, as shown in Listing 6.2, lines 5 to 6. This transformation does not take place until an action is performed, and in this case the action is the \textit{collect} method from the \textit{run} method in the MLlib package. The main difference among implementations is in the way the data in the RDD is accessed. In the ML package, the RDD created for compatibility from a DataFrame keeps track of its lineage and in particular the descriptive data layout. This effectively means that a special class is generated
at runtime in order to provide a way to access the data from the DataFrame. Despite the fact that data can be accessed more effectively in ML, the overhead of the extra operations for transforming and accessing the data in this case is high, and it requires more time than utilizing the non-optimized data layout from the RDD that MLlib implementation provides. This behavior is observed in function `apply`, from the dynamically generated class `SpecificSafeProjection`, which is the most costly function in ML with 9.6% of the total clock ticks and 16.2% of the total instructions retired.

```scala
import org.apache.spark.mllib.classification.{NaiveBayes => OldNaiveBayes}
import org.apache.spark.mllib.regression.{LabeledPoint => OldLabeledPoint}

... override protected def train(dataset: Dataset[_]): NaiveBayesModel = {
  val oldDataset: RDD[OldLabeledPoint] =
    extractLabeledPoints(dataset).map(OldLabeledPoint.fromML)
  val oldModel = OldNaiveBayes.train(oldDataset, $(smoothing), $(modelType))
  NaiveBayesModel.fromOld(oldModel, this)
}
```

Listing 6.2: ml.classification.NaiveBayes.scala

Some improvements could be applied to both implementations of this algorithm, as stated in Section 4.1.4. Checking for non-negative features in the input dataset is required only once (in the current implementation is performed twice). Caching the RDD resulting from the input Dataset’s transformation in line 6 of Listing 6.2 is in general recommended. However, in this case, the RDD is only utilized once in the training algorithm before it is collected and transformed into a Sequence. Therefore, the costs incurred by transformation and caching are not amortized across iterations, and if caching is implemented it will result in further overhead. Evidently, an implementation of the algorithm for the ML package that does not require a transformation into an RDD[LabeledPoint] would make the algorithm run faster since it would avoid redundant and expensive computations and will be fully benefit from better data layout in the Dataset API.
6.1.3 Decision Trees

Regression Decision Trees utilizes the implementation from the ML package in both APIs (MLlib and ML). The function that consumes the highest number of clock ticks is `labeledPoint-ToTreePoint`, from the `TreePoint` class, which performs a binary search per feature for each data point. This function accounts for 12.5% of the total clock ticks and 18% of the total instructions retired in MLlib; whereas it accounts for 20% of the total clock ticks and 25.3% of the total instructions retired in ML. A deeper look into this function shown in Figure 6.4 reveals a significant change in behavior. In MLlib the function was neither core-bound nor memory-bound, whereas in ML it appears 18.3% core- and 6.8% memory-bound. There is also a noticeable decrease in the front-end latency category, from 11.8% in MLlib down to 6.4% in ML. However, 20% extra clock ticks per core are spent now in ML for this function. The changes observed in the core- and memory-bound categories in ML are attributed to the improved data layout offered by Datasets. Despite the fact that both APIs are using Datasets underneath, the implementation from MLlib is using an RDD as input that has to be transformed into a Dataset (in a similar way as Listing 6.1), losing the potential advantages of serialization and compression that a Dataset would offer. In addition, the percentage of clock ticks spent in the L1-bound, L3-bound and Store-bound categories have increased in ML. The relative number of clock ticks in these categories is almost 12 times higher than in MLlib. This function in ML is able to execute more instructions without flushing the pipeline due to reduced bad speculation, and since data is now mainly in the caches instead of DRAM, the caches provide data to the back-end faster. As a result, the number of clock ticks spent in the core-bound category increases since all the functional units are full, improving overall performance. Overall, this translates to a speedup of 1.29 for ML over MLlib.
### 6.2 Clustering

K-means and Gaussian Mixture execute the implementation from the MLlib API regardless of the selected package (i.e., ML or MLlib), and therefore they require a transformation from the input Dataset into and RDD to execute properly if the ML API is used. In both cases, we show how caching the transformed data in the right place can lead to performance improvements that otherwise, will cause slowdown in the algorithms.

#### 6.2.1 K-means

<table>
<thead>
<tr>
<th></th>
<th>ML</th>
<th>ML (C)</th>
<th>MLlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>8.30%</td>
<td>5.90%</td>
<td>5.00%</td>
</tr>
<tr>
<td>ML (C)</td>
<td>29.00%</td>
<td>31.20%</td>
<td>31.50%</td>
</tr>
<tr>
<td>MLlib</td>
<td>24.50%</td>
<td>15.70%</td>
<td>18.60%</td>
</tr>
</tbody>
</table>

Figure 6.5: Top-Down second level analysis for K-means. ML corresponds to the regular implementation of ML, whereas ML (C) corresponds to the modified version from Listing 6.3.

K-means performs worse for ML package than MLlib, with a slowdown of 0.88. The function consuming the largest amount of clock ticks in MLlib is `dot` from the BLAS package (the input vector is normalized, so dot products are utilized), with 9.6% of the total clock ticks and 3.5% of the total instructions retired. In ML, however, this function accounts for only 1.3% of the total clock ticks and 2.7% of the total instructions retired. In absolute terms for this function,
the ML implementation has 5.73× less clock ticks with respect to MLlib and the same amount of instructions retired; it requires less clock ticks to do the same job. To understand the reason behind the performance difference, after code inspection, we observe that a transformation from DataFrame to RDD is required to run the ML implementation; this transformation imposes an overhead every time the data is accessed. For this reason, in ML the function with the highest number of clock ticks, apply from the dynamically generated class SpecificSafeProjection, accounts for 11.9% of the total clock ticks and 12% of the total instructions retired. There is also extra overhead incurred by the dynamic code generated to access the data, with 1.79× extra clock ticks and 1.55× extra instructions retired compared to the MLlib implementation. In Figure 6.5 we observe that the algorithm in ML is less memory-bound and more core-bound in comparison to MLlib. The ML implementation would perform much better if the transformation from Dataset to an RDD is cached, so the overhead of the transformation would be incurred only once; after that, the data would be reused with no transformation overhead every time it is accessed.

In support of this point, we modified the implementation calling the cache method, as shown in Listing 6.3, line 5. The results in Figure 6.5 show that after executing the algorithm with this modification, the ML implementation is 1.10× faster than the MLlib implementation. Since the data is cached and the transformation overhead every time the data is accessed is eliminated, this translates in a reduction of the total number of clock ticks in the function responsible for data access from 11.9% down to 0.7%, improving the overall performance.

### 6.2.2 Gaussian Mixture

Gaussian Mixture performs better with ML implementation with respect to MLlib implementation with a speedup of 1.32. The function consuming the largest number of clock ticks in ML, apply from the class BinaryRegistry, requires 27.9% less clock ticks while retiring the same amount of instructions as in the MLlib implementation. This function is 12.2% less memory-bound in ML, and slightly more core-bound (1.4% more). It also has 8.4% more general retirement clock ticks. Figure 6.7 presents the memory-bound category breakdown for this function. The ML implementation is more L1- and L3-bound, instead of DRAM-bound as in MLlib, pointing to improved cache utilization. The implementation of this algorithm in ML transforms a DataFrame to an RDD in order to execute the training implementation from MLlib. However,
the RDD is cached after the transformation, so the overhead of the transformation is eliminated as it appears only the first time the data is accessed. Elimination of these overheads and the superior cache (L1 and L3) performance are responsible for the speedup.

6.3 Feature Extraction

Word2Vec performs better in the ML API. However, this API is using the implementation from the MLlib package underneath, so a transformation from a Dataset into an RDD is required. Even with the overhead of this transformation, the performance is better. Caching the transformed data can improve the performance even more, reducing the garbage collection overhead and utilizing better the core’s resources.

6.3.1 Word2Vec

Word2Vec achieves 1.18 speedup with ML over MLlib. However, both APIs are utilizing the implementation from MLlib. In MLlib package the functions with the highest number of clock ticks are \textit{saxpy} and \textit{sdot}, which belong to the BLAS library and perform inner products with loop unrolling. These functions combined account for 65.8% of the total clock ticks and 83.4% of the total instructions retired. However, in ML these are the second and third functions with
the highest number of clock ticks; the function with most clock ticks, \texttt{steal\_best\_of\_2}, from the \texttt{GenericTaskQueueSet} class, is related to the garbage collection and accounts for 28.3\% of the total clock ticks and only 7.4\% of the total instructions retired. In contrast, \texttt{saxpy} and \texttt{sdot} together account for 30.4\% of the total clock ticks in ML and 70.9\% of the total instructions retired. Comparing the total number of clock ticks between MLlib and ML, it appears that ML consumes 63\% more clock ticks, however, it is 18\% faster.

In order to understand the performance advantages of the ML implementation, we observe that Word2Vec is an optimization algorithm that continuously iterates over the input data solving a complex optimization problem. By default the algorithm runs on a single core with a single data partition. More cores can be involved by creating more data partitions (each core working on a partition) but that reduces the accuracy/generalization of the entire algorithm, therefore in our experiments we utilize the default settings. In the ML implementation the use of Datasets provides advantages in data representation and access, and the algorithm iterates over the input data (strings in this case) faster. However, as in K-means in Section 6.2.1, the Dataset is transformed into an RDD in order to run the training, and there is no caching after this transformation. During an iteration, the algorithm generates array objects that are only used inside the iteration loop over individual data items; consequently, the garbage collector is triggered multiple times to free the resources used by these local arrays. As the time to access the data is smaller (due to improved data layout), the overheads of garbage collection are more visible. Furthermore, the transformation overhead due to the absence of caching is also responsible for the increased amount of clock ticks in garbage collection. The collection process is multi-threaded, and even though the garbage collector overhead is larger, it is able to collect using multiple threads. This behavior is reflected in the average CPU utilization, which is 1.23 for MLlib, whereas in ML is 2.35. Overall, the implementation from ML requires more clock ticks than the implementation from MLlib, with the extra clock ticks attributed to garbage collection. Nevertheless, since this process takes place in parallel, the overall ML
implementation runs faster. The extra garbage collection clock ticks are also responsible for the increase in the memory-bound category as shown in Figure 6.8, because this function is 76% memory-bound. By enabling caching in the right place, the speedup goes from 1.18 up to 1.24, and the algorithm turns more core-bound, as shown in Figure 6.8.

6.4 Summary

In this Chapter, we compared Datasets and RDDs on Spark 2.0.0 at the µarch and application levels for a subset of algorithms from Chapter 4. In general, we identified that for MLAs, caching in software a Dataset or RDD before performing an action is recommended if the data is going to be reused (e.g., Logistic Regression, K-means, Word2Vec). Garbage collection has been reduced with Datasets, however, the algorithms are still memory- and core-bound. All the algorithms suffer from NUMA latency, therefore improving the data layout being aware of this limitation would help increasing performance. Nonetheless, MLAs usually require iterating multiple times over the entire data, so isolating the data in a single socket is most of the times impossible. A possible solution for this limitation is the use of prefetchers that would fetch the data from the other socket in advance. If this is done at the right time, by the time the thread accesses the data, it will be already in the cache. In MLAs, the access pattern of the data is hard to identify in advance due to the dynamic properties of the training, e.g., as the model trains, the algorithm will converge to a solution, so data points that were accessed frequently in the beginning may not be accessed again, or could be accessed in a less frequent manner as the model converges. A prefetcher that is aware of this characteristics could improve the performance considerably.

Another solution is to redistribute the data in memory as the model is trained, bringing the data closer to the cores that are using it, however the cost of moving the data from one socket to another could overcome the benefits of having the data closer to the core that is frequently using it.

For Datasets, Tungsten engine improves performance by reducing the overhead of the data structures in memory using encoders/decoders, and dynamically generating the code required to access these structures. It would be interesting to analyze this (en/de)coding system to see if a hardware accelerator for this task is feasible. In this way the data could be accessed in a regular way, but it would be decoded in hardware before bringing the data to the core, and encoded before saving the results in memory. If the data is packed properly, advanced features such as SIMD instructions (e.g., AVX, SSE) should be compatible with this accelerator.

In general, we observed that a uniform programming style should be used in Spark. However, due to the open-source characteristics of the project and the increasing number of developers, establishing a uniform programming style is hard. Right now, the programmer that is using the MLlib API does not know if the input data should be cached or not, as we showed in Logistic Regression for MLlib. Being able to cache the input Dataset/RDD before the training should be an available option. We showed how this can improve the performance in K-means, Logistic
Regression and Word2Vec implementations.

Finally, having a Dataset wrapped as an RDD or vice-versa in MLlib allows code reusability and developing time savings. However, the new features and performance improvements brought by Tungsten cannot be fully exploited in this environment. We suggest a new implementation for Datasets from scratch that is capable of extract all the potential brought by Tungsten.
Chapter 7

Impact of System Parameters

This chapter compares the impact of system choices to the algorithms performance on Spark 2.0.0 for MLlib and ML APIs. Section 7.1 analyzes the impact of the Java Virtual Machine (JVM) choice to the overall performance. Section 7.2 measures the impact of running multiple versus single threads per core. Finally, Section 7.3 summarizes the findings in the previous sections and the possible optimizations.

7.1 Java Virtual Machine

We compared the three most popular implementations of the Java Virtual Machine for Java 8. These implementations are OpenJDK, Oracle’s Java HotSpot (Oracle) and IBM J9 (IBM). Default Spark system parameters are used in all cases, only the binary for Java has been replaced with its corresponding implementation. We analyzed the algorithms from Chapter 6 for both MLlib and ML APIs; the results are depicted in Figure 7.1. We observe that the Oracle JVM is on average (geometric mean) 4% faster than OpenJDK using the MLlib API, and 5% faster using the ML API. The results for IBM show that is on average 8% slower than OpenJDK using the MLlib API, and 10% slower using the ML API for the set of algorithms presented. These results differ from those reported by Yasin et al. [62], where using Java 6 it was observed that Oracle’s JVM was $1.43 \times$ faster than OpenJDK. In our case we observe a performance improvement from Java 6 to Java 8 in OpenJDK’s JVM implementation that closes the gap with Oracle’s JVM which is only $1.05 \times$ faster. Also Yasin et al., used a different Java version and evaluated Apache Mahout and Hadoop, and not Spark. Comparing OpenJDK and Oracle to IBM JVM, the latter are consistently more often front-end bound, as shown in Table 7.1. We postulate that the increase in this category is due to a larger number of instructions generated, which are also more complex than the instructions generated by OpenJDK or Oracle, therefore hurting the performance of the instruction decoders. This contributes to an increase in the front-end latency due to instruction cache misses (the code does not fit in the instruction cache) and the Micro-Sequencer (MS) switches (the MS is used for CISC instructions not fully decoded by the default decoders). Subsequently, this also increases the front-end bandwidth in some
cases due to the Micro Instruction Translation Engine (MITE), which is the unit that decodes instructions into µops; and the Decoded Stream Buffer (DSB), which is a unit that caches these µops after they have been decoded by the MITE (as well as the Micro Sequence ROM). In order to solve these, the JIT compiler should be optimized in order to generate smaller code that fits in the cache, and use simpler and more efficient instructions which are easier to decode by the cores. However, replacing a complex instruction with simpler ones would increase the code size, exacerbating the i-cache problem. Table 7.1 shows that IBM has consistently higher retirement percentage with respect to OpenJDK and Oracle. This is related to the previous observation since if more instructions are required to perform the same task, also more instructions will be retired by the cores.

Table 7.1: First level of Top-Down metrics including the average CPU utilization, CPI and speedup for ML API with three different JVMs, taking OpenJDK as a baseline in ML API in Spark 2.0.0.
7.2 Simultaneous Multi-Threading vs. Single-Threading

Simultaneous Multi-Threading (SMT) executes multiple threads on a single core without switching so that a single physical core appears as multiple logical cores. Some studies [19, 36, 37] show that SMT can improve performance up to 30% on common server application benchmarks while requiring just 5% extra chip size. We have compared the same algorithms from Section 7.1 with MLlib and ML APIs using Spark 2.0.0 and the following CPU configurations for SMT and ST (Single-Threaded):

- **2×4×1**: ST configuration with 2 sockets × 4 cores per socket × 1 thread per core, 8 threads in total.
- **2×4×2**: SMT configuration with 2 sockets × 4 cores per socket × 2 threads per core, 16 threads in total.
- **2×8×1**: ST configuration with 2 sockets × 8 cores per socket × 1 thread per core, 16 threads in total.
- **2×8×2**: We do not use this configuration because we cannot compare against 32 single threads.

![Figure 7.2: SMT vs. ST speedup, normalized to 2 sockets × 4 cores/socket × 1 thread/core execution time.](image)

Using configuration 2×4×1 as a baseline, the geometric mean of the speedups for all algorithms in MLlib and ML APIs is shown in Figure 7.2. In addition, the average CPU utilization is shown in Figure 7.3. It is evident that the resources are underutilized and this represents a potential for further improvements, given the fact that the maximum theoretical speedup for twice as many cores would be 2.0, and the ideal average CPU utilization should be around 16. Overall in the Top-Down analysis, categories move from bandwidth-bound towards more latency-bound due to the multiplexing of some shared resources (e.g., the instruction decoders in the front-end, or the cache in the back-end). This is fine when we have multithreading which typically sacrifices individual latency for overall throughput. In addition, as the threads share CPU resources, more contention due to core and memory-bound is observed. D. Marr
et al. [19] showed performance gains of up to 30% on common server application benchmarks for the Intel® Xeon® processors. In order to estimate if SMT is performing effectively, we use the Hyper-Threading Effectiveness [49] metric, shown in Equation 7.1, which estimates how close Hyper-Threading is from 30% of the achievable speedup in a multicore configuration with the same number of logical threads. \( HT \text{ Scaling}_{\text{observed}} \) is the speedup observed when HT is enabled, whereas \( DP \text{ Scaling}_{\text{observed}} \) is the speedup observed by doubling the number of cores with respect to the single-threaded baseline. The derivation of the constants in the Equation come from Amdahl’s Law, and are explained in detail in [49]. Figure 7.4 shows a geometric mean among the algorithms of 1.02 for MLlib and 1.04 for ML (values greater than or equal to 1 are good). This means that SMT is performing effectively in general, and should be enabled.

\[
HT \text{ Effectiveness} = HT \text{ Scaling}_{\text{observed}} \times \left( 0.538 + \frac{0.462}{DP \text{ Scaling}_{\text{observed}}} \right) \quad (7.1)
\]

![Figure 7.3: Average CPU Utilization for ML API.]

![Figure 7.4: HT Effectiveness. Values greater than or equal to 1 are desirable, meaning that HT is achieving 30% of what could be achieved by doubling the number of cores.]

### 7.3 Summary

This Chapter compared the impact of system choices to the performance of the algorithms on Spark 2.0.0 for MLlib and ML APIs.
Starting with the choice of the Java Virtual Machine, and setting OpenJDK as a baseline, we showed that Oracle performs on average 4% faster for MLlib API, and 5% faster for ML API. On the other hand, IBM performs 8% slower for MLlib API, and 10% slower for ML API. IBM produces more and more complex instructions than OpenJDK or Oracle, which increases front-end latency. An optimization of the JIT compiler for IBM is required to address this problem.

Finally, we compared how SMT performs. It improves performance 13% on average for MLlib, and 18% for ML. This is around 50% of the speedup possible when doubling the number of cores on average for both APIs, therefore enabling HT will help on MLAs with Spark. The speedup difference shown in Figure 7.2 suggests that there is room for more parallelism in the application, given that by doubling the number of cores the algorithms do not fully utilize all the cores in the processor.
Chapter 8

Conclusions

This thesis presented a detailed analysis at the application and microarchitectural levels of popular Machine Learning Algorithms on Spark, and their performance differences between Spark 1.6.1 and Spark 2.0.0.

We adopted a trace driven approach, deploying the Top-Down methodology from Yasin et al. [61], and attributing the bottlenecks at the µarch level back to specific functions. In addition, we analyze the performance differences of RDDs and Datasets in Spark 2.0.0. Finally, we show how the choice of a different Java Virtual Machine and multi-threading influence the performance on Spark 2.0.0.

Our study shows how new technologies that facilitate the application development such as dynamic code generation, software abstractions and garbage collection, facilitate application development, but introduce significant overheads. Our work also shows that MLAs on Spark are mainly memory-bound due to the NUMA architecture of modern systems, and this problem is exacerbated by the aforementioned technologies.

We identified that the main problems for MLAs on Spark are the overhead in the data representation, excessive use of wrappers in libraries, overhead in dynamic dispatching due to interfaces, garbage collection, and lack of vectorization.

As applications and platforms evolve and software nesting becomes deeper, we believe that studies as the one presented will become crucial to understanding and improving application performance in the Big Data and Machine Learning era.

8.1 Future Work

The analysis and conclusions presented in Chapters 4 to 7 open doors to further research.

A natural step would be studying the performance as the number of computation nodes increase, now that we have established a good baseline and methodology. VTune has tools that would allow the instrumentation of more than one node in the cluster. However, due to the multiple agents in Spark, it will be a challenge to determine which thread in one node
triggered other tasks in other nodes. Creating a complete call-stack trace of the entire cluster is a challenging problem.

As described in Chapter 5, Top-Down and VTune are not intended for analyzing the JVM garbage collector. Therefore, it would be interesting to use additional tools such as JProfiler [12] and GCViewer [53] to analyze the performance issues related with garbage collection, and also study the possible tuning of the garbage collector in the JVM for Spark and MLAs.

As suggested in Chapter 3, it would be interesting to analyze the impact of other system parameters such as TurboBoost, or the different types of prefetchers available in the processor.

In Chapter 6 we suggest three possible optimizations to alleviate congestion in NUMA systems and improve the performance. The first one consists of a prefetcher that would be aware of the machine learning algorithm being executed, and in this way would be able to prefetch the data timely to the corresponding core. This prefetcher could take into account the different latencies that depend on where the data resides, e.g., local RAM or remote socket RAM, local LLC, or remote socket LLC.

The latency access problem of NUMA systems observed in our analysis could also be addressed by redistributing the data in the system, either programming data structures that are aware of this characteristic, or distributing the data dynamically as the application runs. In some MLAs, it is possible that as the model gets trained, some data points are not accessed anymore. The second optimization would be to allow the system to detect this, making possible to redistribute the data among the NUMA system in a way that favors the latency access to the data by bringing it closer to the cores that are going to access it. However, it may happen that redistributing the data actually increases congestion and hurts the performance even more.

The third potential improvement that we propose is a hardware encoder/decoder that is able to understand the layout of the data structures from the Dataset API and the Tungsten project. In this way, the size of the information stored in memory would be reduced, and the software encoding/decoding part would be replaced by this piece of hardware. If the data is packed properly, advanced features such as SIMD instructions (e.g., AVX, SSE) could be used in combination with the encoder/decoder, improving the performance. The aforementioned solutions to deal with NUMA systems are orthogonal and could be used combined.

Finally, another avenue of research would be to extend this study into popular workloads and additional Spark packages, e.g., Spark SQL, and embed such methodologies into specific systems that would be utilized for performance monitoring as well as prediction of popular workloads.
Appendices
Appendix A

Spark Configuration

Spark has multiple properties and configuration options. We only list those properties that override the default configuration. For the rest of properties and their corresponding values, please refer to https://spark.apache.org/docs/1.6.1/configuration.html for Spark 1.6.1, and https://spark.apache.org/docs/2.0.0/configuration.html for Spark 2.0.0. The configuration parameters that have been overridden are listed in Table A.1.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>local[*]</td>
<td>The cluster manager to connect to.</td>
</tr>
<tr>
<td>driver.memory</td>
<td>40g</td>
<td>Amount of memory to use for the driver process.</td>
</tr>
<tr>
<td>executor.extraJavaOptions</td>
<td>-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xcomp -XX:-UseLoopCounter</td>
<td>A string of extra JVM options to pass to executors.</td>
</tr>
</tbody>
</table>

Table A.1: Spark 1.6.1 and 2.0.0 non-default configuration parameters.
Appendix B

VTune Configuration

In this study, we used Intel® VTune Amplifier XE 2016 [13]. We used the analysis type “Microarchitecture Analysis/General Exploration”. Inside this configuration, we performed two kinds of analysis: 1) Multiple runs analysis, where the application is run multiple times, and each time the performance counters that usually are multiplexed among events in the same run are now dedicated for an event per run. In this way, we achieve higher confidence in the results we get. 2) Call-stack trace analysis, where the application is run only once, and therefore the performance counters are multiplexed in the same run, but we get complete call-stack traces of the application. When the aforementioned analysis are combined, we are able to identify the performance bottlenecks accurately and link them to their corresponding functions back in the source code. We summarize in Listing B.1 the configuration parameters for multiple-runs that are exclusive of this analysis, in Listing B.2 those for call-stack traces analysis exclusively, and finally in Listing B.3 those that are used regardless of the analysis type.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-knob enable-stack-collection=false</td>
</tr>
<tr>
<td>2</td>
<td>-allow-multiple-runs</td>
</tr>
</tbody>
</table>

Listing B.1: VTune configuration for multiple runs.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-knob enable-stack-collection=true</td>
</tr>
<tr>
<td>2</td>
<td>-no-allow-multiple-runs</td>
</tr>
</tbody>
</table>

Listing B.2: VTune configuration for call-stack traces.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-collect-with runsa</td>
</tr>
<tr>
<td>2</td>
<td>-knob stack-size=0</td>
</tr>
<tr>
<td>3</td>
<td>-knob stack-type=software</td>
</tr>
</tbody>
</table>

Listing B.3: VTune configuration for general analysis.
-knob enable-call-counts=false
-knob enable-trip-counts=false
-knob event-config=
  CYCLE_ACTIVITY.STALLS_LDM_PENDING, 
  CYCLE_ACTIVITY.CYCLES_NO_EXECUTE, 
  OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_DATA_RD, 
  OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD:cmask=6, 
  FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE, 
  FP_COMP_OPS_EXE.SSE_PACKED_SINGLE, 
  SIMD_FP_256.PACKED_SINGLE, 
  SIMD_FP_256.PACKED_DOUBLE, 
  UOPS_EXECUTED.THREAD, 
  FP_COMP_OPS_EXE.SSE_SCALAR_SINGLE, 
  FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE, 
  FP_COMP_OPS_EXE.X87, 
  BACLERS.ANY, 
  OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HITM_OTHER_CORE_0, 
  MACHINE_CLEARS.COUNT, 
  LD_BLOCKS.NO_SR, 
  MEM_LOAD_UOPS_RETIRED.L1_MISS, 
  L1D_PEND_MISS.PENDING, 
  DSB2MITE_SWITCHES.PENALTY_CYCLES, 
  RS_EVENTS.EMPTY_END, 
  ICACHEIFOETCH_STALL, 
  RS_EVENTS.EMPTY_CYCLES, 
  RESOURCE_STALLS.SB, 
  DTLB_STORE_MISSES.WALK_DURATION, 
  DTLB_STORE_MISSES.STLB_HIT, 
  MEM_LOAD_UOPS_RETIRED.LLC_MISS_PS, 
  IDQ_MS_UOPS, 
  IDQSTALL.LCP, 
  UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC, 
  UOPS_EXECUTED.CYCLES_GE_2_UOPS_EXEC, 
  CYCLE_ACTIVITY.STALLS_L1D_PENDING, 
  CYCLE_ACTIVITY.STALLS_L2_PENDING, 
  UOPS_EXECUTED.CYCLES_GE_3_UOPS_EXEC, 
  IDQ_ALL_MITE_CYCLES_4_UOPS, 
  IDQ_ALL_MITE_CYCLES_ANY_UOPS, 
  IDQ_ALL_DSB_CYCLES_4_UOPS, 
  IDQ_ALL_DSB_CYCLES_ANY_UOPS, 
  IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE, 
  ARITH.FPU_DIV_ACTIVE, 
  BR_MISP_RETIRED.ALL_BRANCHES_PS, 
  CPU_CLK_UNHALTED.REF_TSC, 
  CPU_CLK_UNHALTED.THREAD, 
  DTLB_LOAD_MISSES.WALK.Duration, 
  ICACHE.MISSES, 
  IDQ_UOPS_NOT_DELIVERED.CORE, 
  INST_RETIRED.ANY, 
  ITLB_MISSES.WALK_DURATION, 
  L1D.REPLACEMENT, 
  L2_LINES_IN.ALL, 
  LD_BLOCKS.STORE_FORWARD, 
  LD_BLOCKS_PARTIAL.ADDRESS_ALIAS, 
  MACHINE_CLEARS.MASKMOV, 
  MACHINE_CLEARS.MEMORY_ORDERING,
MACHINE_CLEARS.SMC;, MEM_LOAD_UOPS.LLC_HIT.RETIRED.XSNP_HIT.PS;, MEM_LOAD_UOPS.LLC_HIT.RETIRED.XSNP_HIT_PS;, MEM_LOAD_UOPS.LLC_MISS.RETIRED.LOCAL_DRAM;, MEM_LOAD_UOPS.RETIRED.LLC_HIT.PS;, MEM_UOPS.RETIRED.ALL_STORES.PS;, MEM_UOPS.RETIRED.SPLIT_LOADS.PS;, MEM_UOPS.RETIRED.SPLIT_STORES.PS;, OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_DRAM.0:, DTLB_LOAD_MISSES.STLB_HIT:, UOPS_ISSUED.ANY:, UOPS_RETIRED.RETIRE_SLOTS:, INT_MISC.RECOVERY_CYCLES:, MEM_LOAD_UOPS.LLC_MISS.RETIRED.REMOTE_DRAM:, MEM_LOAD_UOPS.LLC_MISS.RETIRED.REMOTE_HITM:, MEM_LOAD_UOPS.LLC_MISS.RETIRED.REMOTE_FWD:, CPU_CLK_UNHALTED.THREAD_P, IDQ.MS_SWITCHES, MEM_LOAD_UOPS_RETIRED.HIT_LFB, ITLB_MISSES.STLB_HIT, LSD.CYCLES.ACTIVE, LSD.CYCLES_4_UOPS, MEM_LOAD_UOPS.LLC_HIT.RETIRED.XSNP_MISS_P, OTHER_ASSISTS.ANY.WB_ASSIST, OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO, MEM_UOPS_RETIRED.LOCK_LOADS.PS, UOPS_DISPATCHED.PORT.PORT_0, UOPS_DISPATCHED.PORT.PORT_1, UOPS_DISPATCHED.PORT.PORT_2, UOPS_DISPATCHED.PORT.PORT_3, UOPS_DISPATCHED.PORT.PORT_4, UOPS_DISPATCHED.PORT.PORT_5, OFFCORE_REQUESTS_BUFFER.SQ_FULL
-knob collectMemBandwidth=false
-knob collectMemObjects=false
-knob memoryObjectMinSize=1024
-knob enable-user-tasks=false
-knob enable-system-cswitch=false
-knob enable-gpu-usage=false
-knob gpu-counters-mode=none
-knob gpu-sampling-interval=1
-knob enable-gpu-runtimes=false
-knob collectPreciseClockticks=false
-knob dram-bandwidth-limits=false
-knob analyze-loops=false
-knob mrte-type=java
-knob analyze-openmp=false
-knob event-mode=all
-knob analyze-active-power-consumption=false
-knob enable-context-switches=false
-knob preciseMultiplexing=false
-follow-child
-mrte-mode=auto
-target-duration-type=short
-no-analyze-system
-data-limit=0
| 116 | `-ring-buffer=0`                  |
| 117 | `-no-trace-mpi`                   |

Listing B.3: VTune configuration common options.
Bibliography


[24] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.


