ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ML-driven Automated Framework for Tuning Spark Applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δήμητρα Ν. Νικητοπούλου
Α.Μ. : 03114954

Επιβλέπων:

- Δημήτρης Σούντρης, Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ML-driven Automated Framework for Tuning Spark Applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δήμητρα Ν. Νικητοπούλου
Α.Μ. : 03114954

Επιβλέπων:

- Δημήτρης Σούντρης, Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 20 Ιουλίου 2020.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
....	....	....
Δ. Σούντρης Καθηγητής Ε.Μ.Π	Π. Τσανάκας Καθηγητής Ε.Μ.Π	Γ. Γκούμας Επίκουρος Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2020

(Υπογραφή)

....

Δήμητρα Ν. Νικητοπούλου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright ©- All rights reserved Δήμητρα Ν. Νικητοπούλου, 2020.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σήμερα, ένας ολοένα αυξανόμενος αριθμός από δεδομένα χαρακτηρίζει την εποχή μας, επειδή συλλέγονται με εύκολο και φθηνό τρόπο από διάφορες συσκευές που είναι συνδεδεμένες στο διαδίκτυο. Ο χειρισμός αυτών των δεδομένων απαιτεί πόρους, που παρέχονται εύκολα από το υπολογιστικό νέφος, αλλά και νέα εργαλεία που να επιταχύνουν τη διαδικασία. Προς αυτή την κατεύθυνση, η κατανεμημένη εκτέλεση των διεργασιών και η αξιοποίηση της υψηλής ταχύτητας που προσφέρει η μνήμη έναντι του σκληρού δίσκου είναι υψίστης σημασίας. Το Spark είναι ένα εργαλείο που εκμεταλλεύεται αυτές τις δύο παρατηρήσεις και μπορεί να χειρίζεται εύκολα μεγάλο όγκο δεδομένων. Η βέλτιστη εκτέλεση των προγραμμάτων του, όμως, εξαρτάται σε μεγάλο βαθμό από τη σωστή ρύθμιση μιας σειράς παραμέτρων, οι οποίες μάλιστα είναι πολλές σε αριθμό.

Στην παρούσα εργασία, σχεδιάζουμε ένα σύστημα που εκτελεί αυτόματη ρύθμιση των παραμέτρων του Spark, ανάλογα με την εφαρμογή που εκτελείται και το μέγεθος των δεδομένων εισόδου αυτής. Εντοπίζουμε τις παραμέτρους που έχουν τη μεγαλύτερη επίδραση στην εκτέλεση των εφαρμογών και σκιαγραφούμε τη μεθοδολογία για τη ρύθμισή τους με σκοπό την ελαχιστοποίηση του χρόνου εκτέλεσης. Κατόπιν, ενσωματώνουμε τη λύση μας στο Spark μέσω ενός wrapper script και παρέχουμε στον χρήστη την ικανότητα να τρέχει την εντολή spark-submit που θα έτρεχε, με τη διαφορά ότι η αντίστοιχη εντολή που θα εκτελεστεί στην πραγματικότητα είναι αυτή που χρησιμοποιεί τη βέλτιστη παραμετροποίηση. Τέλος, παρουσιάζουμε την επιτάχυνση της εκτέλεσης που πετύχαμε για ένα σύνολο εφαρμογών, με χρήση των οποίων κατασκευάσαμε το σύστημα, καθώς και άλλων άγνωστων εφαρμογών για να εντοπίσουμε την ικανότητα γενίκευσης της βελτιστοποιητικής ικανότητας της μεθοδολογίας μας.

Λέξεις Κλειδιά – Spark, ρύθμιση παραμέτρων, βελτιστοποίηση, επίδραση παραμέτρων, μοντελοποίηση επίδοσης

Abstract

Nowadays, there is an ever-increasing number of data that characterizes our era, since they are easy and cheap to collect from various devices connected to the Internet. Manipulating big data demands resources, which are provided from the cloud in a convenient way, and some tools to speed up the process. Towards this direction, the distributed execution of processes as well as the exploitation of the high speed that the use of memory has to offer over the hard disk is of utmost importance. Spark is a tool that takes advantage of these remarks and can manipulate easily a vast volume of data. Achieving an optimal execution of its workloads, though, depends to a great extent on the appropriate tuning of a large number of parameters.

In this thesis, we design a framework that tunes in an automated way Spark's parameters, depending on the workload and the size of the input data. We locate the parameters with the greatest impact on the execution of the applications and we form a methodology to tune them accordingly so as to minimize the execution time. Next, we integrate our solution into Spark, with the use of a wrapper script and we provide the user the chance to run a simple spark-submit command but actually executing the one with the optimal configuration. Finally, we present the speedup we achieved for the set of applications we used to construct the framework as well as for other unseen applications in order to evaluate the framework's ability to generalize its optimization capacity.

Keywords – Spark, parameter tuning, optimization, parameter impact, performance modeling

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντά μου, Καθηγητή ΕΜΠ Δημήτριο Σούντρη, ο οποίος με εμπιστεύτηκε και μου έδωσε την ευκαιρία να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων (Microlab) στο ΕΜΠ.

Επίσης, θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή και καθηγητή Σωτήριο Ξύδη και τον υποψήφιο διδάκτορα Δημοσθένη Μασούρο για τη βοήθεια και τη συνεργασία τους καθ'όλη τη διάρκεια της διπλωματικής μου. Η συνεχής επικοινωνία μας κατά τη διάρκεια της διπλωματικής με βοήθησε να αποκτήσω γρηγορότερα γνώσεις σχετικές με το αντικείμενο, οι οποίες είναι εφαρμόσιμες σε πολλούς τομείς της σύγχρονης τεχνολογίας και των συστημάτων υπολογιστών. Παράλληλα με εισήγαγαν στην ερευνητική προσέγγιση κι εργασία. Θα ήθελα, επίσης, να ευχαριστήσω και τον μεταπτυχιακό συνεργάτη του εργαστηρίου Αχιλλέα Τζενετόπουλο που ήταν πάντα παρών και ασχολούταν άμεσα με όποιο δικτυακό πρόβλημα προέκυπτε.

Ακόμα, θα ήθελα να ευχαριστήσω τους γονείς μου Νίκο και Γεωργία, την αδερφή μου Κωνσταντίνα και τους φίλους μου, ιδιαίτερα τον Βλάσση, για την αγάπη τους και τη διαρκή τους υποστήριξη. Τέλος, ένα μεγάλο ευχαριστώ στο αγόρι μου Ανδρέα, ο οποίος με στήριξε με υπομονή σε στιγμές πίεσης και άγχους.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor, Professor of NTUA Dimitrios Sountris, who trusted me and gave me the opportunity to develop my thesis at the Microprocessors and Digital Systems Laboratory (MicroLab) in National Technical University of Athens (NTUA).

In addition, I would like to thank the Post-Doctoral Researcher and Professor Sotirios Xydis and the Ph.D candidate Dimosthenis Masouros for their assistance and cooperation throughout my diploma thesis development. Our continuous collaboration during this thesis helped me gain useful knowledge regarding to the subject we examined, which is applicable to many areas of modern technology and computer engineering. They also introduced me to the research approach and environment. Furthermore, I would like to thank the postgraduate collaborator of the laboratory, Achilleas Tzenetopoulos, who was always present and willing to repair any network problem that would come up.

Last but not least, I would like to thank my parents Nikos and Georgia, my sister Konstantina and my friends, especially Vlassis, for their love and constant support. Finally, a big thank you to my boyfriend Andreas who patiently supported me in times of stress and anxiety.

Contents

List of Figures 15

List of Tables 17

Εκτεταμένη Περίληψη 19

1 Εισαγωγή . . . 19
2 Αυτοματοποιημένη μεθοδολογία για τη ρύθμιση των εφαρμογών του Spark . . . 20
2.1 Πειραματικό στήσιμο κι επεξηγήσεις . . . 21
2.2 ΦΑΣΗ 1: Εξερεύνηση και Περικοπή του χώρου Αναζήτησης των παραμέτρων του Spark . . . 23
2.3 Πρόβλεψη Επίδοσης Παραμετροποιημένων Εφαρμογών του Spark . . . 26
2.3.1 ΦΑΣΗ 2: Ομαδοποίηση των Εφαρμογών . . . 26
2.3.2 ΦΑΣΗ 3: Μοντελοποίηση των Εφαρμογών . . . 27
2.4 ΦΑΣΗ 4: Καθορισμός της Βέλτιστης Παραμετροποίησης Εφαρμογών του Spark . . . 28
3 Πειραματικά Αποτελέσματα και Αξιολόγηση . . . 29
3.1 Αξιολόγηση στα Benchmarks Ανάπτυξης . . . 29
3.2 Αξιολόγηση στα Benchmarks Ελέγχου . . . 30
3.2.1 Ανάθεση Cluster και Ακρίβεια Μοντέλου . . . 30

1 Introduction 33

1.1 Rise of distributed, in-memory computing . . . 33
1.2 Motivation . . . 35
1.3 Objectives and Contributions . . . 36

2 Related Work 39

3 The Apache Spark Framework 43

3.1 Apache Spark Overview and Architecture . . . 43
3.2 Apache Spark Properties . . . 45
3.3 Apache Spark Parameters . . . 46

4 An automated framework for tuning spark applications 47

4.1 Experimental Setup and Specifications . . . 48
4.2 STEP 1: Exploration and Pruning of Spark Parameters Design Space . . . 51
4.2.1 Kruskal Wallis Test . . . 53
4.3 Per-application Impact of important spark parameters . . . 56
4.4 Performance Prediction of tuned spark applications . . . 57

4.4.1 End-to-end 57
4.4.2 STEP 2: Clustering Inference 58
4.4.2.1 Production of Representative Signals per Benchmark 58
4.4.2.2 Kmeans clustering algorithm 58
4.4.3 STEP 3: Model Inference 59
4.4.3.1 Experiments 59
4.4.3.2 Description of Model Inputs and Outputs 60
4.4.3.3 Comparing Algorithms for the Performance Models 61
4.4.3.4 Energy Models 63
4.5 STEP 4: Determining the Optimal Configuration of Spark Applications 64
4.5.1 Comparing Optimization Algorithms 64
4.5.2 Genetic Algorithm 64
4.5.3 Integration of proposed framework with apache spark 66

5 Experimental Results and Evaluation 67
5.1 Exploration on Developing Workloads 67
5.2 Exploration on Unseen Workloads 70
5.2.1 Cluster Assignment and Model Scores 70
5.2.2 Result Analysis and Evaluation 71
5.3 Comparison with a Model Free Optimization Process 74

6 Conclusion and Future Work 77

References 79

Appendices 83

A Low level metric plots to characterize benchmarks 85

B Density plots showing the significance of all spark parameters 89

List of Figures

1 Κατανομή του χρόνου εκτέλεσης για διαφορετικές παραμετροποιήσεις ανά εφαρμογή . . . 20
2 Απεικόνιση της προτεινόμενης μεθοδολογίας για αυτόματη ρύθμιση των παραμέτρων του Spark . . . 21
3 Density plots διαφορετικών παραμέτρων . . . 24
4 Περιγραφή εισόδων και εξόδου των μοντέλων . . . 28
5 Αφομοίωση της προτεινόμενης μεθοδολογίας στο Spark . . . 29
6 Σύγκριση speedups για όλα τα benchmarks ανάπτυξης και μεγέθη δεδομένων εισόδου . . . 30
7 Σύγκριση speedups για όλα τα ζεύγη benchmark-δεδομένα με μη αρνητικό σκορ 32

1.1 Execution time distribution for different spark configurations per benchmark . . . 36

3.1 Apache Spark architecture . . . 44
3.2 YARN architecture . . . 44
3.3 Spark Memory Distribution . . . 45

4.1 Overview of the proposed framework for spark parameters auto-tuning . . . 47
4.2 PR as example of data intensive application . . . 50
4.3 LDA as example of compute intensive application . . . 50
4.4 Content of file spark.conf . . . 52
4.5 Density plots of different parameters . . . 53
4.6 Impact of parameters on the execution of PR with small input size . . . 56
4.7 Time-series clustering of developing applications . . . 59
4.8 Description of features and target output of the model . . . 61
4.9 Prediction of execution time versus real measurement for different input sizes of Kmeans . . . 63
4.10 Description of the wrapper script of our framework . . . 66

5.1 Time optimization for developing benchmarks and tiny size of input data . . . 68
5.2 Time optimization for developing benchmarks and small size of input data . . . 68
5.3 Time optimization for developing benchmarks and large size of input data . . . 69
5.4 Comparing speedups for all the developing benchmarks and inputs . . . 69
5.5 Error distribution illustrating prediction versus real measurement for validating benchmarks . . . 71
5.6 Time optimization for validating benchmarks and tiny size of input data . . . 72
5.7 Time optimization for validating benchmarks and small size of input data . . . 73
5.8 Time optimization for validating benchmarks and large size of input data . . . 73

5.9 Comparing speedups for all the validating benchmarks and inputs with no negative scores 74

List of Tables

1 Πειραματικά benchmarks για ανάπτυξη 22
2 Πειραματικά benchmarks για έλεγχο 22
3 Σημαντικές PCM μετρικές 23
4 Σημαντικές παράμετροι Spark 25
5 Ομαδοποίηση των benchmarks ανάπτυξης 27
6 Ανάθεση cluster στα benchmarks ελέγχου 31
7 Σκορ ανά benchmark ελέγχου 31

2.1 Studies about spark and hadoop applications' performance improvement 41

4.1 Experimented benchmarks for developing 48
4.2 Experimented benchmarks for validating 48
4.3 Important PCM Metrics 49
4.4 Characterization of the benchmarks 51
4.5 Selected Spark Parameters 55
4.6 Important Spark Parameters per benchmark 57
4.7 Clustering developing benchmarks 59
4.8 Model scores of each cluster per algorithm 62
4.9 Model scores of each cluster per energy metric 64

5.1 Comparing default, best measured and best predicted execution time for developing benchmarks 67
5.2 Assigning validating benchmarks to clusters 70
5.3 Score per validating benchmark 71
5.4 Comparing default, best measured and best predicted execution time for validating benchmarks 72
5.5 Comparing our optimization process with the one applied directly on spark . . . 75

Εκτεταμένη Περίληψη

1 Εισαγωγή

Στη σημερινή εποχή, που χαρακτηρίζεται από τον τεράστιο όγκο δεδομένων που έχουμε να αποθηκεύουμε και να επεξεργαζόμαστε καθημερινά, είναι προφανής η σπουδαιότητα εργαλείων όπως το Cloud, αφού επιτρέπει στους χρήστες πρόσβαση σε υλικούς πόρους όποτε το θελήσουν. Για τον χειρισμό των ολοένα αυξανόμενων δεδομένων, ακόμη, είναι χρήσιμος ο χειρισμός τους με κατανεμημένο τρόπο και η αξιοποίηση σε μεγαλύτερο βαθμό της μνήμης RAM έναντι του σκληρού δίσκου ώστε να αυξηθεί η επίδοση. Το σημαντικότερο εργαλείο που πετυχαίνει αυτά τα δύο απαιτούμενα είναι το Apache spark, το οποίο μπορεί να χειρίζεται αρκετά petabytes δεδομένων, κατανεμημένα σε συστάδες από πολλούς πραγματικούς ή και εικονικούς εξυπηρετητές, με εντυπωσιακή ταχύτητα καθώς περιορίζει τις αναγνώσεις κι εγγραφές στο δίσκο κατά την εκτέλεση παράλληλων λειτουργιών.

Το Apache Spark διαθέτει ένα μεγάλο αριθμό παραμέτρων προς ρύθμιση που ξεπερνά τις 100. Όπως φαίνεται στην εικόνα 1, η αλλαγή της παραμετροποίησης του spark μπορεί να επηρεάσει σημαντικά την επίδοση, για ακριβώς τα ίδια δεδομένα εισόδου. Αυτή η επίδραση εξαρτάται από την εφαρμογή και μπορεί να είναι μικρότερη, όπως στην περίπτωση του Bayes, ή και αρκετά μεγάλη όπως στο SVM. Συγκεκριμένα, το SVM έχει μεταβλητότητα που σε καμία περίπτωση δε θεωρείται αμελητέα και ισούται με:

$$variation = \frac{t_{max} - t_{min}}{t_{min}} = 10.25$$

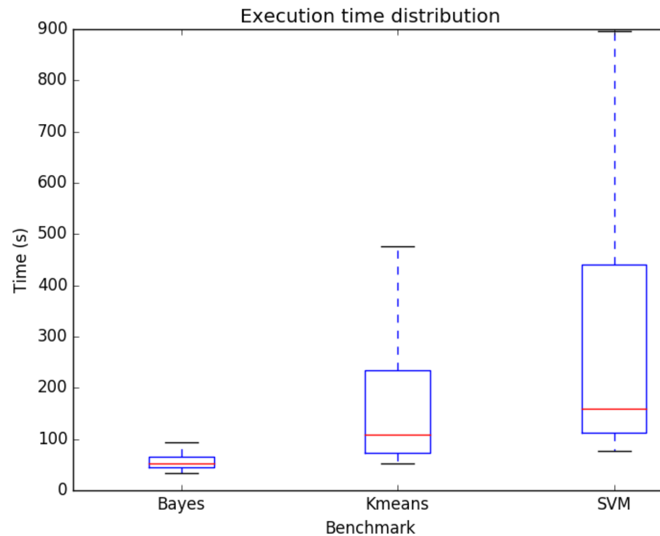


Figure 1: Κατανομή του χρόνου εκτέλεσης για διαφορετικές παραμετροποιήσεις ανά εφαρμογή

Για την επίτευξη της βέλτιστης απόδοσης, λοιπόν, είναι απαραίτητη η ρύθμιση κάποιων παραμέτρων από τον προγραμματιστή, καθώς η προεπιλεγμένη τιμή τους μπορεί να μην είναι ικανή να εκμεταλλευτεί στο έπακρο τις δυνατότητες του συγκεκριμένου συστήματος. Από αυτό γίνεται εμφανής η ανάγκη αυτοματοποίησης της διαδικασίας επιλογής αυτών των τιμών, ώστε να επιτυγχάνεται γρήγορα κι εύκολα, χωρίς μεγάλη ανάμιξη του ανθρώπινου παράγοντα, το καλύτερο δυνατό αποτέλεσμα. Προτείνουμε, λοιπόν, ένα framework που αυτοματοποιεί τη ρύθμιση των παραμέτρων του spark, λαμβάνοντας υπόψη τις ανάγκες τις κάθε εφαρμογής.

Κάποιες παράμετροι, ωστόσο, δεν έχουν μεγάλη επίδραση στην επίδοση των εφαρμογών οπότε είναι σημαντικό να τις ξεχωρίσουμε ώστε να μειωθούν οι διαστάσεις του προβλήματος. Προς αυτή την κατεύθυνση, λοιπόν, το framework μας εντοπίζει τις πιο σημαντικές παραμέτρους με τη βοήθεια του στατιστικού τεστ Kruskal-Wallis και ακολούθως χωρίζει τις εφαρμογές σε ομάδες με βάση κάποια παραπλήσιά τους χαρακτηριστικά σε επίπεδο αρχιτεκτονικής. Έπειτα κατασκευάζει και χρησιμοποιεί μοντέλα μηχανικής μάθησης για την πρόβλεψη του χρόνου εκτέλεσης ορισμένων προγραμμάτων για διαφορετικά μεγέθη αρχείων εισόδου και διαφορετικά διανύσματα τιμών των σημαντικών αυτών παραμέτρων. Με αυτά τα μοντέλα επικοινωνεί ο βελτιστοποιητής ώστε να αποκτήσει γρήγορη πληροφόρηση περί του χρόνου εκτέλεσης και να αξιολογήσει τις διάφορες υποψήφιες λύσεις. Με αυτό τον τρόπο καταλήγει στη λύση που είχε την καλύτερη αξιολόγηση, που πέτυχε δηλαδή τον ελάχιστο χρόνο εκτέλεσης, και δίνει τη βέλτιστη παραμετροποίηση που αφορά το συγκεκριμένο μηχάνημα που διαθέτουμε, για μία εκ των εφαρμογών που μελετήσαμε και για το μέγεθος αρχείου εισόδου που ορίζουμε.

2 Αυτοματοποιημένη μεθοδολογία για τη ρύθμιση των εφαρμογών του Spark

Όπως συζητήθηκε στο προηγούμενο κεφάλαιο, ο καθορισμός των πιο σημαντικών παραμέτρων του Spark είναι μια δύσκολη διαδικασία. Η μεθοδολογία που προτείνουμε βασίζεται στον καθορισμό τους με ένα απόλυτα επιστημονικό τρόπο, χωρίς να ακολουθήσουμε όσα μας υπαγορεύει η λογική και το ένστικτό μας. Με αυτό τον τρόπο στοχεύουμε στην εύρεση ενός

αρκετά ακριβούς τρόπου να βελτιστοποιούμε οποιαδήποτε εφαρμογή.

Συγκεκριμένα η μεθοδολογία μας χωρίζεται σε 4 βασικές φάσεις. Στην πρώτη φάση, που περιγράφεται στον τομέα 2.2, εκτελούμε μια σειρά πειραμάτων και μελετάμε την επίδραση των διαφόρων παραμέτρων ώστε να δούμε ποιες αξίζει να επαναρυθμιστούν. Στη δεύτερη φάση, που περιγράφεται στον τομέα 2.3.1, εκτελούμε μια σειρά πειραμάτων για να δημιουργήσουμε το σύνολο που θα χρησιμοποιήσουμε για την εκπαίδευση των μοντέλων μας κι έπειτα χωρίζουμε τις εφαρμογές που θα χρησιμοποιηθούν για την εκπαίδευση σε ομάδες παρομοίων. Στην τρίτη φάση, που περιγράφεται στον τομέα 2.3.2, ελέγχουμε αρκετούς αλγορίθμους ώστε να κατασκευάσουμε ένα μοντέλο ανά ομάδα που να μπορεί να προβλέψει με ακρίβεια το χρόνο εκτέλεσης μιας εφαρμογής για συγκεκριμένο μέγεθος δεδομένων και παραμετροποίηση. Στην τέταρτη και τελευταία φάση, που περιγράφεται στον τομέα 2.4, τα μοντέλα επικοινωνούν με έναν αλγόριθμο βελτιστοποίησης που τα χρησιμοποιεί για να αξιολογεί τις υποψήφιες παραμετροποιήσεις γρήγορα κι εύκολα, χωρίς να απαιτείται η εκτέλεση πειραμάτων, ώστε να δώσει τη βέλτιστη παραμετροποίηση.

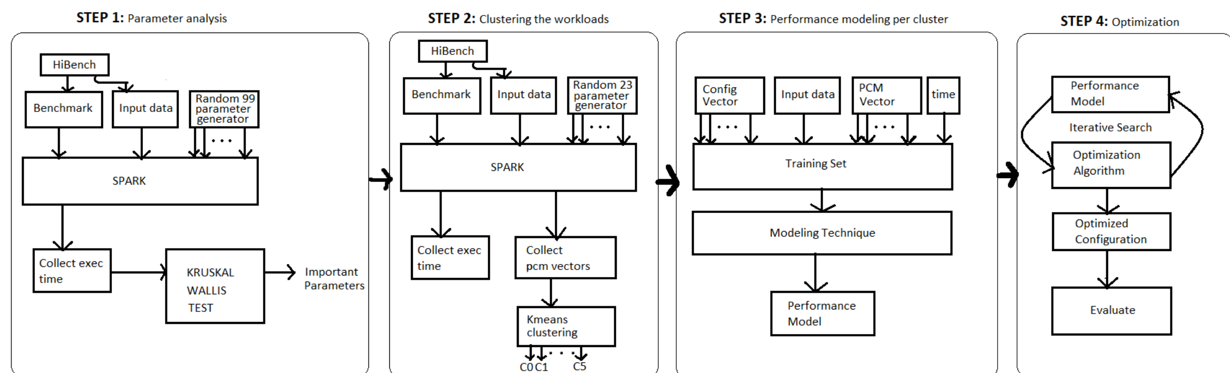


Figure 2: Απεικόνιση της προτεινόμενης μεθοδολογίας για αυτόματη ρύθμιση των παραμέτρων του Spark

2.1 Πειραματικό στήσιμο κι επεξηγήσεις

Όλα τα πειράματα πραγματοποιήθηκαν σε ένα server, με ένα ψευδοκατανεμημένο τρόπο, με χρήση του Hadoop Distributed File System (HDFS) αντί για το τοπικό σύστημα αρχείων. Ο server μας είναι εξοπλισμένος με 48 Intel(R) Xeon(R) CPU E5-2658A v3 2.20GHz 12-πύρηνους επεξεργαστές, 125 GB μνήμη και 1 TB σκληρό δίσκο. Η L1 instruction cache όπως επίσης και η data cache είναι 32 KB, η L2 cache είναι 256 KB και η L3 cache είναι 30.72 MB. Το λειτουργικό σύστημα είναι Linux, έκδοση 4.15.0-101-generic. Η πειραματική Apache Spark έκδοση είναι 1.6.0 και η έκδοση του Hadoop είναι η 2.6.0.

Τα benchmarks που επιλέχθηκαν για την ανάπτυξη της αυτοματοποιημένης μεθοδολογίας μας, όπως επίσης κι αυτά που θα ελέγξουν την ικανότητα του συστήματος να γενικέυει τα συμπεράσματα του με αποτελεσματικό τρόπο ακόμα και σε άγνωστες εφαρμογές, προέρχονται από το HiBench Benchmark Suite. Το HiBench περιέχει διάφορες εφαρμογές όπως αλγορίθμους μηχανικής μάθησης και συναρτήσεις sql. Επιλέγουμε 12 benchmarks για ανάπτυξη, τα οποία παρουσιάζονται στον πίνακα 1, ώστε να έχουμε μεγάλη βάση γνώσεις και 6 για έλεγχο, τα οποία παρουσιάζονται στον πίνακα 2.

		Μέγεθος Δεδομένων για κάθε κατηγορία		
Benchmark	**Συντόμευση**	**Tiny**	**Small**	**Large**
Bayesian Classification	Bayes	93.1 MB	111.7 MB	377.1 MB
K-means clustering	Kmeans	1.3 MB	602.4 MB	4 GB
Support Vector Machine	SVM	8 MB	800.6 MB	20 GB
Aggregate	Aggr	54.3 KB	3.7 MB	37.2 MB
Join	Join	199.2 KB	19.2 MB	191.9 MB
Scan	Scan	206.2 KB	20.1 MB	201 MB
PageRank	PR	10.8 KB	1.8 MB	259.9 MB
Linear Regression	Linear	4 GB	16 GB	48 GB
Gradient Boosting Trees	GBT	11.3 KB	408.4 KB	16MB
Sort	Sort	36 KB	3.2 MB	328.4 MB
Latent Dirichlet Allocation	LDA	21.7 MB	97.5 MB	258.7 MB
TeraSort	TS	3.2 MB	320 MB	3.2 GB

Table 1: Πειραματικά benchmarks για ανάπτυξη

		Μέγεθος Δεδομένων για κάθε κατηγορία		
Benchmark	**Συντόμευση**	**Tiny**	**Small**	**Large**
Alternating Least Squares	ALS	67.4 KB	6 MB	120.6 MB
Logistic Regression	LR	808.4 KB	80 MB	8 GB
Random Forest	RF	11.3 KB	408.4 KB	8 MB
Principal Components Analysis	PCA	88.4 KB	8 MB	32.1 MB
Singular Value Decomposition	SVD	805 KB	16 MB	64 MB
WordCount	WC	36.2 KB	328.4 MB	3.2 GB

Table 2: Πειραματικά benchmarks για έλεγχο

Προφανώς, οι πληροφορίες που είναι χρήσιμες για τον χαρακτηρισμό μιας εφαρμογής αφορούν τη σχετική με τους επεξεργαστές, με το δίκτυο και με τη μνήμη συμπεριφορά. Συγκεκριμένα, αυτό που πρέπει να μελετηθεί είναι το IPC, η συχνότητα, η επίδοση των κρυφών μνημών και οι προσβάσεις στη μνήμη. Το πρόγραμμα PCM (Performance Counter Monitor) είναι ένα πρόγραμμα της intel που τρέχει στο παρασκήνιο κατά το χρόνο εκτέλεσης μιας εφαρμογής και συλλέγει όλες τις παραπάνω πληροφορίες. Συγκεκριμένα, με τη λήξη της εκτέλεσης της εφαρμογής, συλλέγονται σε ένα αρχείο .csv τιμές που λαμβάνονταν ανά 0.1 sec, όπως είχε οριστεί, και αφορούν σε επίπεδο αρχιτεκτονικής τη συμπεριφορά της. Αυτό το αρχείο περιλαμβάνει πολυάριθμες στήλες, που προσφέρουν πληροφορίες πρώτα γενικά για το σύστημα και μετά για συγκεκριμένα sockets και πυρήνες. Οι πληροφορίες του αρχείου που αξιολογήθηκαν ως μεγαλύτερης σημασίας για αυτό που θέλουμε να πετύχουμε είναι αυτές που αφορούν το σύστημα και είναι οι εξής:

PCM Μετρική	**Περιγραφή**
PhysIPC%	Πλήθος εντολών ανά κύκλο ρολογιού (IPC), πολλαπλασιασμένο με τον αριθμό των νημάτων κάθε πυρήνα (2 threads/core) και διαιρεμένο με το μέγιστο δυνατό IPC (4) και ξανά πολλαπλασιασμένο με 100%
TotalQPIout	Η εκτίμηση κίνησης στο QuickPath Interconnect (QPI), δηλαδή το σύνολο των δεδομένων σε MB που βγαίνουν από τους πυρήνες ή τα sockets μέσω των συνδέσεων του δικτύου διασύνδεσης QPI
READ	Οι αναγνώσεις από τη μνήμη που έγιναν σε GB
WRITE	Οι εγγραφές στη μνήμη που έγιναν σε GB
AFREQ	Η συχνότητα διαιρεμένη με την ονομαστική συχνότητα του επεξεργαστή (2.10 GHz), εξαιρώντας το χρόνο που ο επεξεργαστής κοιμάται
L3MISS	Το πλήθος των misses στην cache επιπέδου 3 μετρημένο σε εκατομμύρια
L2MISS	Το πλήθος των misses στην cache επιπέδου 2 μετρημένο σε εκατομμύρια
L3HIT	Το ποσοστό των hits στην cache επιπέδου 3
L2HIT	Το ποσοστό των hits στην cache επιπέδου 2

Table 3: Σημαντικές PCM μετρικές

2.2 ΦΑΣΗ 1: Εξερεύνηση και Περικοπή του χώρου Αναζήτησης των παραμέτρων του Spark

Σε αυτή τη φάση η μεθοδολογία που ορίσαμε προβλέπει το ψαλίδισμα του χώρου αναζήτησης με τον καθορισμό των παραμέτρων που έχουν τη μεγαλύτερη επίδραση στην επίδοση των εφαρμογών, ώστε να κρατηθούν μόνο αυτές. Αρχικά, έγινε μία πρώτη εκκαθάριση των παραμέτρων και αφαιρέθηκαν κάποιες που αφορούσαν απλά ονομασίες, διαδρομές στο σύστημα αρχείων και άλλες περιττές πληροφορίες που δε θα χρησίμευαν στο να γίνει το σύστημα πιο αποδοτικό. Επελέγησαν με αυτό τον τρόπο 99 παράμετροι που θα μπορούσαν να θεωρηθούν σημαντικές, και ακολούθησε περαιτέρω ανάλυσή τους προκειμένου να διαπιστωθεί η επίδρασή τους στο χρόνο εκτέλεσης των πειραμάτων και κατ'επέκταση η επίδρασή τους (impact) στις εφαρμογές.

Η προτεινόμενη μεθοδολογία μας στηρίζεται στον OpenTuner, όχι μόνο κατά τη διαδικασία της βελτιστοποίησης αλλά και για την παραγωγή τυχαίων παραμετροποιήσεων. Ο OpenTuner είναι ένα εργαλείο που μπορεί να χρησιμοποιηθεί για αυτόματη ρύθμιση προγραμμάτων, δηλαδή να δοκιμάζει διάφορες τιμές για τις παραμέτρους που αυτά προσφέρουν και να καταλήγει σε αυτές που θα κάνουν την εκτέλεση του προγράμματος πιο αποδοτική, με βάση κάποια μετρική που έχουμε ορίσει. Σε αυτό το στάδιο, όμως, δεν ορίζουμε κάποια μετρική αλλά αντιθέτως βγαίνουμε από τη διεργασία και την επανακαλούμε πολλές φορές, παράγοντας τυχαία διανύσματα που δεν αξιολογούνται με κάποιο τρόπο.

Προκειμένου να γίνει η ανάλυση αυτή, μελετήθηκε πολύ καλά το εύρος τιμών κάθε παραμέτρου κι επελέγησαν τρεις αντιπροσωπευτικές τιμές, η προεπιλεγμένη, μία μεγαλύτερη και μία μικρότερη όταν επρόκειτο για αριθμητικές τιμές ενώ για τις υπόλοιπες περιπτώσεις τοποθετήθηκαν απλά οι εναλλακτικές τιμές. Στη συνέχεια, παρήχθησαν 300 τυχαία διανύσματα με διάφορες τιμές σε κάθε παράμετρο - μεταξύ αυτών που είχαμε ορίσει - με τη βοήθεια του open-

Tuner. Τα διανύσματα αυτά είχαν τη μορφή:

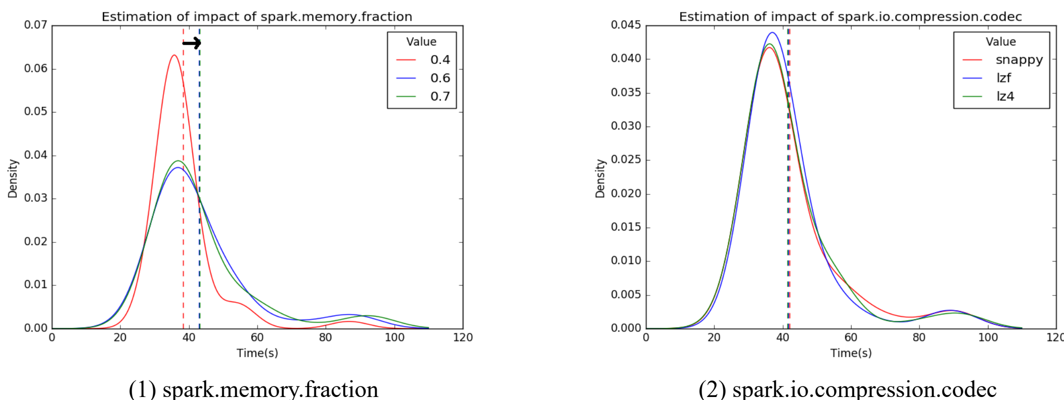
$$conf_i = \{c_{i1}, c_{i2}, .., c_{ij}, .., c_{i99}\}, 1 \leqslant i \leqslant 300$$

, όπου το $conf_i$ είναι η i-οστή παραμετροποίηση και το c_{ij} είναι η τιμή της j-οστής παραμέτρου στην i-οστή παραμετροποίηση.
Εκτελέστηκε, μάλιστα, μία σύντομη ανάλυση των διανυσμάτων για να επιβεβαιωθεί ότι είναι επαρκώς διαφορετικά μεταξύ τους και αποδείχτηκε ότι όλα τα διανύσματα ανά 2 διέφεραν μεταξύ τους σε περισσότερες από 40 παραμέτρους.

Μετά την παραγωγή των διαφορετικών παραμετροποιήσεων, η μεθοδολογία μας προβλέπει την εκτέλεση μιας σειράς πειραμάτων. Συγκεκριμένα, εκτελέστηκαν τα 7 πρώτα benchmarks του πίνακα 1 για tiny, small και large μεγέθη δεδομένων εισόδου και για τις 300 διαφορετικές παραμεροποιήσεις. Τα αποτελέσματα των παραπάνω πειραμάτων ήταν ένα σύνολο από τους αντίστοιχους χρόνους εκτέλεσης $t = \{t_1, t_2, .., t_i, .., t_{300}\}$. Για να διαπιστωθεί η σημασία μιάς παραμέτρου έγινε ταξινόμηση των χρόνων εκτέλεσης με βάση την τιμή που είχε η δεδομένη παράμετρος στην εν λόγω παραμετροποίηση, δημιουργώντας ισάριθμα διανύσματα με το πλήθος των αντιπροσωπευτικών τιμών που είχαν επιλεγεί.

Στη συνέχεια, αυτά τα διανύσματα τροφοδοτούνται στη συνάρτηση που εκτελεί το τεστ Kruskal-Wallis για να διαπιστωθεί αν οι μέσοι όροι τους διαφέρουν σημαντικά. Δε χρησιμοποιήθηκε ANOVA τεστ γιατί τα δεδομένα ανήκουν σε κατανομές που βασίζονται σε πειραματικά δεδομένα και δεν μπορούν να θεωρηθούν κανονικές. Για να αξιολογηθεί το αποτέλεσμα του τεστ, πότε δηλαδή η αρχική υπόθεση επιβεβαιώνεται, τίθεται ένα όριο αποδοχής pvalue% ώστε να εξασφαλιστεί ότι η πιθανότητα να μετρηθούν διαφορετικοί μέσοι όροι όταν οι κατανομές είναι ίσες είναι κάτω του 5%. Όταν, λοιπόν, το pvalue που επιστρέφεται είναι μεγαλύτερο από 0.05 τότε η αρχική υπόθεση επιβεβαιώνεται και δεν υπάρχει σημαντική διαφορά στους μέσους όρους, ενώ αν είναι μικρότερο συμβαίνει το αντίθετο.

Τα density plots του παρακάτω σχήματος απεικονίζουν δύο αντιπροσωπευτικά παραδείγματα, παρουσιάζοντας τη διαδικασία που ακολουθήθηκε για τον καθορισμό της σημασίας κάθε παραμέτρου. Συγκεκριμένα, δείχνουν την κατανομή του χρόνου εκτέλεσης για 3 διαφορετικές τιμές που λαμβάνουν οι παράμετροι. Το πρώτο διάγραμμα απεικονίζει μία σημαντική παράμετρο κι αυτό γίνεται εμφανές αφού για την τιμή 0.4 έχουμε σημαντική μετακίνηση της κατανομής προς τα αριστερά. Το δεύτερο διάγραμμα αντιστοιχεί σε μη σημαντική παράμετρο.



(1) spark.memory.fraction (2) spark.io.compression.codec

Figure 3: Density plots διαφορετικών παραμέτρων

Παράμετρος	**Περιγραφή**
spark.executor.instances	Το πλήθος των executors
spark.executor.cores	Το πλήθος των πυρήνων που χρησιμοποιεί κάθε executor
spark.task.cpus	Το πλήθος των πυρήνων που δεσμεύονται για κάθε task
spark.shuffle.compress	Το κατά πόσο θα συμπιεστούν τα αρχεία εξόδου κατά τη διαδικασία του map
spark.executor.memory	Το μέγεθος της μνήμης που χρησιμοποιείται ανά executor
spark.memory.fraction	Το ποσοστό του σωρού, αφού του αφαιρέσουμε τη δεσμευμένη για το σύστημα και για το αντικείμενο του spark μνήμη των 300 MB, που θα χρησιμοποιηθεί είτε για εκτέλεση είτε για αποθήκευση
spark.memory.storageFraction	Το ποσοστό της μνήμης που έχει διατεθεί για εκτέλεση ή αποθήκευση που θα χρησιμοποιηθεί συγκεκριμένα για αποθήκευση
spark.serializer	Η κλάση που χρησιμοποιείται για σειριοποίηση των αντικειμένων που θα σταλούν εντός του δικτύου ή που χρειάζεται να αποθηκευτούν σε σειριοποιημένη μορφή
spark.scheduler.maxRegisteredResourcesWaitingTime	Ο μέγιστος χρόνος αναμονής για καταχώρηση πόρων μέχρι να ξεκινήσει η χρονοδρομολόγηση
spark.default.parallelism	Το προεπιλεγμένο πλήθος διαμερίσεων (partitions) στα RDDs που επιστρέφεται από μετασχηματισμούς όπως join, reduce-ByKey και parallelize
spark.sql.shuffle.partitions	Το πλήθος των διαμερίσεων που χρησιμοποιούνται κατά το ανακάτεμα δεδομένων για ενώσεις και αθροίσεις
spark.cleaner.periodicGC.interval	Το πόσο συχνά εκτελείται συλλογή σκουπιδιών
spark.io.compression.lz4.blockSize	Το μέγεθος του μπλοκ σε bytes που χρησιμοποιείται κατά τη συμπίεση με LZ4 που είναι και το προεπιλεγμένο
spark.yarn.am.memory	Το μέγεθος της μνήμης που χρησιμοποιείται για τον YARN Application Master κατά την εκτέλεση σε client mode
spark.scheduler.revive.interval	Η διάρκεια του διαστήματος για τον χρονοδρομολογητή να επαναφέρει τις προσφορές πόρων για τον εργάτη (worker) να εκτελέσει τα tasks
spark.locality.wait.process	Ρυθμίζει το χρόνο αναμονής για την τοπικότητα των διεργασιών. Επηρεάζει tasks που προσπαθούν να προσπελάσουν δεδομένα που είναι προσωρινά αποθηκευμένα σε μιά συγκεκριμένη διεργασία ενός executor
spark.shuffle.sort.bypassMergeThreshold	Ορίζει ένα κατώφλι ώστε να αποφεύγεται η ταξινόμηση μέσω συγχώνευσης αν υπάρχουν λιγότερες διαμερίσεις από αυτές που ορίζει αυτό για το reduce καθώς και αν δεν υπάρχει άθροιση από την πλευρά του map
spark.shuffle.io.preferDirectBufs	Το κατά πόσο χρησιμοποιούνται buffers εκτός σωρού προκειμένου να μειωθεί η συλλογή σκουπιδιών κατά τη διάρκεια του ανακατέματος δεδομένων και της μετακίνησης cache blocks
spark.task.maxFailures	Το πλήθος των αποτυχιών για ένα συγκεκριμένο task μέχρι να εγκαταλειφθεί η εργασία
spark.files.openCostInBytes	Το εκτιμώμενο κόστος για το άνοιγμα αρχείου, μετρημένο με τον αριθμό των bytes που μπορούν να διαβαστούν παράλληλα
spark.shuffle.file.buffer	Το μέγεθος του εντός μνήμης buffer για κάθε ανακάτεμα σε αρχεία
spark.cleaner.referenceTracking.blocking	Το κατά πόσο το νήμα που καθαρίζει πρέπει να μπλοκάρει στα tasks καθαρίσματος εκτός του shuffle που ελέγχεται από άλλη παράμετρο
spark.kryoserializer.buffer.max	Το μέγιστο επιτρεπτό μέγεθος buffer κατά τη σειριοποίηση με Kryo serializer

Table 4: Σημαντικές παράμετροι Spark

Τα αποτελέσματα του τέστ δεν έδειξαν προφανώς σε όλα τα benchmarks ακριβώς τις ίδιες σημαντικές παραμέτρους. Κάποιες, όπως spark.executor.memory, spark.task.cpus, hibench.yarn.executor.cores, αξιολογήθηκαν σε όλες τις περιπτώσεις ως σημαντικές, ενώ για άλλες το αποτέλεσμα εξαρτήθηκε από τα χαρακτηριστικά του προγράμματος ή ακόμα και από το μέγεθος των δεδομένων εισόδου. Τελικά κρίθηκε ότι 23 παράμετροι είναι συνολικά οι πιο σημαντικές για την εκτέλεση όλων των πειραμάτων μας. Αυτές παρουσιάζονται στον πίνακα 4.

2.3 Πρόβλεψη Επίδοσης Παραμετροποιημένων Εφαρμογών του Spark

Προκειμένου να βρεθεί η βέλτιστη παραμετροποίηση μιας εφαρμογής, είναι απαραίτητο να δοκιμάζονται πολλές υποψήφιες λύσεις και να αξιολογούνται με βάση μια μετρική. Πρέπει, λοιπόν, να κατασκευαστεί ένα μοντέλο που θα ενημερώνει στιγμιαία τον βελτιστοποιητή για το χρόνο εκτέλεσης μιας υποψήφιας λύσης. Οι εφαρμογές ανάπτυξης του πίνακα 1 όμως που θα χρησιμοποιηθούν διαφέρουν σημαντικά μεταξύ τους σε διάφορες μετρικές χαμηλού επιπέδου. Για αυτό το λόγο θα χωριστούν σε ομάδες παραπλήσιων εφαρμογών και θα υλοποιηθεί ένα μοντέλο πρόβλεψης ανά ομάδα.

2.3.1 ΦΑΣΗ 2: Ομαδοποίηση των Εφαρμογών

Όπως έχει ήδη αναφερθεί, οι πληροφορίες που είναι χρήσιμες για τον χαρακτηρισμό μιας εφαρμογής παρουσιάζονται στον πίνακα 3. Συνεπώς, για να ομαδοποιηθούν οι εφαρμογές εκτελούνται παράλληλα με το PCM πρόγραμμα για την προεπιλεγμένη παραμετροποίηση και μεγάλο μέγεθος δεδομένων εισόδου. Για κάθε εκτέλεση παράγεται ένα csv αρχείο το (το οποίο θα καλούμε PCM trace). Κάθε στήλη αυτού του αρχείου αντιστοιχεί σε μία μετρική και μπορεί να θεωρηθεί ένα σήμα αυτής της μετρικής που μεταβάλλεται με το χρόνο. Ο χωρισμός των εφαρμογών σε ομάδες έγινε με την παραγωγή ενός νέου σύνθετου σήματος που υπολογίζεται ως ο γεωμετρικός μέσος των αρχικών σημάτων ανά κελί. Συνεπώς τα νέα σήματα έχουν το ίδιο μήκος με τα παλιά και τα κελιά τους περιέχουν το γεωμετρικό μέσο των αντίστοιχων κελιών των αρχικών:

$$new = \{geo_t1, geo_t2, ..., geo_tn\}$$

, όπου geo ισούται με $\sqrt[9]{PhysIPC\% \cdot TotalQPIout \cdot ... \cdot L2HIT}$

Αυτά τα νέα σήματα που δημιουργήθηκαν θεωρούμε ότι είναι αντιπροσωπευτικά των benchmarks και δηλωτικά του χρόνου εκτέλεσής τους. Συνεπώς, μπορούν να χρησιμοποιηθούν σε έναν αλγόριθμο συσταδοποίησης για να βρεθούν ομοιότητες μεταξύ αυτών και να επιτευχθεί ο χωρισμός τους σε ομάδες (clusters).

Εφόσον ένα σήμα αποτελεί μία χρονική σειρά, ο αλγόριθμος που χρησιμοποιήθηκε είναι ο TimeSeriesKMeans της βιβλιοθήκης tslearn της Python και η εκπαίδευση έγινε πάνω στο time series dataset που περιλαμβάνει τα 12 νέα σήματα με τους γεωμετρικούς μέσους ανά χρονική στιγμή κάθε εφαρμογής. Επειδή, όμως, δεν είχαν τρέξει όλα τα benchmarks για το ίδιο χρονικό διάστημα, δεν έχουν και το ίδιο μήκος. Με τη βοήθεια της βιβλιοθήκης tslearn της Python για την ανάλυση των time series, πάλι, εκτελούμε επαναδειγματοληψία (resampling) στα σήματα και τα μετασχηματίζουμε όλα να έχουν μήκος 2000 τιμών. Αφού δοκιμάζονται διάφορες τιμές για τον αριθμό των clusters καθώς και για τη μετρική που θα χρησιμοποιηθεί για τον υπολογισμό των αποστάσεων (euclidean, dtw, softdtw), καταλήγουμε ότι η πιο αντιπροσωπευτική ομαδοποίηση των εφαρμογών είναι αυτή που προκύπτει από τη μετρική softdtw με την παράμετρο γάμμα ίση με 0.005 και 6 clusters:

No Cluster	Benchmarks
Cluster 0	Aggr, Scan
Cluster 1	Bayes,Linear,Sort
Cluster 2	Kmeans, SVM
Cluster 3	Join,GBT,LDA
Cluster 4	PR
Cluster 5	TS

Table 5: Ομαδοποίηση των benchmarks ανάπτυξης

2.3.2 ΦΑΣΗ 3: Μοντελοποίηση των Εφαρμογών

Προκειμένου να κατασκευαστεί ένα training set ικανό να εκπαιδεύσει ένα μοντέλο που να προβλέπει με επιτυχία τη συμπεριφορά των εφαρμογών σε τυχαίες παραμετροποιήσεις, ακολούθησε μία σειρά εκτεταμένων πειραμάτων που αναφέρονται στα 12 διαφορετικά benchmarks που ομοδοποιήθηκαν προηγουμένως. Από το σύνολο των παραπάνω πειραμάτων, κατά τα γνωστά, κρατήθηκε ο χρόνος εκτέλεσης.

Προκειμένου να γίνει εφικτή η αποτελεσματική πρόβλεψη των χρόνων εκτέλεσης των 12 benchmarks κατασκευάζεται ένα μοντέλο πρόβλεψης ανά cluster, όπως τονίσαμε και προηγουμένως. Κάθε μοντέλο πρόβλεψης εκπαιδεύεται ώστε να μαθαίνει καλά τη συμπεριφορά των εφαρμογών που ανήκουν στη συγκεκριμένη ομάδα, με την ελπίδα ότι θα μπορεί να γενικεύσει τη γνώση του για να προβλέψει και τις νέες άγνωστες σε αυτό εφαρμογές ελέγχου.

Το training set του καθενός θα αποτελείται από τις παραμετροποιήσεις που χρησιμοποιήθηκαν στις τρεις προηγούμενες σειρές πειραμάτων, το μέγεθος των δεδομένων εισόδου και κάποιες πληροφορίες από το default pcm trace, ενώ το response vector θα είναι οι αντίστοιχοι χρόνοι εκτέλεσης. Τα features του training set που αφορούν το default pcm trace είναι 9 κι υπολογίζονται από το μέσο όρο του κάθε σήματος/μετρικής του pcm. Αντί να χρησιμοποιούμε time series, δηλαδή, κρατάμε μοναδική τιμή για το κάθε σήμα, η οποία υπολογίζεται ως ο μέσος όρος όλων των τιμών του, χωρίς resampling σε μήκος 2000. Αυτά τα επιπλέον features είναι απαραίτητα ώστε να διαφοροποιούνται οι προβλέψεις μεταξύ των διαφορετικών benchmarks που έχουν ανατεθεί στο ίδιο cluster. Το σχήμα 4 απεικονίζει τις εισόδους και την έξοδο των μοντέλων.

Ο αλγόριθμος που επιλέχθηκε για την υλοποίηση των μοντέλων είναι ο Random Forest, ο οποίος αποδείχθηκε να υπερέχει έναντι πολλών άλλων regression algorithms της βιβλιοθήκης sklearn της python. Η επίδοση των μοντέλων, μάλιστα, είναι ικανοποιητικά καλή καθώς σε καμία περίπτωση δεν πέφτει κάτω από το 0.73, ενώ γενικά κινείται σε ακρίβεια μεγαλύτερη του 0.85.

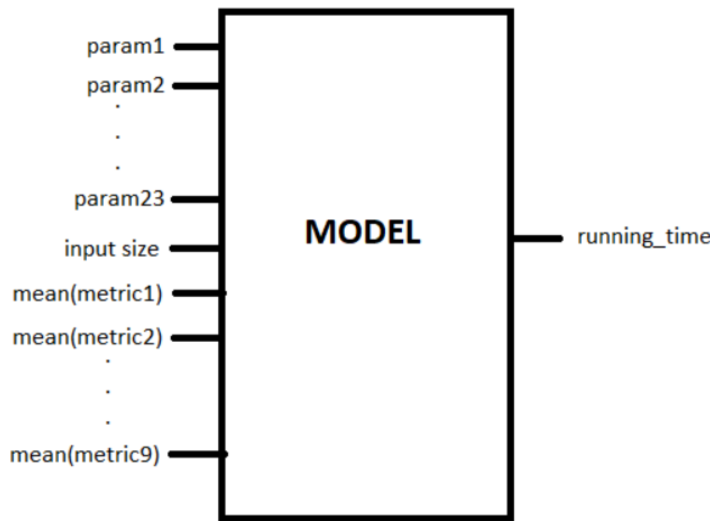


Figure 4: Περιγραφή εισόδων και εξόδου των μοντέλων

2.4 ΦΑΣΗ 4: Καθορισμός της Βέλτιστης Παραμετροποίησης Εφαρμογών του Spark

Ο βασικός στόχος αυτής της μεθοδολογίας εξ αρχής ήταν η εύρεση της βέλτιστης παραμετροποίησης για την εκτέλεση μιας συγκεκριμένης εφαρμογής στο spark. Προφανώς όλοι οι αλγόριθμοι βελτιστοποίησης ακολουθούν την ίδια γενική πορεία. Δοκιμάζουν, δηλαδή, διαφορετικές παραμετροποιήσεις, κινούμενοι με ξεχωριστούς τρόπους στο χώρο αναζήτησης, και σημειώνουν τα αποτελέσματα για να καταλήξουν στο βέλτιστο.

Η διαδικασία της βελτιστοποίησης μπορεί να γίνει εύκολα με χρήση του OpenTuner. Αυτός προσφέρει πλήθος τεχνικών που διαφέρουν στον τρόπο διάσχισης του χώρου αναζήτησης. Καμία από αυτές δεν εγγυάται ότι θα βρει ένα ολικό ελάχιστο του χρόνου εκτέλεσης, αλλά ένα ικανοποιητικό τοπικό ελάχιστο. Είναι, επίσης, δυνατό να δημιουργηθούν μετατεχνικές που αποτελούνται από ένα σύνολο επιμέρους τεχνικών, σε καθεμιά από τις οποίες διαμοιράζεται ένα μέρος του προβλήματος. Η πιο γνωστή μετατεχνική είναι η multi-armed bandit with sliding window, area under the curve credit assignment (AUC Bandit). Οι διάφορες τεχνικές αναπτύσσουν ένα εύρος στρατηγικών και είναι φτιαγμένες ώστε να λειτουργούν βέλτιστα σε διαφορετικούς τύπους χώρων αναζήτησης, άρα δεν υπάρχει κάποια που να υπερέχει των άλλων σε όλα τα προβλήματα.

Η τεχνική που επιλέχθηκε στην προκειμένη περίπτωση είναι οι γενετικοί αλγόριθμοι. Αυτοί διατηρούν έναν πληθυσμό πιθανών λύσεων, πραγματοποιούν αναζήτηση σε πολλές κατευθύνσεις και υποστηρίζουν καταγραφή και ανταλλαγή πληροφοριών μεταξύ αυτών των κατευθύνσεων. Ο πληθυσμός υφίσταται μια προσομοιωμένη γενετική εξέλιξη χρησιμοποιώντας διάφορους γενετικούς τελεστές όπως η επιλογή, η διασταύρωση και η μετάλλαξη.

Είναι, επίσης, αρκετά απλοί στην υλοποίησή τους. Αρχικά, ο Γενετικός Αλγόριθμος παράγει πολλαπλά αντίγραφα της μεταβλητής/γεννητικού κώδικα, συνήθως με τυχαίες τιμές, δημιουργώντας ένα πληθυσμό λύσεων. Κάθε λύση (τιμές για τις παραμέτρους του συστήματος) δοκιμάζεται για το πόσο κοντά φέρνει την αντίδραση του συστήματος στην επιθυμητή, μέσω μιας συνάρτησης που δίνει το μέτρο ικανότητας της λύσης και η οποία ονομάζεται συνάρτηση ικανότητας (Σ.Ι). Οι λύσεις που βρίσκονται πιο κοντά στην επιθυμητή, σε σχέση με τις άλλες,

σύμφωνα με το μέτρο που μας δίνει η Σ.Ι, αναπαράγονται στην επόμενη γενιά λύσεων και λαμβάνουν μια τυχαία μετάλλαξη. Επαναλαμβάνοντας αυτή τη διαδικασία για αρκετές γενιές, οι τυχαίες μεταλλάξεις σε συνδυασμό με την επιβίωση και αναπαραγωγή των γονιδίων/λύσεων που πλησιάζουν καλύτερα το επιθυμητό αποτέλεσμα θα παράγουν ένα γονίδιο/λύση που θα περιέχει τις τιμές για τις παραμέτρους που ικανοποιούν όσο καλύτερα γίνεται την Σ.Ι. Ο αλγόριθμος τερματίζει είτε όταν παραχθεί κάποιος μέγιστος αριθμός γενεών είτε όταν επιτευχθεί κάποιο ικανοποιητικό αποτέλεσμα από τη Σ.Ι. για τον πληθυσμό.

Συγκεκριμένα από όλους τους γενετικούς αλγορίθμους του OpenTuner αυτός που επιλέχθηκε είναι ο ga-PX. Για την εκτέλεση του βελτιστοποιητή, ορίστηκε πάλι ένας configuration manipulator, όπως θέλει ο OpenTuner, ο οποίος όρισε τον χώρο αναζήτησης του προβλήματος. Στη συνέχεια, η συνάρτηση run ορίστηκε με τέτοιο τρόπο ώστε να ρωτά το αντίστοιχο μοντέλο για το χρόνο εκτέλεσης του ζεύγους εφαρμογή-δεδομένα για την εν λόγω παραμετροποίηση, ώστε να την αξιολογήσει στη συνέχεια και να καταλήξει στην καλύτερη δυνατή. Συνολικά, η προτεινόμενη μεθοδολογία μας παίρνει την κλασική spark-submit εντολή και από πίσω τρέχει ολη τη διαδικασία που απεικονίζεται στο σχήμα 5, καταλήγοντας στην εκτέλεση της εφαρμογής με τη βέλτιστη παραμετροποίηση.

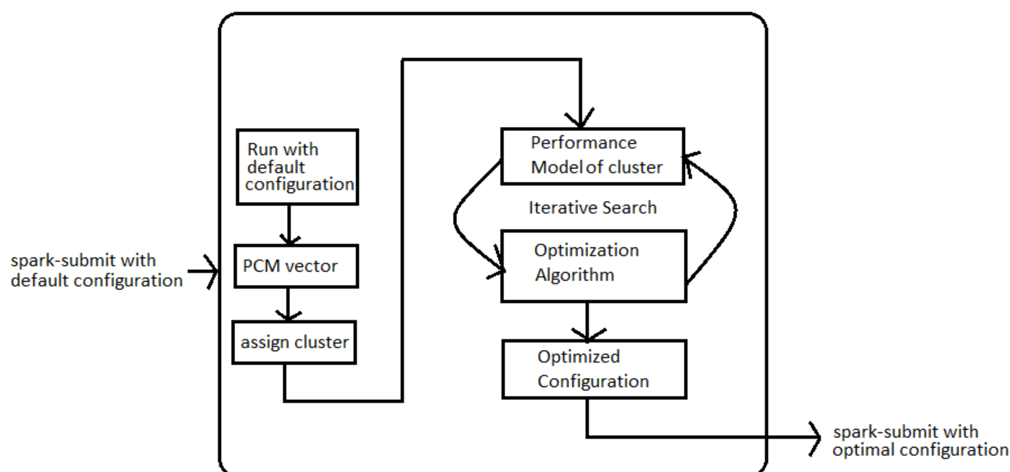


Figure 5: Αφομοίωση της προτεινόμενης μεθοδολογίας στο Spark

3 Πειραματικά Αποτελέσματα και Αξιολόγηση

3.1 Αξιολόγηση στα Benchmarks Ανάπτυξης

Μετά από εφαρμογή της μεθοδολογίας μας σε όλα τα ζεύγη benchmark-μέγεθος δεδομένων συλλέχθηκαν οι βέλτιστοι χρόνοι που προβλέφθηκαν καθώς και οι παραμετροποιήσεις με τις οποίες επιτεύχθηκαν αυτοί. Στη συνέχεια, ακολούθησε πραγματική εκτέλεση στο ίδιο σύστημα όλων των παραπάνω ζευγών με τις βέλτιστες παραμετροποιήσεις για να διαπιστωθεί πόσο καλύτερο ήταν το αποτέλεσμα από το αντίστοιχο με την προεπιλεγμένη παραμετροποίηση αλλά και για να αξιολογηθεί η ακρίβεια των μοντέλων μας.

Τα αποτελέσματα έδειξαν πολύ καλή ακρίβεια στις προβλέψεις, εκτός από κάποιες περιπτώσεις

εφαρμογών που αντιστοιχούν στο τρίτο cluster, καθώς εκεί εντοπιζόταν κάποιος θόρυβος. Το σχήμα 6 παρουσιάζει το speedup που επιτυγχάνεται, δηλαδή το κλάσμα της εκτέλεσης με την προεπιλεγμένη παραμετροποίηση που αποτελεί η δική μας βέλτιστη εκτέλεση κατόπιν μετρήσεως.

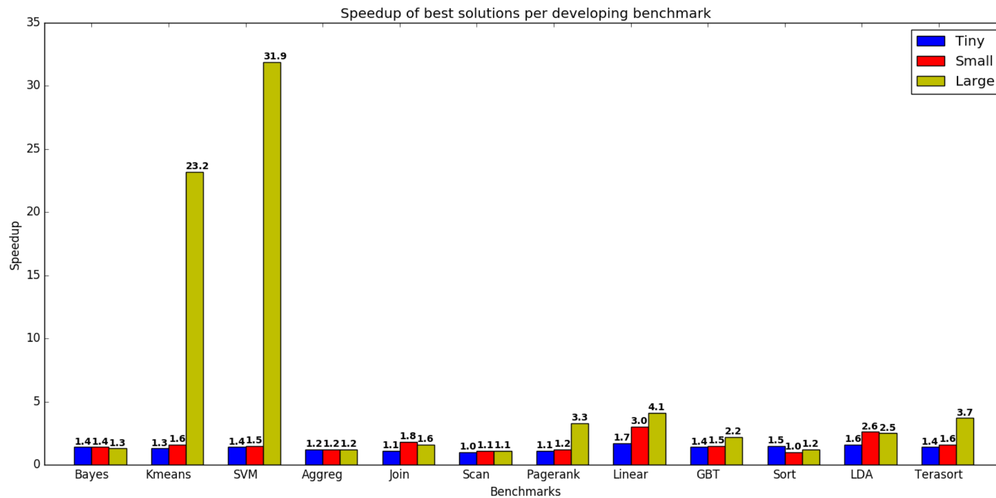


Figure 6: Σύγκριση speedups για όλα τα benchmarks ανάπτυξης και μεγέθη δεδομένων εισόδου

Παρατηρούμε ότι ο χρόνος εκτέλεσης μειώθηκε για όλα τα ζεύγη εφαρμογή-δεδομένα με τη χρήση της παραμετροποίησης που επέλεξε η προτεινόμενη μεθοδολογία μας. Ακόμη, όλες οι εφαρμογές βελτιώνονται σημαντικά, πλην του Scan που πετυχαίνει οριακή βελτίωση. Το μέγιστο speedup πετυχαίνει $31.9\times$ όταν το SVM εκτελείται με 20 GB μέγεθος δεδομένων. Αυτό είναι λογικό μιας και πρόκειται για εφαρμογή που είναι και compute και data intensive και συνεπώς υπάρχει πλήθος παραμέτρων που μπορούν να επηρεάσουν σημαντικα την εκτέλεσή της. Το μέσο speedup για όλα τα ζεύγη εφαρμογών-δεδομένων είναι $3.07\times$. Τέλος, είναι εμφανές ότι η βελτίωση που πετυχαίνουμε κατά τη γενική περίπτωση αυξάνεται όσο αυξάνεται και το μέγεθος των δεδομένων, γεγονός υψίστης σημασίας για την εποχή μας που χαρακτηρίζεται από ολοένα αυξανόμενα δεδομένα.

3.2 Αξιολόγηση στα Benchmarks Ελέγχου

Η μεγάλη καινοτομία του έργου μας είναι ότι μπορεί επίσης να βελτιστοποιεί την εκτέλεση μιας άγνωστης εφαρμογής που δεν έχει χρησιμοποιηθεί στη διαδικασία της εκπαίδευσης των μοντέλων. Αυτός είναι, άλλωστε, ο λόγος που αφιερώσαμε χρόνο και προσπάθεια στην εύρεση τηε σύνδεσης μεταξύ των βασικών χαρακτηριστικών μιας εφαρμογής σε επίπεδο αρχιτεκτονικής και του χρόνου που απαιτείται για την εκτέλεσή της.

3.2.1 Ανάθεση Cluster και Ακρίβεια Μοντέλου

Για να αξιολογήσουμε τη μεθοδολογία μας πάνω σε μια άγνωστη εφαρμογή που συναντάμε για πρώτη φορά και δεν έχει χρησιμοποιηθεί στη διαδικασία της εκπαίδευσης του μοντέλου πρέπει πρώτα να την εκτελέσουμε μια φορά ώστε κατά τα γνωστά να συλλέξουμε το PCM trace. Με τον ίδιο τρόπο που περιγράφηκε πριν, παράγονται νέα σήματα για καθένα εκ των benchmarks ελέγχου, με χρήση του γεωμετρικού μέσου. Αφού φορτωθεί το μοντέλο του clustering

που παρήχθη προηγουμένως, εφαρμόζεται η μέθοδος predict πάνω στο time series dataset που περιλαμβάνει τα νέα σήματα ελέγχου. Αυτή έχει ως αποτέλεσμα την ανάθεση μιάς ετικέτας (label) σε κάθε εφαρμογή, η οποία δηλώνει με ποιο cluster έχει αντιστοιχιστεί. Τα αποτελέσματα της ομοδοποίησης για τις άγνωστες εφαρμογές φαίνονται στον πίνακα 6.

Benchmark	**Cluster 0**	**Cluster 1**	**Cluster 2**	**Cluster 3**	**Cluster 4**	**Cluster 5**
ALS		✓				
LR			✓			
RF		✓				
PCA				✓		
SVD		✓				
WC		✓				

Table 6: Ανάθεση cluster στα benchmarks ελέγχου

Η παραπάνω ανάθεση δείχνει ποιο από τα προηγουμένως κατασκευασμένα μοντέλα πρέπει να χρησιμοποιηθεί για την πρόβλεψη του χρόνου εκτέλεσης κάθε εφαρμογής. Προκειμένου να μετρηθεί η ακρίβεια των μοντέλων στις άγνωστες εφαρμογές ελέγχου, εκτελούμε μια σειρά πειραμάτων για να μετρήσουμε τους πραγματικούς χρόνους εκτέλεσης για 100 παραμετροποιήσεις και για τις τρεις κατηγορίες μεγέθους δεδομένων εισόδου ώστε να μπορέσουμε να συγκρίνουμε με τις προβλεφθείσες τιμές. Ο πίνακας 7 δείχνει το cluster στο οποίο έχει ανατεθεί κάθε benchmark και το R^2 σκορ που επιτυγχάνει ανά κατηγορία μεγέθους.

Benchmark	**No Cluster**	**Score-Tiny**	**Score-Small**	**Score-Large**
ALS	1	< 0	< 0	< 0
Logistic	2	0.77	0.81	0.53
RF	1	0.92	0.77	0.66
PCA	3	0.64	0.91	< 0
SVD	1	0.80	< 0	< 0
WC	1	0.68	0.70	0.92

Table 7: Σκορ ανά benchmark ελέγχου

Το σχήμα 7 παρουσιάζει το speedup που επιτυγχάνεται, δηλαδή το κλάσμα της εκτέλεσης με την προεπιλεγμένη παραμετροποίηση που αποτελεί η δική μας βέλτιστη εκτέλεση κατόπιν μετρήσεως, μόνο για τα ζεύγη εφαρμογών-δεδομένων που πετύχαμε μη αρνητικό σκορ.

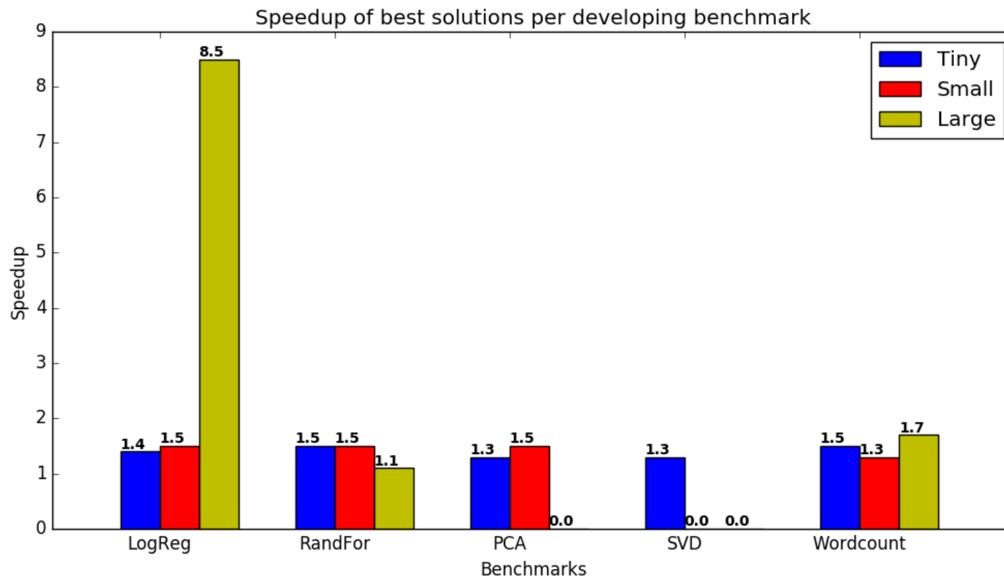


Figure 7: Σύγκριση speedups για όλα τα ζεύγη benchmark-δεδομένα με μη αρνητικό σκορ

Παρατηρούμε ότι οι χρόνοι εκτέλεσης για όλα τα ζεύγη που πέτυχαν θετικό R^2 σκορ, βελτιώνονται όταν τρέχουν με την παραμετροποίηση που επιλέγουμε αντί για την προεπιλεγμένη. Επίσης, η βελτίωση που επιτυγχάνεται σε αυτές τις περιπτώσεις είναι σημαντική. Το μέγιστο speedup πετυχαίνει $8.5\times$ όταν το LR τρέχει με 8 GB μέγεθος δεδομένων. Αυτό είναι λογικό μιας και η εφαρμογή LR είναι και compute και data intensive και συνεπώς υπάρχει πλήθος παραμέτρων που μπορούν να επηρεάσουν σημαντικα την εκτέλεσή της. Το μέσο speedup για όλα τα ζεύγη εφαρμογών-δεδομένων είναι $2.01\times$. Τέλος, είναι εμφανές ότι η βελτίωση που πετυχαίνουμε κατά τη γενική περίπτωση αυξάνεται όσο αυξάνεται και το μέγεθος των δεδομένων άλλη μια φορά.

Chapter 1

Introduction

1.1 Rise of distributed, in-memory computing

Cloud computing is a relatively new technology that has been of utmost importance for the past few years. It is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user. In other words, the cloud is comprised of software and services residing and operating on the Internet instead of a local computer or on-premise network of servers.

Cloud computing is one of the most important technological innovations of the 21st century. This is because it has seen the fastest adoption into the mainstream than any other technology in the domain. This adoption has been fueled mainly by the ever-increasing number of smartphones and mobile devices that can access the internet. Cloud computing is not just for organizations and businesses; it is also useful for the average person as well. It enables people to run software programs without installing them on their computers; it enables them to store and access their multimedia content via the internet, it enables them to develop and test programs without necessarily having servers and so on.

Moreover, it is helping the society cope with future problems such as managing big data, cybersecurity and quality control. Every year more and more enterprises engage in the practice of cloud computing or using remote servers hosted on the Internet to store, manage, and process critical data. Before the cloud era, teams often found it difficult to share insights widely. Coordination was cumbersome, sharing was difficult and transferring data, especially large amounts of it, was slow. The cloud has reduced many of these limitations, making it easy for teams to coordinate across any distance and to widely share data, ideas and information.

The benefits of the cloud are hard to overestimate. The most significant ones are:

- **Accessibility**: It facilitates the access of applications and data from any location worldwide and from any device with an internet connection.
- **Cost savings**: It offers businesses with scalable computing resources hence saving them on the cost of acquiring and maintaining them.
- **Security**: Cloud providers have strived to implement the best security standards and procedures in order to protect client's data saved in the cloud.
- **Disaster recovery**: It offers the most efficient means for enterprises to backup and restore their data and applications in a fast and reliable way.

- **Scalability**: It is the best option for businesses with fluctuating workloads since cloud infrastructure scales depending on the demands of the business.

Cloud's greatest advantage, though, is that it has emerged as an absolutely vital tool for big data. This is a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software. Data sets grow rapidly, to a certain extent because they are increasingly gathered by cheap and numerous information-sensing Internet of things devices. The world's technological per-capita capacity to store information has roughly doubled every 40 months since the 1980s; as of 2012, every day 2.5 exabytes (2.5×260 bytes) of data are generated. Based on an IDC report prediction, the global data volume was predicted to grow exponentially from 4.4 zettabytes to 44 zettabytes between 2013 and 2020. By 2025, IDC predicts there will be an astonishing number of 163 zettabytes of data.

Data with many cases (rows) offer greater statistical power, while data with higher complexity (more attributes or columns) may lead to a higher false discovery rate. Big data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source. Big data was originally associated with three key concepts: volume, variety, and velocity. When we handle big data, we may not sample but simply observe and track what happens, so big data often includes data whose sizes exceed the capacity of traditional software to process within an acceptable time and value. The current usage of the term big data, though, tends to refer to the use of predictive analytics, user behavior analytics, or certain other advanced data analytics methods that extract value from data, rather than to a particular size of data set.

Big data processing used to be cumbersome, and expensive. This also meant that big data efforts were reactionary, providing insights from out-of-date data. Businesses, however, need to be proactive and able to access, analyze and act upon the most recent data. Cloud-enabled big data, less so. No more data warehouses, no more hiring dedicated programmers just to run basic analyses. No more sweating collection, compiling, or analysis. And with the best big data tools, even presentation features are built right in. For example, when gathering customer analytics, with the cloud and big data, companies can quickly gather data from multiple sales, marketing, and web analytics, clickstream data, call center, and inventory sources. Then, without needing to use their own massive servers but instead the cloud, companies can compile the data, analyze it, quickly refine it into a presentation, and then act on it.

However, in the field of big data analytics collecting the necessary system resources, which can be done easily via the cloud, is not the only important thing for achieving good performance. In order to speed up the computation process when such big data are involved, it is important to do some scaling out. This means that many computers (or autonomous processes that run on the same physical computer) work together in order to solve a computational problem. This problem is divided into many tasks, each of which is solved by one or more computers (autonomous processes), which communicate with each other via message passing. In this way the time needed to solve the problem is reduced to the time needed to solve the most time-consuming task, plus the time needed for communicating and aggregating the results. This distributed way of running a computation has been proven to be more effective and less costly than scaling up, which means adding more resources to a unique computer in order to execute a process faster.

Another important aspect of handling big data in an efficient way is the in-memory computing. This means using a type of middleware software that allows one to store data in RAM, across a cluster of computers, and process it in parallel. As it is already known, RAM is roughly 5,000 times faster than traditional spinning disk. So this, along with the parallel processing, makes the

computations significantly faster. Frameworks that provide in-memory computing are nowadays preferred for many different tasks and have become highly popular.

Each year, there seem to be more and more distributed frameworks on the market to manage data volume, variety, and velocity, but not all of them support in-memory computing. The most important ones are:

- **Apache Hadoop**: It's a general-purpose form of distributed processing that has several components such as the Hadoop Distributed File System (HDFS), which stores files in a Hadoop-native format and parallelizes them across a cluster, YARN, a schedule that coordinates application runtimes and MapReduce, the algorithm that actually processes the data in parallel. Hadoop is built in Java, and is accessible through many programming languages for writing MapReduce code.

- **Apache Spark**: It is also a top-level Apache project focused on processing data in parallel across a cluster, but the biggest difference is that it works in-memory. Whereas Hadoop reads and writes files to HDFS, Spark processes data in RAM using a concept known as an RDD, Resilient Distributed Dataset.

- **Apache Flink**: It is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments and perform computations at in-memory speed and at any scale.

- **Disco MapReduce**: It is a lightweight, open-source framework for distributed computing based on the MapReduce paradigm. It is powerful and usage of Python makes it robust and easy-to-use. Disco distributes and replicates the data, and schedules jobs efficiently. The framework helps build and query in real-time indices with billions of keys and values, using DiscoDB and it also provides its own file system, Disco Distributed File System (DDFS).

The most popular framework is undoubtedly the Apache Spark. The in-memory computing it provides and its DAGs enable optimizations between steps make it a lot faster for iterative and interaction applications than the frameworks that use on-disk computing. Finally, spark has very strong community support and a good number of contributors, so this is the framework that we have chosen to work with.

1.2 Motivation

Apache Spark, as well as the other frameworks mentioned in the previous section, offers a great number of parameters for tuning. Modifying their value from the default can affect more or less the performance of the execution, depending on the significance of this particular parameter.

Zhibin Yu et al. [1] state that there are 41 parameters that can be easily tuned and significantly affect performance. Zhendong Bei et al. [2] state that there are 29 parameters that are important and determine the application's execution time. The two papers may disagree on the number of the significant parameters but they both agree on the fact that parameters can actually have a powerful effect on an application's performance. Their disagreement leads us to study further the parameters of the spark framework and search once again for the most important ones.

In order to be absolutely sure that a significant improvement in an app's performance can be achieved by reconfiguring spark's parameters, we ran a little experiment. Specifically, we ran the Bayes, Kmeans and the SVM benchmarks from the HiBench Suite for a large data size and for 100

different configurations, the default and other 99 randomly chosen ones. The goal is to observe the performance variation, which we define as follows.

$$per_{var} = \frac{t_{max} - t_{min}}{t_{min}}$$

The results are shown in the figure that follows.

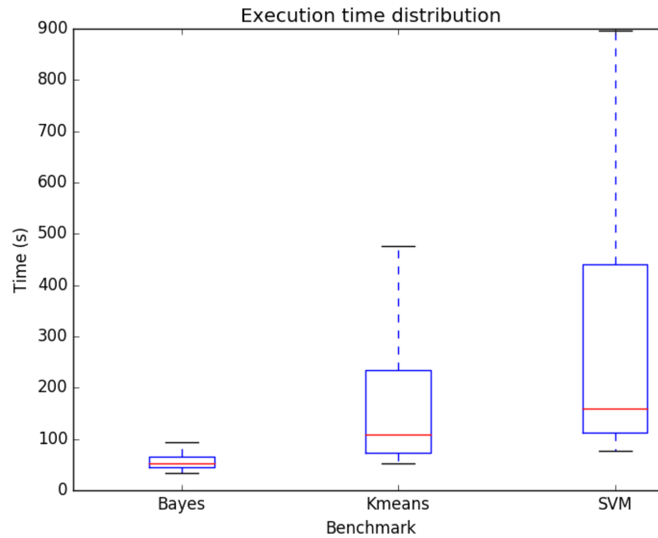


Figure 1.1: Execution time distribution for different spark configurations per benchmark

As shown in the figure, a different tuning of an application's parameters can affect significantly its performance for the exact same input dataset. Depending on the workload, this impact can be smaller, as in the case of Bayes, or can be as large as 10.25, as in the case of SVM. In any case, the variation is not negligible and, therefore, it is worth finding the most important parameters and create a program to autotune an application to run with the optimal configuration.

1.3 Objectives and Contributions

As it has already been stated, the default configuration that is proposed from the apache spark framework cannot always accomplish a small enough execution time and, as a result, it becomes necessary for the user to change some of the parameters based on the program's and the machine's characteristics. Finding the best configuration, though, is not a task that can be easily done manually because of the number of different parameters and the range of the possible values for them. Some of the parameters do not have a great impact on performance, so it is very important to distinct them in order to reduce the dimensions of the problem. This can be done with some experiments and with the use of statistic Kruskal Wallis test.

Even so, the search space remains very large and, evidently, the optimization process needs to be automated. OpenTuner's optimizers are apt for that job. They produce candidate configuration vectors and move through the search space based on the time results each of them brings, with the objective to minimize execution time. However, in order to measure the execution time and evaluate the solutions, it would be necessary to run repeatedly an application on spark with the

different configurations and collect the results. This might take an enormous amount of time since, in certain cases, one only execution could take up to 1-3 hours. Therefore, experiments only ran to collect enough data to train a model, based on the random forest algorithm, which could be later used for predicting execution times and inform the optimizer on the spot. There is certainly a loss in accuracy but the optimization process becomes a matter of seconds.

The most important aspiration is to make the predicting model achieve high scores of accuracy even on unseen applications that have not been used at the training process. In order to do that, all the different applications that were executed to collect training data were separated to groups based on their similarity in some characteristics relative to their behavior on an architecture level. For each group there was constructed a different prediction model that was used only on applications that are assigned to this group. This made it possible to predict with some accuracy unseen applications by checking in which group they belong to and using the model of that group.

Consequently, an automated tuning of spark parameters has been made possible, through the use of regression models that predict the execution time of an application, which offers a significant improvement in performance over the default configuration.

Chapter 2

Related Work

As mentioned before, there has already been a few papers trying to optimize the execution of programs running on spark. This is not surprising at all, since the default configuration is most of the times not satisfyingly good.

First, the Spark official web site provides a performance tuning guide for Spark workloads [3], but this is a manual approach and the user cannot easily take advantage of it without having a deep knowledge of Spark. In contrast, our approach is automated and the user runs the workload he wants with an optimal configuration without doing anything.

Also, Zhibin Yu et al. [1] distinguished 41 parameters and proposed a methodology to find the optimal values for them with the use of genetic algorithms, by collecting execution time information from a performance model. Specifically, they used an hierarchical modeling technique, employing regression trees as sub models, instead of creating a single sophisticated model. This was proven to create more accurate models that resist the overfitting problem and produce more reliable results with lower error rate. However, this work builds a single performance model for all the experimented benchmarks and for that reason it cannot generalize the results to unknown applications. Instead, the model can only predict the behavior of the benchmarks that have been used in the training process. Therefore their methodology can only be applied to a limited number of workloads, while ours can be used to optimize even unknown applications, though sometimes with lower accuracy.

Moreover, Zhendong Bei et al. [2] used genetic algorithms and a performance model based on a three-step random forest to find the optimal configuration. They produced one model for each benchmarks because their behaviors are significantly different and a single model could not be satisfyingly accurate. They also showed that their results remained accurate enough even with different workloads used in the training process. Guolu Wang et al. [10] selected 14 parameters and proposed to tune Spark configurations by using decision tree modeling and search the parameter space with the use of Recursive Random Search algorithm. However, these works again cannot generalize the results to unknown applications. Instead, the model can only predict the behavior of the benchmarks that have been used in the training process. Therefore their methodology can only be applied to the limited number of the experimented workloads, while ours takes into account the deep architectural characteristics of a workload and can be used to optimize even unknown applications.

Furthermore, Liang Bao et al. [4] proposed a tuning method that runs under a time constraint, using a smaller-scale testbed. In order to build a predictive model, they only ran the complete job on small samples of the input data provided by the user and collected the corresponding times, making assumptions about what the execution time of the whole input data would be. The model was built using random forest algorithm and the search of the exploration space was then made

with the Latin hypercube sampling, instead of genetic algorithms that we used. This new insight they provide is interesting but, even though more time consuming on its production process, our work produced a script that can be used on the spot, with minimum effort and time cost for the user. Also, their results have only been tested on one datasize so they may not scale out appropriately.

It is also important to point out that the above papers select the important parameters based on intuition, common logic and the spark team's recommendations. In contrast, we based the parameter selection on a whole process of experiments and statistical analysis of the produced results with the Kruskal Wallis test. Therefore, our study of the parameters' significance is better established.

In addition, Tatsuhiro Chiba et al. [5] carefully characterize the memory, network, JVM, and garbage collection behavior of TPC-H queries on Spark. Then they manage to optimize the performance of Spark workloads by reducing the garbage collection overhead and by increasing the IPC. However, this work only focuses on TPC-H workloads while ours focuses on general-purpose Spark applications.

Finally, Ayat Fekry et al. [6] propose a methodology to find the cluster setups and the resources to be allocated, concerning number of CPUs, network bandwidth, memory/disk size etc, so that a certain workload will run optimally. In this direction, also, Kewen Wang et al. [7] propose a model driven methodology to increase spark performance by identifying possible data locality and task distribution problems and recommend ways to address these problems. Even though finding the appropriate cluster setup for a workload and addressing some data locality problems is of great importance, it should be combined with a tuning of some more of the internal spark parameters, such as the one we have proposed, in order to actually achieve an optimized execution.

Kay Ousterhout et al. [8] propose a block analysis approach to systematically analyze the bottleneck of Spark programs using two SQL workloads. While this work provides a good performance analysis tool and reveal some bottlenecks of the SQL-like programs, it does not optimize general Spark programs such as those implementing machine learning algorithms. In contrast, our framework provides an approach to automatically optimize the performance of general Spark programs. Yao Zhao et al. [9] tried to improve the performance of Spark programs by applying adaptive tuning of the serialization techniques, while we take into account a large set of Spark configuration parameters, instead of only the serialization aspect.

Moreover, Juwei Shi et al. [24] compared the MapReduce with Spark and provided interesting insights but they do not propose an approach to improve the performance of Spark workloads. Kenli Li et al. [25] propose a stochastic dynamic level scheduling algorithm which could be applied in optimizing Spark workloads.

Another class of related work is to optimize the configurations of MapReduce/Hadoop workloads. Herodotou et al. propose to build analytical performance models first and then leverage genetic algorithm to search the optimum configurations for Hadoop workloads [26, 27, 28]. Adem et al. [29] suggest using a statistic reasoning technique named response surface (RS) to construct performance models for MapReduce/Hadoop programs and then implement the models in a MapReduce simulator. These studies work well for Hadoop workloads but Spark workloads are significantly different from Hadoop workloads from design concepts to implementation, implying that the configuration techniques for the Hadoop workloads can not be easily extended for the Spark workloads.

In summary, there are pioneer studies on the field of finding the optimal spark configuration and minimizing an application's execution time. Table 2.1 shows the basic methodology that each of them follows. However, to our best knowledge, there has been no study yet that has achieved, on some extent, to optimize even unseen applications with the use of a model built on information extracted by a group of similar workloads.

Main Writer	Description
Spark website	Manual tuning guide
Zhibin Yu	Spark parameter selection and use of hierarchical modeling and genetic algorithms
Zhendong Bei	Spark parameter selection and use of three-step random forest based model per benchmark and genetic algorithms
Guolu Wang	Spark parameter selection and use of decision tree modeling and Recursive Random Search algorithm
Liang Bao	Random forest modeling on small samples of data and Latin Hypercube sampling for optimal solution searching
Tatsuhiro Chiba	Characterization of memory, network, JVM, and garbage collection behavior of TPC-H queries on Spark and improvement of those
Ayat Fekry	Search of optimal cluster setup and resource allocation for Spark
Kewen Wang	Model for predicting data locality and task distribution problems in Spark and addressing them
Kay Ousterhout	Block analysis to systematically analyze the bottleneck of Spark programs using two SQL workloads
Yao Zhao	Adaptive tuning of the serialization techniques in Spark
Juwei Shi	Comparing MapReduce with Spark
Kenli Li	Scheduling algorithm for Spark workload optimization
Herodotos Herodotou	Performance model selection and genetic algorithms for Hadoop workloads optimization
Adem	Use of Response Surface based model for Hadoop workloads and implementation of them in a MapReduce simulator.
Dimitra Nikitopoulou	Well established parameter selection, spark workloads clustering based on architectural metrics and use of random forest modeling per cluster and genetic algorithms

Table 2.1: Studies about spark and hadoop applications' performance improvement

Chapter 3

The Apache Spark Framework

Spark [15, 16] is a general-purpose distributed data processing engine that is suitable for use in a wide range of circumstances. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation, and stream processing, which can be used together in an application. It supports a range of programming languages, including Java, Python, R, and Scala. Spark's capabilities are accessible via a set of rich APIs, all designed specifically for interacting quickly and easily with data at scale. These APIs are well-documented and structured in a way that makes it straightforward for data scientists and application developers to quickly put Spark to work. Last but not least, Spark is designed for speed, operating both in memory and on disk, making it perfect for the processing of big data. Its jobs perform multiple operations consecutively, in memory, and only spilling to disk when required by memory limitations. Spark offers an integrated whole, a data pipeline that is easier to configure, easier to run, and easier to maintain.

Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. What is stored is not the actual data but the way towards their computation from other stored data, when this becomes necessary (lazy evaluation). Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. Without this innovation, data would be read from the disk every time and the time cost would be significantly larger. Caching RDDs is very useful in machine learning operations such as applying a function repeatedly to the same dataset to optimize a parameter or even when querying repeatedly the same large dataset.

3.1 Apache Spark Overview and Architecture

The Apache Spark framework uses a master–slave architecture that consists of a driver, which runs as a master node, and many executors that run across as worker nodes in the cluster. Driver Program in the Apache Spark architecture calls the main program of an application and creates SparkContext. A SparkContext consists of all the basic functionalities.

Spark Driver and SparkContext collectively watch over the job execution within the cluster. Spark Driver works with the Cluster Manager to manage various other jobs. Cluster Manager does the resource allocating work. Spark Context takes the job, breaks the job in tasks and distribute them to the worker nodes. Worker nodes are the slave nodes whose job is to basically execute the tasks. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main Spark Context.

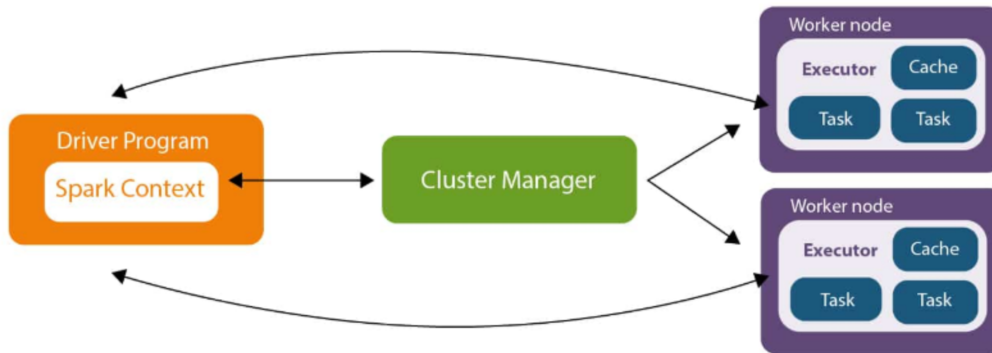


Figure 3.1: Apache Spark architecture

SparkContext supports many cluster managers such as Standalone, Yarn [22], Mesos [30], Kubernetes [31], which allocate resources so that the workers can execute their tasks. The one that has been selected is Hadoop YARN (Yet Another Resource Negotiator).

YARN has two basic components:

- **Resource Manager**: It manages the resources for all system applications and it consists of the Scheduler and an Application Manager. The Scheduler is responsible for allocating resources to the various running applications without monitoring or tracking of status for the application and offering guarantees about restarting failed tasks either due to application failure or hardware failures. The Application Manager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure.
- **Node Manager**: It is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager

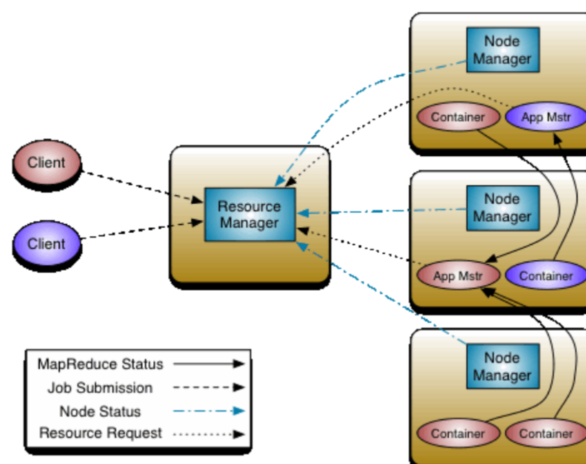


Figure 3.2: YARN architecture

The Resource Manager is only one along the computer cluster whereas the Node Managers are as many as the machines. Being unique, the resource manager is responsible for all the different applications of the system.

The per-application manager is the Application Master, which negotiates resources with the Resource Manager and works with the Node Managers for the execution and monitoring of tasks. If the system runs on cluster mode, the ApplicationMaster is also responsible for executing the driver, so the client initiating the application is free to leave. However, if it runs on client mode, which is used in this diploma, the driver runs inside the client ans so it must remain present.

3.2 Apache Spark Properties

Because of the in-memory nature of most Spark computations, Spark programs can be bottlenecked by any resource in the cluster: CPU, network bandwidth, or memory. Most often, if the data fits in memory, the bottleneck is network bandwidth, but sometimes, there is also need for some tuning, such as storing RDDs in serialized form, to decrease memory usage.

◇ **Spark Memory Distribution**

Memory in spark is divided in 3 different regions:

- **Reserved Memory**: This is the memory reserved by the system and its size is hardcoded at 300 MB.
- **User Memory**: This is the memory pool that remains after the allocation of Spark Memory, and it can be used to store data structures that would be used in RDD transformations.
- **Spark Memory**: This is the memory pool managed by Apache Spark. Its size can be calculated as $(Java\ Heap–Reserved\ Memory) * spark.memory.fraction$. It is further divided into 2 categories:
  * **Storage Memory**: This pool is used for both storing Apache Spark cached data and for temporary space serialized data unroll. Also all the broadcast variables are stored there as cached blocks.
  * **Execution Memory**: This pool is used for storing the objects required during the execution of Spark tasks. It also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other threads (tasks).

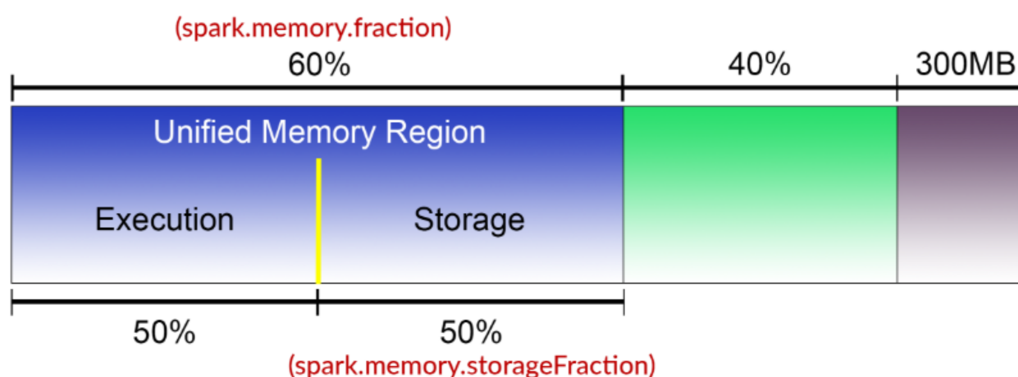


Figure 3.3: Spark Memory Distribution

◇ **Data Serialization**

Serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation. Spark provides two serialization libraries, Java serialization, which is flexible but often quite slow and leads to large serialized formats for many classes, and Kryo serialization which is significantly faster and more compact than Java serialization, but does not support all Serializable types.

◇ **Data shuffling**

Shuffling is a process of redistributing data across partitions, that may or may not cause moving data across JVM processes or even over the wire (between executors on separate machines). Shuffling is the process of data transfer between stages. By default, shuffling does not change the number of partitions, but their content.

◇ **Level of Parallelism**

Clusters will not be fully utilized unless the level of parallelism for each operation is set high enough. The default is set to the largest parent RDD's number of partitions.

◇ **Garbage collection**

JVM garbage collection can be a problem when there are large data in terms of the RDDs stored by a program. When Java needs to evict old objects to make room for new ones, it will need to trace through all Java objects and find the unused ones. Spark offers the opportunity to the programmer to choose how often garbage collection is triggered, to disable it and in general to do some tuning on it.

3.3 Apache Spark Parameters

Spark has over a hundred parameters configured at default values, which are presented in the spark's configuration webpage [32]. Obviously, not all of them have the same impact on the performance of the applications. Some refer to names, paths in the file system and other similar information that could not in any case affect performance, so they were not taken into account. The ones that were deemed as possibly significant are 99 and they concern a wide range of configurations that can be applied on the spark system. Reducing this number of parameters even more could actually hide some that are significant, so we chose to test all of them.

Some of them control primary characteristics such as memory size and number of cores to be used by the executors and the driver, or offheap memory and memory fraction to be used for storage and execution. Others decide important information about the reduce task, the shuffling and the compression processes, the data serialization and the broadcast behavior. There are also plenty parameters that define the Spark UI and the garbage collection behavior and others that control the manipulation of files and remote procedure calls (RPC). Moreover, some of them control the behavior of the scheduling process, the waiting time in order to achieve locality and the speculative execution of tasks. Finally, there are parameters that configure the task execution and others that activate executor blacklisting and control some important aspects of it.

Chapter 4

An automated framework for tuning spark applications

As discussed in the previous chapters, determining the most efficient parameters of Spark workloads is a challenging task. Former scientific efforts have proposed frameworks that are able to automate the regulation of spark parameters. However, none of the existing papers have presented a well established study about the significance of the different spark parameters that they chose to reconfigure. Moreover, an effort to optimize unseen spark workloads has not yet been made, in spite of the great importance it would have.

Our framework is based upon finding the most significant parameters in a purely scientific and not intuitive way and it aims at finding an accurate enough way to optimize any spark workload. Specifically, our framework consists of four steps. In the first step, described in section 4.2, we perform some experiments and study the significance of each of the 99 parameters so as to decide which of them are worth reconfiguring. In the second step, described in section 4.4.2, we execute a series of experiments to create our training dataset and then we perform a clustering of the different applications that we have selected to use for developing and testing the results, in order to split them into groups of similar ones. In the third step, described in section 4.4.3, we test several regression algorithms to make a representative model of each group/cluster that predicts accurately enough the execution time of an application-input pair for a certain configuration. In the fourth and final step, described in section 4.5, the models are fed to an optimization algorithm, which uses them for evaluating the different configuration vectors easily and quickly, and producing the optimal configuration.

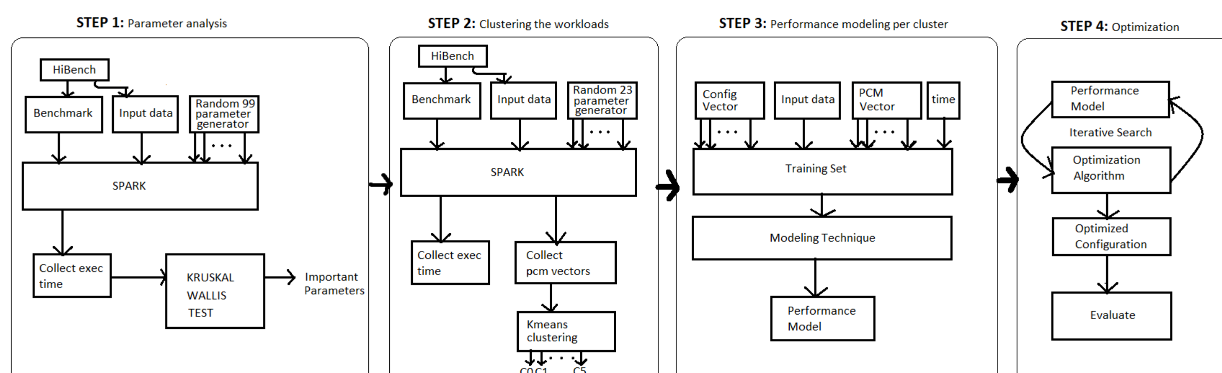


Figure 4.1: Overview of the proposed framework for spark parameters auto-tuning

4.1 Experimental Setup and Specifications

All the experiments were executed on a single server, in a pseudo distributed way, with the use of the Hadoop Distributed File System (HDFS) [23] instead of the local file system. Our server is equipped with 48 Intel(R) Xeon(R) CPU E5-2658A v3 2.20GHz 12-core processors, 125 GB memory and 1 TB hard disk. The L1 instruction cache as well as the data cache are 32 KB, the L2 cache is 256 KB and the L3 cache is 30.72 MB. The operating system is Linux, release 4.15.0-101-generic. The experimental Apache Spark version is 1.6.0 and the Hadoop version is 2.6.0. The data block size of HDFS is left to the default value 128 MB.

The benchmarks selected for developing our automated framework, as well as those used for validating its ability to produce good results even on unknown workloads come from the HiBench Benchmark Suite [11]. The in-memory version of HiBench has different kinds of workloads, including machine learning, sql functions and web search. These benchmarks represent a sufficiently broad set of typical Spark workload behaviors. We select 12 benchmarks for developing, which are shown in table 4.1, so that we have a powerful knowledge base and 6 benchmarks for validating, shown in table 4.2.

		Input Data Size for each category		
Benchmark	**Abbr.**	**Tiny**	**Small**	**Large**
Bayesian Classification	Bayes	93.1 MB	111.7 MB	377.1 MB
K-means clustering	Kmeans	1.3 MB	602.4 MB	4 GB
Support Vector Machine	SVM	8 MB	800.6 MB	20 GB
Aggregate	Aggr	54.3 KB	3.7 MB	37.2 MB
Join	Join	199.2 KB	19.2 MB	191.9 MB
Scan	Scan	206.2 KB	20.1 MB	201 MB
PageRank	PR	10.8 KB	1.8 MB	259.9 MB
Linear Regression	Linear	4 GB	16 GB	48 GB
Gradient Boosting Trees	GBT	11.3 KB	408.4 KB	16MB
Sort	Sort	36 KB	3.2 MB	328.4 MB
Latent Dirichlet Allocation	LDA	21.7 MB	97.5 MB	258.7 MB
TeraSort	TS	3.2 MB	320 MB	3.2 GB

Table 4.1: Experimented benchmarks for developing

		Input Data Size for each category		
Benchmark	**Abbr.**	**Tiny**	**Small**	**Large**
Alternating Least Squares	ALS	67.4 KB	6 MB	120.6 MB
Logistic Regression	LR	808.4 KB	80 MB	8 GB
Random Forest	RF	11.3 KB	408.4 KB	8 MB
Principal Components Analysis	PCA	88.4 KB	8 MB	32.1 MB
Singular Value Decomposition	SVD	805 KB	16 MB	64 MB
WordCount	WC	36.2 KB	328.4 MB	3.2 GB

Table 4.2: Experimented benchmarks for validating

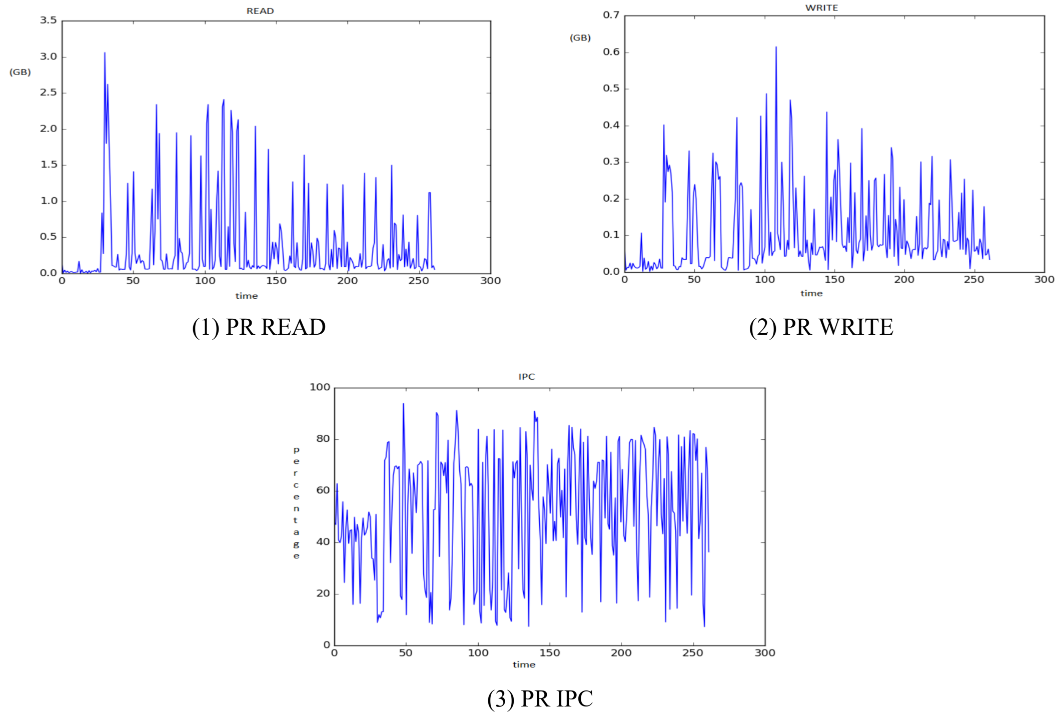
Evidently, the information that is helpful at characterizing an application concern CPU, network and memory behavior. Specifically, what needs to be studied is IPC, frequency, cache behavior

and memory IO. Performance Counter Monitor (PCM) [33] is an intel program that can run in the background, while an application is running, and measure all the above system information. What it actually does is collect in a .csv file with numerous columns wide information that concerns the application's behavior on an architecture level. This information was received every 0.1s, as it was defined, throughout the application's execution. The columns of the PCM file (metrics) that were evaluated as most significant are the ones shown in table 4.3.

PCM Metric	Description
PhysIPC%	Instructions per cycle (IPC), multiplied by the number of threads per core (2 threads/core) and divided by the maximum IPC (4) and again multiplied by 100%
TotalQPIout	Traffic estimation in the QuickPath Interconnect (QPI), which is the size of data and non-data in MB that go out of the CPUs and sockets through the links of the interconnection network QPI
READ	The memory reads in GB
WRITE	The memory writes in GB
AFREQ	Frequency divided by nominal CPU frequency (2.10 GHz), excluding the time when CPU is sleeping
L3MISS	Number of cache line misses in the level 3 cache, measured in millions
L2MISS	Number of cache line misses in the level 2 cache, measured in millions
L3HIT	The cache hit ratio in cache level 3
L2HIT	The cache hit ratio in cache level 2

Table 4.3: Important PCM Metrics

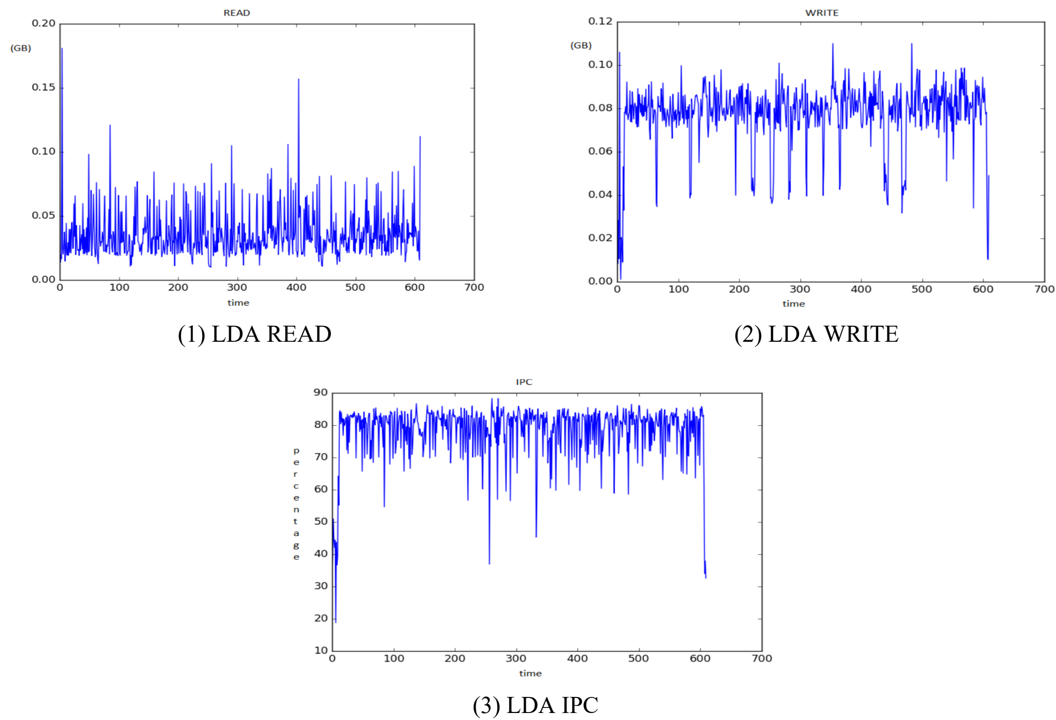
Firstly, we distinguish the three metrics that concern number of reads and writes from/to the memory and the IPC so as to make a categorization concerning compute and data intensive benchmarks. Figures 4.2 and 4.3 show the respective READ, WRITE and IPC trace signal plots for two of our 12 developing workloads as derived from the PCM tool. In the first figure, that corresponds to the PR benchmark, we have large values of memory reads and writes and an IPC that goes up and down, probably due to the stalls that the memory accesses cause. On the other hand, the second figure that corresponds to the LDA benchmark maintains a high IPC throughout its execution and has few reads and writes in memory.



(1) PR READ (2) PR WRITE

(3) PR IPC

Figure 4.2: PR as example of data intensive application



(1) LDA READ (2) LDA WRITE

(3) LDA IPC

Figure 4.3: LDA as example of compute intensive application

By constructing plots as the ones above, we managed to characterize all of the developing and validating benchmarks as compute or/and data intensive. The rest of the plots are presented in

the Appendix A. The ones considered as data intensive have high memory accesses throughout their execution and an IPC that has many spikes while the ones considered as compute intensive maintain a high IPC percentage throughout their execution and rare and low memory accesses. If high IPC and often memory accesses coexist then the benchmark is characterized as both compute and data intensive. The results of our characterization is shown in table 4.4.

Benchmark	Compute-Intensive	Data-Intensive
Bayes	✓	✓
Kmeans		✓
SVM	✓	✓
Aggr	✓	
Join	✓	
Scan	✓	
PR		✓
Linear	✓	✓
GBT	✓	
Sort		✓
LDA	✓	
TS	✓	✓
ALS	✓	
LR	✓	✓
RF	✓	
PCA		✓
SVD	✓	✓
WC		✓

Table 4.4: Characterization of the benchmarks

4.2 STEP 1: Exploration and Pruning of Spark Parameters Design Space

In this step, our framework attempts to prune the spark parameters design space by determining which of them are the most important ones, meaning those that cause a significant difference in performance (for better or worse) if their value is changed from the default. Reconfiguring them, in away that takes into account the primary characteristics of the application as well as those of the machine on which the application runs, can improve the performance significantly and reduce the execution time.

As stated before, what was deemed as possibly significant, were 99 parameters that needed to be further analyzed in order to make assumptions for their impact on the execution time of the applications. This analysis was done by running different applications a number of times, using each time a randomly selected configuration vector. The results of those experiments were then used for statistical analysis.

Our proposed framework relies on OpenTuner [12, 13] not only for the optimization process, but also for the creation of these randomly created configurations. OpenTuner is a framework that can be used for autotuning a program, which means searching among the possible values of its parameters and come up with a configuration that optimizes its execution based on some objective. These possible values form the search space and they are defined by a configuration manipulator.

This manipulator includes a set of parameter objects which OpenTuner will search over. A run function is also needed to be defined, so as to evaluate the fitness of a given configuration in the search space and produce a result. This result is a database record type containing many other optional fields such as time, accuracy, and energy, which are possible optimization objectives. In this step, however, we do not define an objective, but instead we exit the process and call it from the start multiple times because we want randomly created vectors that are not evaluated based on any objective.

In order to make this configuration manipulator, three representative values were selected for every parameter (except for booleans) from within their accepted range. Specifically, for arithmetic values the default value was selected, one significantly higher and one significantly lower and on the other cases the alternative values were chosen.

OpenTuner was then used to make 300 configuration vectors randomly selected from within the search space. A brief analysis of these vectors showed that they all differed from each other in at least 40 parameters. The configuration vectors looked like this:

$$conf_i = \{c_{i1}, c_{i2}, .., c_{ij}, .., c_{i99}\}, 1 \leqslant i \leqslant 300$$

, where $conf_i$ is the i-th configuration and c_{ij} is the value of j-th parameter in the i-th configuration.

After the different configurations have been created, the framework performs a series of experiments. Specifically, the first seven benchmarks from table 4.1 were executed for tiny, small and large input datasizes and for the 300 configurations. Seven applications were selected with different memory and CPU behavior so that the results could be compared to each other and generalized more accurately. In order to change the configuration in spark it is necessary to load a properties file that contains parameter-value pairs like this:

```
hibench.yarn.executor.num       6
spark.task.cpus     1
spark.shuffle.compress      true
spark.memory.fraction       0.5
spark.scheduler.maxRegisteredResourcesWaitingTime       10s
spark.sql.shuffle.partitions        75
spark.cleaner.periodicGC.interval       5min
spark.io.compression.lz4.blockSize      64k
```

Figure 4.4: Content of file spark.conf

The result of the previous experiments described above was a vector of the corresponding execution times $t = \{t_1, t_2, .., t_i, .., t_{300}\}$. In order to establish the importance of a parameter, the execution times were grouped by the value of this parameter on each configuration. Thus, there were created as many different vectors as the representative values chosen for the parameter.

For example, if for the first parameter c_{i1} in the first 5 configurations we have

- $c_{1\ 1} = true$ and $t_1 = 25.1$
- $c_{2\ 1} = true$ and $t_2 = 26.1$
- $c_{3\ 1} = false$ and $t_3 = 35.1$
- $c_{4\ 1} = true$ and $t_4 = 24.5$
- $c_{5\ 1} = false$ and $t_5 = 34.1$

then

- $t_{true} = \{25.1, 26.1, 24.5\}$
- $t_{false} = \{35.1, 34.1\}$

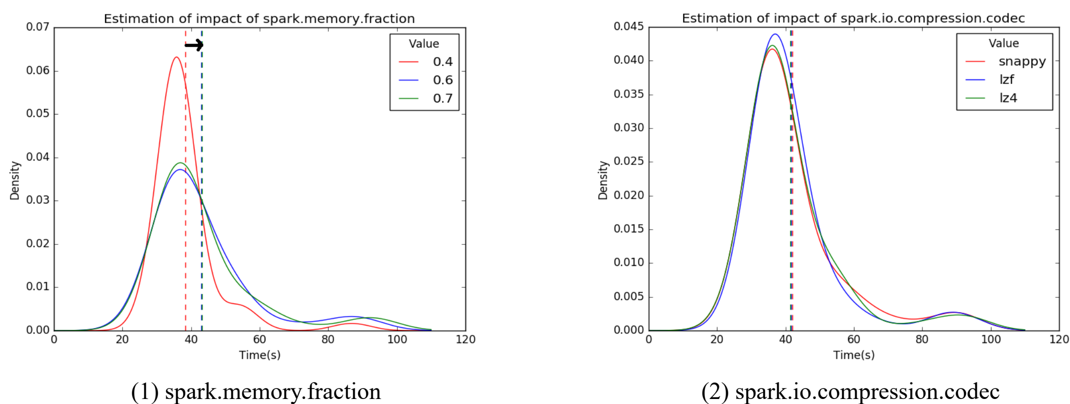
We emphasize that for parameters such as spark.kryoserializer.buffer.max that needs for spark.serializer to be set at the KryoSerializer value in order to take effect, only those configuration vectors that satisfied this condition were used at the grouping stage.

4.2.1 Kruskal Wallis Test

After these newly created vectors (groups) have been created, the framework performs a Kruskal-Wallis test [19] on them. This is a non-parametric method for testing whether samples originate from the same distribution. The function checks if the means of the vectors differ significantly. Since the values belong to distributions that cannot be considered as normal because they are based on experimental data, the one-way analysis of variance (ANOVA) test cannot be used instead.

The Kruskal-Wallis tests the null hypothesis H0, which states that samples in all groups are drawn from populations with the same mean values. The alternative hypothesis states that at least two of the groups have mean values that differ. The algorithm returns a p-value. In order to find out if the null hypothesis is confirmed or rejected, the programmer needs to set an acceptance rate. We set that limit at 5%. This guarantees that the possibility to show different means when the distributions are actually the same is below 5%. Thus, when the p-value returned is greater than 0.05 the null hypothesis is confirmed and there is no significant difference in the means, while if it is lower the null hypothesis is rejected. Evidently, a confirmed H0 indicates an insignificant parameter and a rejected H0 a significant one.

The below density plots depict two representative examples, showing the process followed to determine the importance of each spark parameter. Specifically, they depict the execution time distribution for the three different values chosen for the given parameter. The first diagram, obviously, depicts a significant parameter since for the value 0.4 we have a shift of the distribution to the left. The second one has no such significant shift so it corresponds to an insignificant parameter. Such figures for the rest of the parameters are shown in the Appendix B.



(1) spark.memory.fraction (2) spark.io.compression.codec

Figure 4.5: Density plots of different parameters

After comparing the results of the test for every benchmark-input pair, it was decided that the most significant parameters in general case were actually 23, which are shown in table 4.5. They

affect spark's environment either directly or through the cluster manager yarn. The cases with a rejected H0 were actually more, but the ones that had the default and not another value significantly better, according to the density plots, were not taken into account.

Parameter	**Description**
spark.executor.instances	The number of executors
spark.executor.cores	The number of cores used by each executor
spark.task.cpus	Number of cores allocated for each task
spark.shuffle.compress	Whether to compress map output files
spark.executor.memory	Amount of memory to use per executor process
spark.memory.fraction	Fraction of (heap space - 300MB) used for execution and storage. The lower this is, the more frequently spills and cached data eviction occur. The purpose of this configuration is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse, unusually large records
spark.memory.storageFraction	Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by spark.memory.fraction. The higher this is, the less working memory may be available to execution and tasks may spill to disk more often
spark.serializer	Class to use for serializing objects that will be sent over the network or need to be cached in serialized form
spark.scheduler.maxRegisteredResourcesWaitingTime	Maximum amount of time to wait for resources to register before scheduling begins
spark.default.parallelism	Default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by user
spark.sql.shuffle.partitions	The number of partitions used during data shuffling for joins and aggregations
spark.cleaner.periodicGC.interval	How often to trigger a garbage collection
spark.io.compression.lz4.blockSize	Block size in bytes used in LZ4 compression, in the case when LZ4 compression codec is used. Lowering this block size will also lower shuffle memory usage when LZ4 is used
spark.yarn.am.memory	Amount of memory to use for the YARN Application Master in client mode
spark.scheduler.revive.interval	The interval length for the scheduler to revive the worker resource offers to run tasks
spark.locality.wait.process	Customize the locality wait for process locality. This affects tasks that attempt to access cached data in a particular executor process
spark.shuffle.sort.bypassMergeThreshold	Avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions
spark.shuffle.io.preferDirectBufs	Whether to use off-heap buffers to reduce garbage collection during shuffle and cache block transfer
spark.task.maxFailures	Number of failures of any particular task before giving up on the job
spark.files.openCostInBytes	The estimated cost to open a file, measured by the number of bytes could be scanned at the same time. This is used when putting multiple files into a partition
spark.shuffle.file.buffer	Size of the in-memory buffer for each shuffle file output stream. These buffers reduce the number of disk seeks and system calls made in creating intermediate shuffle files
spark.cleaner.referenceTracking.blocking	Controls whether the cleaning thread should block on cleanup tasks
spark.kryoserializer.buffer.max	Maximum allowable size of Kryo serialization buffer

Table 4.5: Selected Spark Parameters

4.3 Per-application Impact of important spark parameters

Since the various workloads are different from each other, for example some are data and others compute intensive, the Kruskal Wallis test did not show for every benchmark exactly the same important parameters. Some of them, such as spark.executor.memory, spark.task.cpus, spark.executor.cores, were deemed in all cases as important, while the rest were not. The result depends on the characteristics of the application as well as the size of the input data. The following diagram shows an example of the impact of all the 23 parameters considered as important for a randomly selected benchmark-datasize pair, which is PR with a small input dataset. Such diagrams were built for all the benchmarks that participated in the parameter analysis process. The parameters that are significant and reject the null hypothesis of the test are those who go over the red line at 0.95, since we have selected a p-value equal to 0.05. It becomes clear that not all 23 of them have a great impact on the execution of this particular benchmark. Based on the above procedure, the importance on a per-application basis were determined, which are shown in Table 4.6.

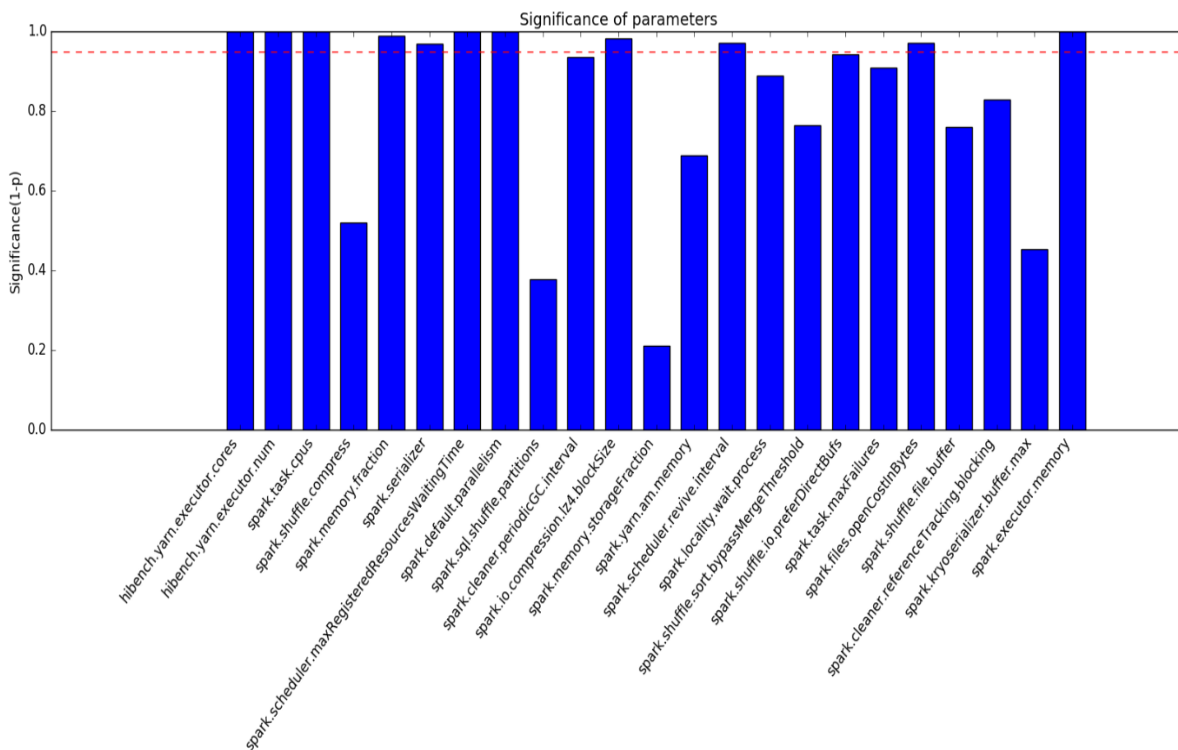


Figure 4.6: Impact of parameters on the execution of PR with small input size

Parameter	Bayes	Kmeans	SVM	Aggr	Join	Scan	PR
spark.executor.instances	✓	✓	✓	✓	✓	✓	✓
spark.executor.cores	✓	✓	✓	✓	✓	✓	✓
spark.task.cpus	✓	✓	✓	✓	✓	✓	✓
spark.shuffle.compress	✓						
spark.executor.memory	✓	✓	✓	✓	✓	✓	✓
spark.memory.fraction		✓			✓	✓	✓
spark.memory.storageFraction							✓
spark.serializer		✓	✓				✓
spark.scheduler.maxRegisteredResourcesWaitingTime	✓	✓			✓	✓	✓
spark.default.parallelism							✓
spark.sql.shuffle.partitions				✓			
spark.cleaner.periodicGC.interval	✓				✓	✓	
spark.io.compression.lz4.blockSize	✓	✓	✓		✓		✓
spark.yarn.am.memory	✓	✓				✓	
spark.scheduler.revive.interval		✓	✓		✓	✓	✓
spark.locality.wait.process			✓	✓	✓	✓	✓
spark.shuffle.sort.bypassMergeThreshold					✓	✓	✓
spark.shuffle.io.preferDirectBufs		✓					✓
spark.task.maxFailures	✓						
spark.files.openCostInBytes				✓	✓		✓
spark.shuffle.file.buffer	✓						✓
spark.cleaner.referenceTracking.blocking		✓	✓			✓	
spark.kryoserializer.buffer.max						✓	

Table 4.6: Important Spark Parameters per benchmark

4.4 Performance Prediction of tuned spark applications

4.4.1 End-to-end

As it has already been mentioned, in order to find the optimal configuration of an application, it is necessary to try multiple candidate configurations and evaluate them based on the execution time. However, since measuring that time for a lot of experiments can be very time consuming and it is not a very good idea for it to be done during the optimization process, we need to find a way to estimate it with as much accuracy as possible. This can be done by adding another stage before the one of the optimization, the modeling stage.

Constructing a model that predicts accurately the performance of any spark application is not a trivial process. Spark workloads can have very different performance behaviors from each other and, thus, one model cannot be accurate enough for all of them. A certain grouping of the workloads needs to take place, so that one model is built for a set of similar applications. In order to find similarities and differences among different workloads, it is necessary to find a way to study the basic characteristics of an application, concerning CPU, memory and network behavior, and use a clustering algorithm in order to split the different applications into groups. The workloads that will be studied and used to construct the different models are the developing benchmarks shown in table 4.1, hoping to extract from them as much useful information as possible. The primary objective here is to produce results that can be generalized to other applications too.

4.4.2 STEP 2: Clustering Inference

4.4.2.1 Production of Representative Signals per Benchmark

As already stated, the information that is helpful at characterizing an application concern CPU, network and memory behavior. Specifically, our framework evaluates the similarity between different workloads by testing their IPC, frequency, cache behavior and memory IO, which are measured with the use of the PCM metrics shown in table 4.3. Therefore, in order to be characterized and clustered the benchmarks needed to be executed alongside PCM program, so that they could be monitored. The configuration was selected to be the default and the input data size large, because a small dataset would lead to a short execution of the application and would bring little useful information.

On each execution a csv file was created, which we will call PCM trace. Every column of this trace corresponds to a metric and can be considered as a signal of that metric, which changes through time. The division of the benchmarks into groups was done with the help of a complex signal that is calculated as the element-wise geometric means of the initial signals. This means that the new signal has the same length as the old ones and its cells contain the geometric means of the corresponding cells of the old signals like this:

$$new = \{geo_t1, geo_t2, ..., geo_tn\}$$

, where geo equals $\sqrt[9]{PhysIPC\% \cdot TotalQPIout \cdot ... \cdot L2HIT}$

In this way, every benchmark has its own complex signal that summarizes its most important characteristics.

4.4.2.2 Kmeans clustering algorithm

The above signals can be used in a clustering algorithm in order to separate the applications into clusters (groups). Since a signal is a kind of time series, the algorithm that was used is TimeSeriesKmeans of Python's tslearn library [34] and the training was done over the time series dataset of the 12 signals.

However, since the benchmarks did not run for the same amount of time, their length differs. With tslearn's help for data series analysis the signals are resampled and reformed to have length of 2000 values. In this way, all three metrics used for both cluster assignment and barycenter computation of kmeans can be used, including euclidean that needs time series of the same length.

After testing different values for the number of clusters to be produced, as well as for the metric to be used for calculating distances (euclidean, dtw, softdtw), we conclude that the best clustering of the applications is the one described in table 4.7 that produces 6 clusters using the metric softdtw with the default number of iterations for the barycenter computation process and gamma parameter equal to 0.005.

Figure 4.7 visualizes the 6 time series clusters that were created. The centroid, which represents the cluster, is depicted with red color and the benchmark signals that belong to each cluster are depicted with gray color.

No Cluster	Benchmarks
Cluster 0	Aggr, Scan
Cluster 1	Bayes,Linear,Sort
Cluster 2	Kmeans, SVM
Cluster 3	Join,GBT,LDA
Cluster 4	PR
Cluster 5	TS

Table 4.7: Clustering developing benchmarks

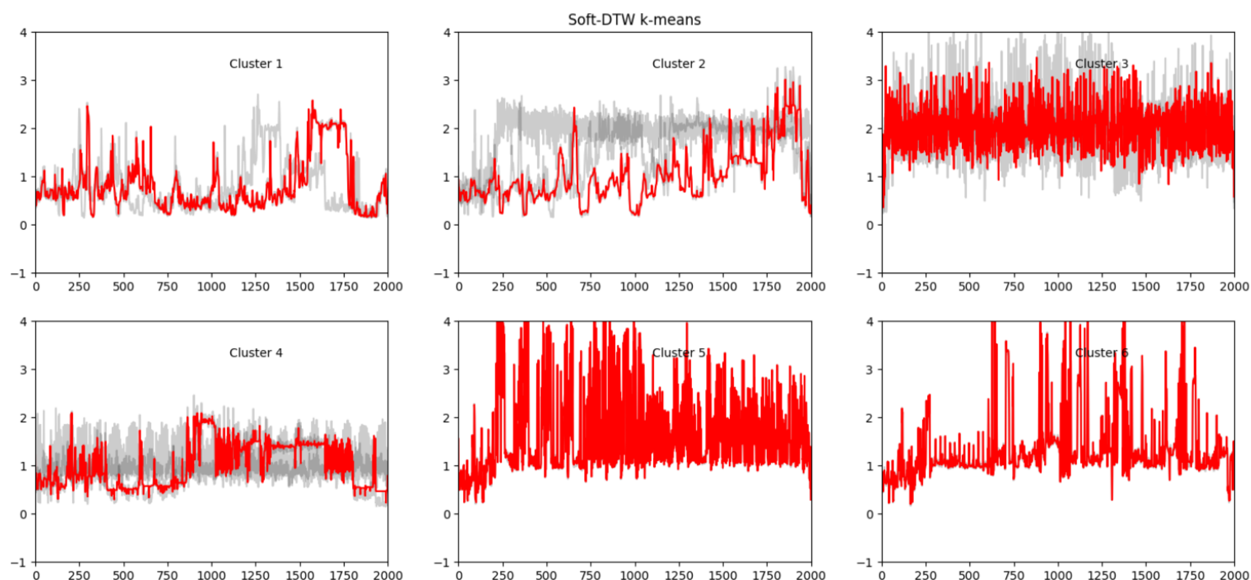


Figure 4.7: Time-series clustering of developing applications

4.4.3 STEP 3: Model Inference

4.4.3.1 Experiments

During the modeling stage, a series of experiments ran so as to collect data that can be used for training a model to predict successfully an application's behavior in random configurations. Initially, like in the previous series of experiments that concerned the parameter analysis, there were produced 300 vectors with random values for the 23 important parameters. This time these values were selected between 6 representative values for every parameter (except for booleans) from within their accepted range. Specifically, for arithmetic values there was selected the default value and both higher and lower values so that the domain is well covered. For the non arithmetic again there were used all the alternatives. Then, every one of the 12 benchmarks that form our knowledge base was executed for the 300 different configurations and for input datasizes tiny, small and large.

Next, another series of experiments followed. The configurations that were used had 22 of the 23 important parameters at their default value and the one left every time had another value selected from the 5 representative ones (except in case of spark.executor.memory where 10 possible values were tested), without the default. Therefore, in every execution only one parameter was written in

the spark properties file. Obviously, the number of experiments that ran for each parameter equal the number of the possible values that it can take, minus the default. The size of the input data at these executions was large.

Finally, in order to be more thorough, another series of experiments was executed. This was similar with in logic with the previous one, except this time the rest 22 important parameters were given random values and not their default. For the assignment of random values on the parameters, OpenTuner was once again used. The size of the input data at these executions was large.

All the above experiments took a significant amount of time, between 1 and 2 months, but this is time spent on the construction of the optimization script and not time that will delay the user of the script, so it is not as bad. From these experiments, we stored not only the execution times but also the PCM traces because we will also study shortly and try to model the benchmarks on an energy level.

4.4.3.2 Description of Model Inputs and Outputs

As aforementioned, not only one model was built for all the applications but instead six, which is as many as the different clusters that were produced. Each model is trained with the collected data of the benchmarks that belong to the cluster it represents and learns their behavior. What it actually does is try to extend its existing knowledge so as to make accurate predictions about the execution time of similar applications in case a random configuration is used.

The features that will be used in the construction of the training sets are the 23 important parameters, the input datasize, and the important information from the PCM trace of the default execution. The training set of a model that corresponds to a cluster with a unique application will have the above information for this particular application only, while those who have more will have this information for all of them. The features that concern the PCM trace are 9 and they are calculated as the average of each important metric/column of the trace. Therefore, instead of using time series and passing a signal as input to the model, we hold only one value per signal, which is calculated as the average of all its values without resampling. These extra features that are relevant to PCM help the model learn the connection between the basic characteristics of an application and its execution time. The target output or else the predicted value is the running time of the application.

The figure that follows shows the logic that was described above, with the interior of the model being a black box, which will be studied in the next section.

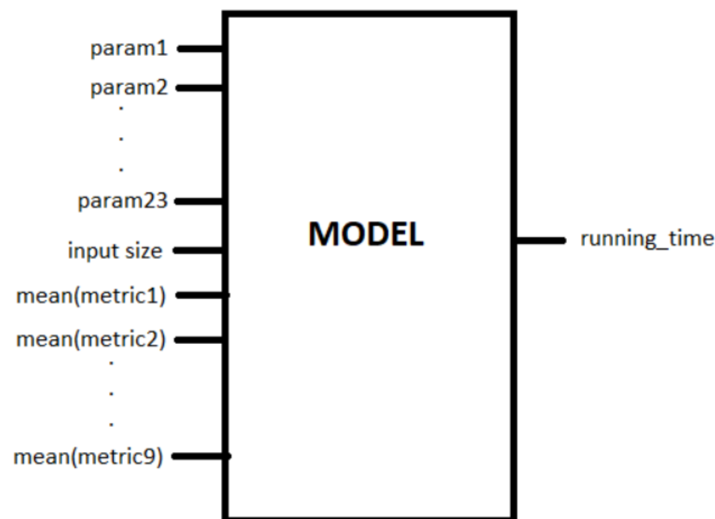


Figure 4.8: Description of features and target output of the model

4.4.3.3 Comparing Algorithms for the Performance Models

Our performance model is based upon a regression algorithm because we need to predict a continuous variable like execution time. A great category of such algorithms is the linear (linear, ridge, lasso, elastic net, bayes ridge and SGD), in which the dependent value is a linear combination of the independent (features) like this:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

However, there are also other algorithms that can model more complex relations between the features and the target output, such as SVM, K Nearest Neighbors, Gaussian Process, Decision trees, Random forest and Multi-layer Perceptron. Since our problem is complex enough, with 33 features, these algorithms will probably achieve higher accuracy.

With the help of Python's library sklearn [35] we used all the above algorithms to build different models for the clusters and then we evaluated them in order to compare them to each other and find the best. Before they are fed to the model, the training sets that were constructed pass through the process of standard scaling, so that they have mean value equal to 0 and standard deviation equal to 1 (normal distribution). This was done because it was observed that it causes a slight improvement on accuracy.

Most of the algorithms that were used, offer some parameters to the programmer for tuning that play a vital role in the model's learning capacity. For this reason, a Randomized search was executed in order to test random combinations of values for them and find out towards which direction lie the most effective ones based on their accomplished score. After that, we define in a grid the winning values and some that are close to them and run a Grid search that tests all possible combinations and finds the most effective configuration. This optimal configuration of each algorithm was used when running it and collecting the accuracy (score) it achieved.

The table that follows shows in every column the best score that was achieved for this particular cluster for each of the regression algorithms that were used. The scores where calculated using the

method ”score” - that uses the R^2 metric - in different testing sets that were selected with KFold’s help. The KFold method splits the train/test datasets into $n_splits = k$ consecutive folds, after shuffling them, as we defined. Each fold is then used as a validation set while the k - 1 remaining folds form the training set. Every one of the k iterations gives a score that is held in a list and at the end of the loop the average of the list is calculated. The final score that is described in table 4.8 is the maximum score that was achieved among all the different executions for different values in variable n_splits.

Algorithm	**Score-C0**	**Score-C1**	**Score-C2**	**Score-C3**	**Score-C4**	**Score-C5**
Linear	0.51	0.36	0.40	0.34	0.39	0.41
Ridge	0.52	0.36	0.40	0.34	0.39	0.41
Lasso	0.16	0.31	0.31	0.33	0.35	0.39
ElasticNet	0.43	0.30	0.29	0.30	0.33	0.38
BayesRidge	0.52	0.36	0.40	0.34	0.39	0.42
SGD	0.51	0.32	0.19	0.22	0.38	0.43
SVR	0.75	0.36	<0	0.28	0.27	0.38
kNeighbors	0.78	0.35	0.50	0.23	0.28	0.16
GaussProc	0.77	0.51	0.56	0.49	0.36	0.31
DecisTree	0.83	0.83	0.90	0.88	0.80	0.56
RF	0.90	0.87	0.93	0.92	0.85	0.73
MLP	0.87	0.71	0.88	0.76	0.50	0.40

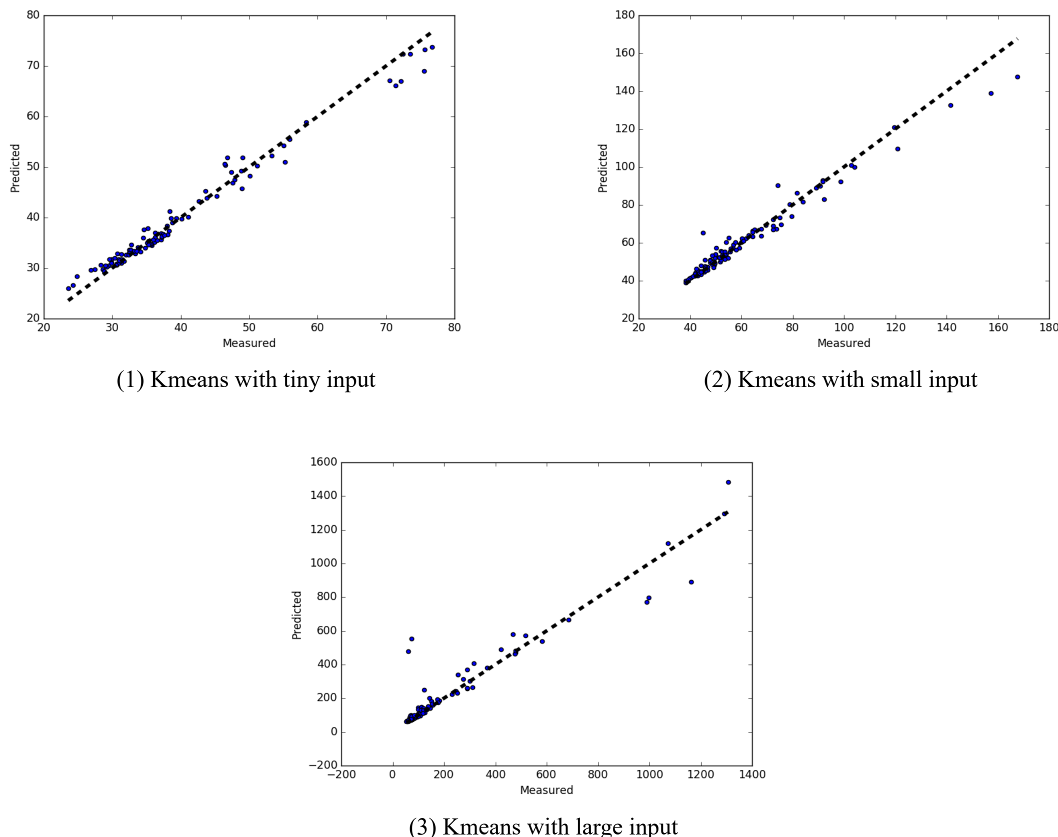
Table 4.8: Model scores of each cluster per algorithm

Obviously the linear algorithms, as anticipated, cannot train highly accurate models for our 33-dimension problem. The SVM, K-neighbors and Gausian Process regressors, also do not have a good enough behavior. On the other hand, the MLP regressor with a neural network of 100 neurons in only one hidden layer can produce much better results but only for clusters whose training set contains data from more than one applications. Finally, the algorithms that seem to make the most effective modeling are the two that are based on decision tree learning, with random forest being the best of them in every cluster. The models built with the Random Forest algorithm are accurate enough, achieving in most cases scores greater than 0.90 and in the worst case a score of 0.73, which again is the best one for this particular cluster.

Since the models that learn better our problem and make more accurate predictions are those that are built with the Random Forest algorithm, these are the ones that were selected for modeling the clusters. This algorithm operates by constructing a multitude of decision trees at training time and outputting the mean prediction of the individual trees. It inserts randomness at splitting the nodes when a tree is being constructed, such as selecting a random subset of features, in order to reduce the model’s variance. In this way, it manages to outmatch Decision Tree learning because of its lack of overfitting. This means that Random Forest algorithm does not train the system to stay too close to a specific dataset because this would cause problems in its ability to generalize its predictions.

Although the R^2 is a good metric to evaluate a regression model’s accuracy statistically, it might hide large errors for particular predictions due to outliers. To address this issue, below we present the error distribution of our prediction models using scatter plots. The three scatter plots depict 100 real measurements and the 100 corresponding model predictions for benchmark Kmeans and for three different inputa datasize categories for 100 randomly selected configurations. The x axis represents the real measurements and the y denotes the predicted execution times. This figure

clearly shows that the models are fairly accurate across the entire Spark configuration space; all 100 data points for each benchmark are located around the corresponding bisector, indicating that the predictions are close to the real measurements. This indicates that there are not many outliers in the predictions of our performance models, which is good for quickly finding the optimum configuration for a Spark workload.



(1) Kmeans with tiny input

(2) Kmeans with small input

(3) Kmeans with large input

Figure 4.9: Prediction of execution time versus real measurement for different input sizes of Kmeans

4.4.3.4 Energy Models

However, in every program execution, time is not the only performance metric of interest. The greatest concern of all programmers is the trade-off between minimizing execution time and energy consumption. Therefore, we also tried to build a model to predict the energy that was consumed by each differently configured spark execution.

In order to measure the energy consumption, the PCM trace was used once again. It offers information about DRAM and processor energy consumption in Joules. These two metrics are two different target outputs, so there were built distinct models for the energy modeling. The algorithm that was used for modeling is Random Forest regression and the clustering remains the same. The score results are shown in table 4.9.

Energy	Score-C0	Score-C1	Score-C2	Score-C3	Score-C4	Score-C5
PROC	0.78	0.86	0.91	0.74	0.91	0.60
DRAM	0.73	0.88	0.91	0.73	0.92	0.50

Table 4.9: Model scores of each cluster per energy metric

Obviously, predicting the energy has not been as accurate as predicting the execution time, especially on the DRAM level. Improving these models, though, would need deeper digging on an architecture level, which would delay us significantly. Therefore, the study of energy consumption was set aside and the minimize energy objective was finally not used alongside the minimize time objective at the optimization stage for the time being.

4.5 STEP 4: Determining the Optimal Configuration of Spark Applications

The final step that completes our framework is finding the optimal configuration to run each application so that its execution time is minimized. This was done with the help of an optimizing algorithm, which can explore different candidate solutions and move through the search space based on the time results each of them brings, with the objective to minimize execution time. This optimizing process can be done easily with the help of the OpenTuner framework [12, 13].

4.5.1 Comparing Optimization Algorithms

OpenTuner offers a number of techniques (42) that differ on the way the search space is being explored. None of them guarantees that it will end up finding the global minimum of the execution time, but only a good enough local minimum. It is also possible to build a metatechnique, which is a technique made up of a collection of other techniques. When the metatechnique gets allocated tests, it incrementally decides how to divide theses tests among its sub-techniques. The most famous metatechnique is the AUC Bandit, which has been created to find the optimal solution to the problem of the multi-armed bandit. This is the problem of picking levers to pull on a slot machine with many arms each with an unknown payout probability.

Unfortunately, there is no technique that always outperforms all the others and that should be selected in every optimization problem. However, there are certain techniques, such as the Bandit, the particle swarm optimization, differential evolution and the genetic algorithms that usually produce interesting results, so the selection will be done among those.

After running the optimization process on the same pair of application-dataset for all the above techniques, it was obvious that the particle swarm optimization and the differential evolution could not compete with the others as they produced significantly worse results. The bandit and the genetic algorithm came up with quite similar results and therefore we selected the one that seemed insignificantly better and took less time to run, which was the genetic algorithm.

4.5.2 Genetic Algorithm

The Genetic algorithm belongs to the larger class of evolutionary algorithms. They are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection.

In a genetic algorithm, a population of candidate solutions to an optimization problem evolves toward better solutions. Each candidate solution has a set of properties which can be mutated and

altered. The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

The OpenTuner offers five different genetic algorithms that differ on the crossover operator. Specifically, they differ on the way that the permutation variables, that correspond to the parents, will be combined with one another in order to produce the offspring. With the word permutation we mean the OpenTuner's parameter type that assigns an order to a given list of values. All of them have a mutation rate of 0.1 and a crossover rate of 0.8. The five different algorithms are:

1. *ga-PX*: As the flag "PX" states, a random cut point is chosen in the first parent and all of his elements up to this point are reordered according to their order in the second parent.

2. *ga-PMX*: As the flag "PMX" states, a random section of the first parent, the size of which is controlled by a hyper-parameter, is replaced with the corresponding section in second. The displaced elements in the first parent are moved to the old position of the elements displacing them.

3. *ga-CX*: As the flag "CX" states, elements of the first parent are repeatedly replaced with the element at the same index in the second one. This is done until a cycle is reached and the original permutation is valid again. The initial replacement index is random.

4. *ga-OX1*: As the flag "OX1" states, a subpath from the second parent is exchanged into the first, while maintaining the order of the remaining elements in the first parent. The size of the exchanged section is controlled by a hyper-parameter.

5. *ga-OX3*: As the flag "OX3" states, a subpath from the second parent is exchanged into the first, while maintaining the order of the remaining elements in the first parent. The two cutting points for the sub-paths are different, as opposed to the previous algorithm. The size of the exchanged section is controlled by a hyper-parameter.

The slightly better algorithm from the above, which was the one that was finally selected, is the ga-PX.

In order to run the optimizer, a configuration manipulator was created, defining six different values for all the arithmetic parameters except for the spark.executor.memory. This parameter was considered very significant and, since it has a large domain, eight possible values were chosen instead of six. For the non arithmetic parameters, the manipulator once again contained all the possible alternative values. Next, the run function was defined in such a way that it would evaluate the fitness of the given configuration by asking the appropriate model for this configuration's output time. In order to find out which the appropriate model is, the optimizer runs a script that searches for the csv file that contains the PCM trace and assigns to it a label corresponding to a specific cluster.

4.5.3 Integration of proposed framework with apache spark

Our work offers a wrapper script that takes the standard spark-submit command and internally runs the workload one time to collect the PCM trace and assign a cluster to it. Then, it uses the appropriate model for the optimizer to evaluate configurations and end up to the optimal one, with which the spark-submit command will finally be executed. This process described is shown in figure 4.10.

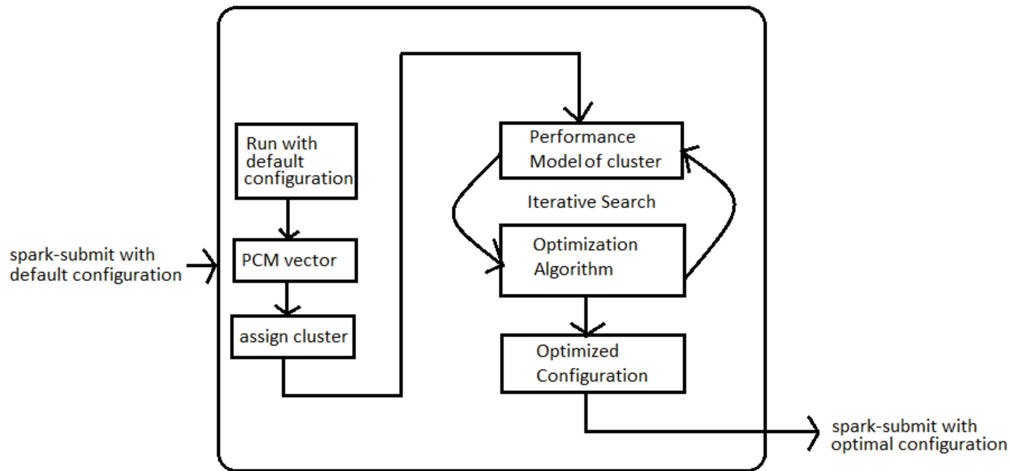


Figure 4.10: Description of the wrapper script of our framework

Chapter 5

Experimental Results and Evaluation

5.1 Exploration on Developing Workloads

Our autotuning framework was executed for all the different developing application-dataset pairs and returned the best configuration and the predicted execution time of it. It is reminded that the optimization's process results are as accurate as the model predictions allow them to be. Next, each application ran with the selected best configurations ten times in order to measure the average time that its execution actually cost. The following table shows the default, the best measured and the best predicted execution time of each application, with three columns per data size category.

	Tiny			Small			Large		
App	**Default**	**Meas**	**Pred**	**Default**	**Meas**	**Pred**	**Default**	**Meas**	**Pred**
Bayes	41.921	30.462	30.015	43.963	32.480	30.469	63.024	48.239	43.07
Kmeans	29.787	23.791	26.207	71.889	46.270	38.783	1351.1	58.202	55.367
SVM	44.125	31.573	35.735	54.002	36.108	39.423	2643.2	82.894	82.707
Aggreg	76.123	61.873	59.364	76.632	63.059	58.638	79.847	66.129	59.604
Join	77.398	68.259	65.426	131.53	75.034	86.813	130.63	79.247	114.94
Scan	58.39	56.883	56.881	60.215	56.856	56.549	63.701	60.516	57.991
PR	23.637	22.467	24.1	30.791	26.433	27.704	265.82	79.75	71.157
Linear	43.702	25.257	32.541	106.89	35.653	37.761	248.56	60.653	61.752
GBT	41.126	30.064	31.839	88.265	57.63	60.02	540.97	249.33	219.63
Sort	29.908	19.844	22.14	23.696	22.749	23.496	29.767	24.488	27.389
LDA	116.54	72.374	117.72	644.84	248.55	167.45	1580.6	632.31	607.77
TS	30.843	21.287	21.659	40.626	25.29	28.467	197.52	52.782	54.261

Table 5.1: Comparing default, best measured and best predicted execution time for developing benchmarks

The figures that follow visualize the above results. The figure for the tiny datasize follows in the next page. It shows that for every benchmark we have achieved reduction of its execution time except for Scan, which almost did not improve at all. This is because Scan is a compute intensive program, with the default value for the CPU-related parameters being suitable, and with the memory related parameters not being important enough. Also, the measured time is very close to the predicted one, except for the LDA benchmark, with which we achieved a remarkable optimization but not a good prediction accuracy. The lack of prediction accuracy is not unexpected since figure 4.7 shows that the 4th cluster, where LDA benchmark belongs, has some noise that can sometimes lower the prediction accuracy.

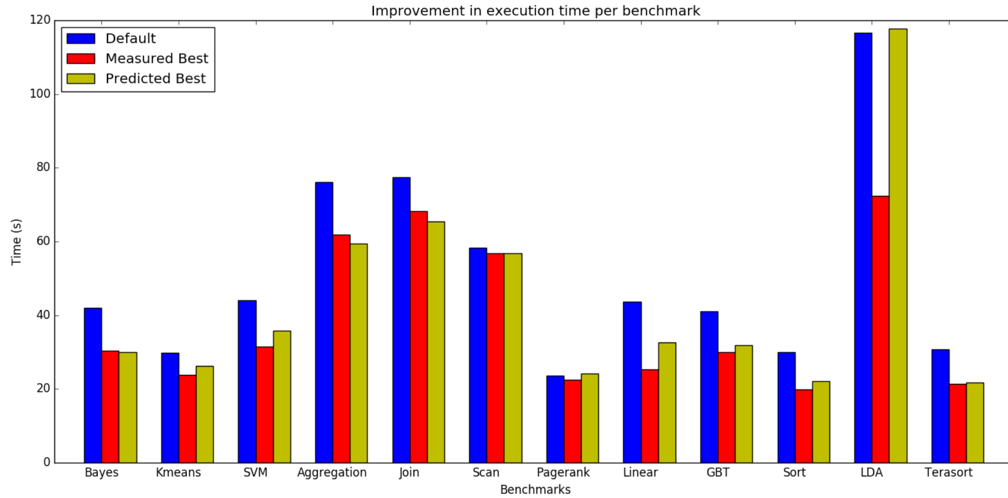
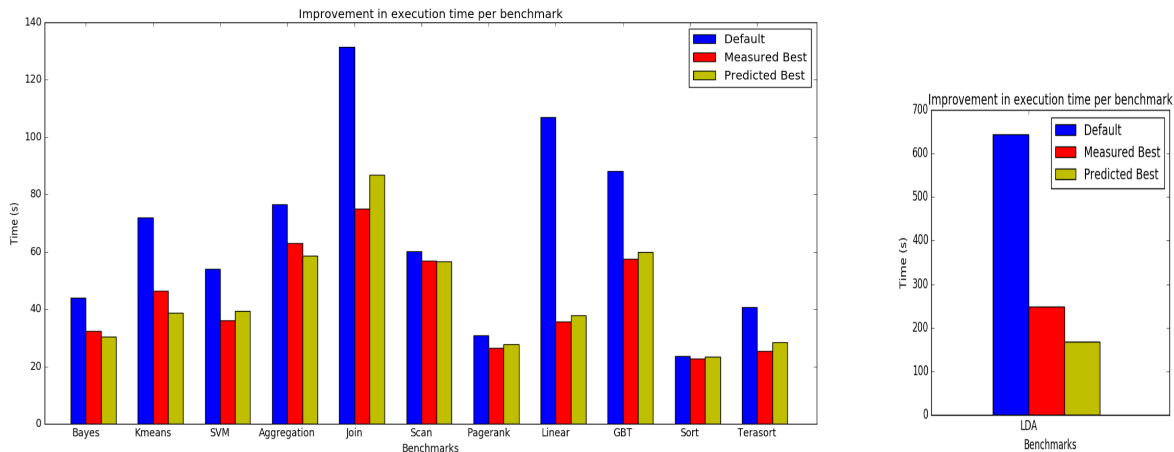


Figure 5.1: Comparing default, best measured and best predicted execution time for developing benchmarks and tiny size of input data

For small and large datasets there are presented two distinct figures for each category because of the different scaling in the time axis.

The figure for the small datasize is:



(1) Comparing smaller execution times

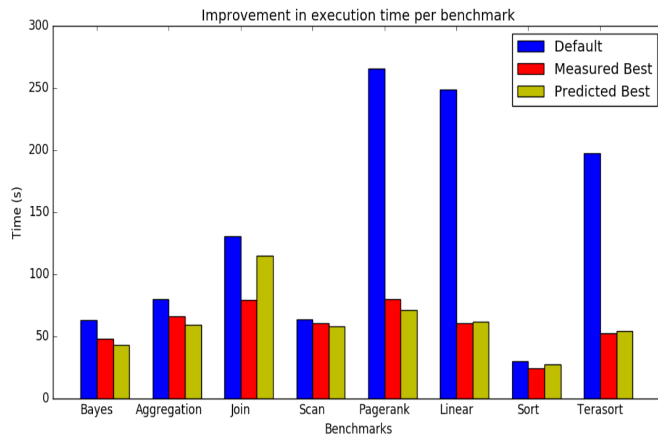
(2) LDA app with high execution time

Figure 5.2: Comparing default, best measured and best predicted execution time for developing benchmarks and small size of input data

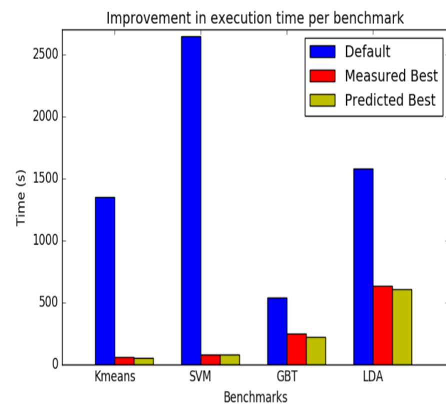
For small input datasize again we have achieved reduction of the execution times, and in the cases of Join, Linear and LDA a very important one indeed, and it starts being obvious that while the input data size increases these improvements will become more and more significant since the memory related parameters will become of great significance. In case of Scan and Sort the execution time almost did not improve at all, because as we stated before the default values happened to be nearly the best that could be acquired. The prediction accuracy remains very good, maybe except for LDA again, which was underestimated and Join that was slightly overestimated, for the

same reason we stated before about the fourth cluster.

The figure for the large datasize is:



(1) Comparing smaller execution times



(2) Comparing larger execution times

Figure 5.3: Comparing default, best measured and best predicted execution time for developing benchmarks and large size of input data

For large input datasizes the only model prediction that is not very good is the one of the join benchmark, for the reason that we have stated. The improvement is very important on nearly all the applications, and as we had predicted is very much greater as the data size increases. Once again Scan and Sort almost did not improve at all.

In order to visualize better the improvement shown in the above three diagrams we present below a speedup diagram that shows what fraction of the default execution time is the best that we achieved, per benchmark and input datasize.

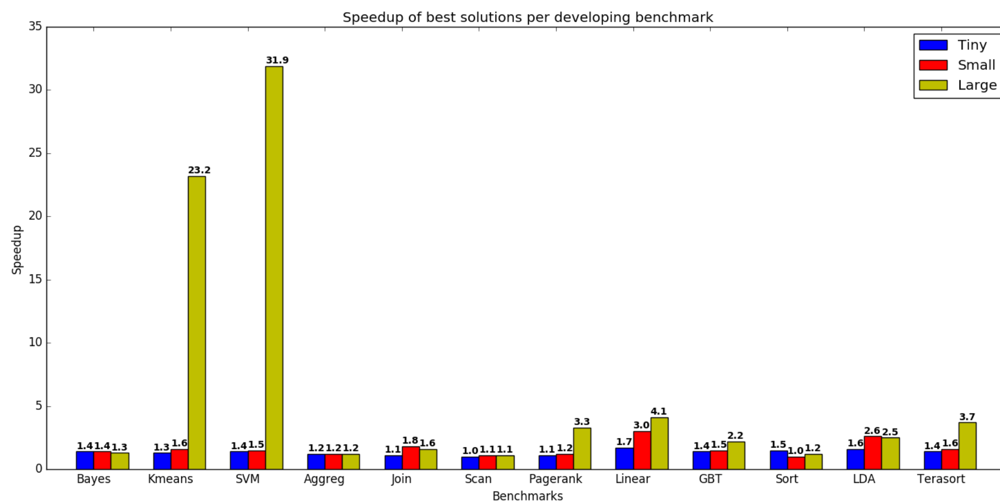


Figure 5.4: Comparing speedups for all the developing benchmarks and inputs

There are plenty of interesting observations to be made here. First, the execution times of all program-pairs running with our optimal configurations are shorter than those of the same program-

pairs running with the default configurations. Second, our framework improves the performance significantly for all the developing benchmarks except Scan, where we obtain marginal performance improvements. The maximum speedup achieves 31.9× when SVM runs with 20 GB of data, which can be explained by the fact that SVM is a benchmark both compute and data intensive and therefore there is a great number of parameters that affect significantly its performance. The average speedup for all the program-input pairs is 3.07×. Third, the performance improvement for a benchmark made by our framework generally increases when the input data size of that benchmark increases. This is a very significant property for our era because one of the important features of data analytics is that the amount of data increases rapidly. Therefore, our results demonstrate that our framework can improve in most cases the performance of Spark workloads significantly.

5.2 Exploration on Unseen Workloads

The great innovation of our tuning framework is that it can also be used to optimize the execution of an unseen application that has not been used in our knowledge base. This is the reason why we put so much effort on trying to find the connection between the behavior of a workload on an architecture level and the execution time.

5.2.1 Cluster Assignment and Model Scores

In order to run a new application optimally using our method, the only experiment that needs to be done is running the workload with the default configuration and large size, so that the PCM trace file is collected. From this file, as described before, new signals are produced with the help of the geometric means and create a time series dataset. After the clustering model that was built before has been loaded, the method ”predict” is applied on the dataset and it assigns a label on this particular application we ran. This label shows which cluster corresponds to this workload and the results for the validating applications are shown in the table 5.2.

Benchmark	Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
ALS		✓				
LR			✓			
RF		✓				
PCA				✓		
SVD		✓				
WC		✓				

Table 5.2: Assigning validating benchmarks to clusters

The above assignment shows which of the previously created models is the one that should be used to predict the execution time of each of the unknown applications. In order to evaluate the accuracy of the models in the unknown workloads, we need to run a series of experiments so that we can measure the real execution times of some configurations and then compare them to the predicted ones. Therefore, each of the validating benchmarks was executed for the 100 first different configuration vectors that were produced in the first series of experiments we ran, and for input data sizes tiny, small and large. The measured execution times of these experiments are used as testing sets in the model of the cluster to which they belong, so that its prediction ability can be

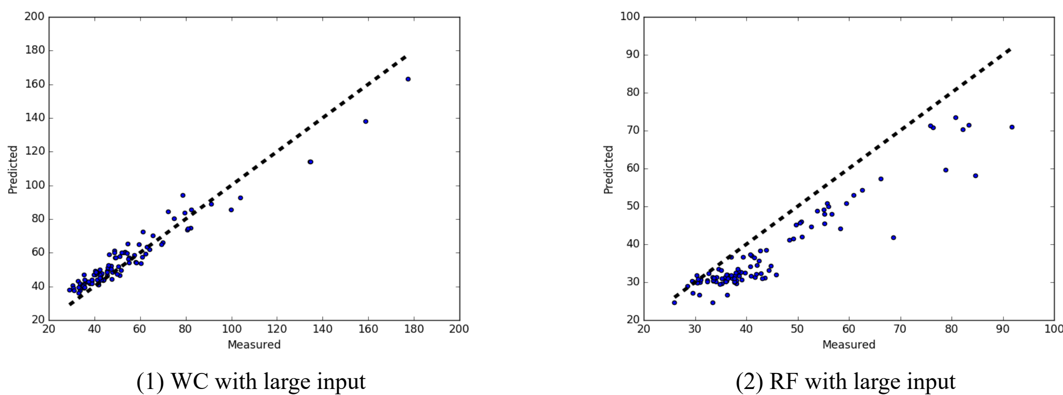
evaluated. The table that follows shows for each application the cluster to which it is assigned and the R^2 score that the corresponding model achieves for every input data size category.

Benchmark	No Cluster	Score-Tiny	Score-Small	Score-Large
ALS	1	< 0	< 0	< 0
Logistic	2	0.77	0.81	0.53
RF	1	0.92	0.77	0.66
PCA	3	0.64	0.91	< 0
SVD	1	0.80	< 0	< 0
WC	1	0.68	0.70	0.92

Table 5.3: Score per validating benchmark

Evidently, our methodology produces accurate enough results for most of the validating benchmarks and for all the datasize categories. The ALS benchmark, though, cannot be accurately predicted whatever the input data size.Also, with SVD and PCA the good predictions cannot keep up with the input data size increase. In spite of these inaccuracies, our modeling technique's achievements are remarkable. It accomplishes a satisfying accuracy in many cases in predicting unseen applications and so, the optimal configuration that is proposed even in this case is very close to the actual best, which is more than it has ever been achieved before by other autotuning frameworks.

We also present two scatter plots that correspond to different benchmark-input pairs. The first represents the WC-large pair that, according to table 5.3, has a score of 0.92 and the second represents the RF-large with a score of 0.66. This difference in the accuracy is obvious, because in the second figure the points are not as close to the bisector.



(1) WC with large input (2) RF with large input

Figure 5.5: Error distribution illustrating prediction versus real measurement for validating benchmarks

5.2.2 Result Analysis and Evaluation

In order to test also the optimization results, our autotuning framework was executed for all the different validating application-dataset pairs and returned the best configuration and the predicted execution time of it. Next, each application ran with the selected best configurations ten times in order to measure the average time that its execution actually cost. The following table shows the default, the best measured and the best predicted execution time of each application, with three columns per data size category.

	Tiny			Small			Large		
App	**Default**	**Meas**	**Pred**	**Default**	**Meas**	**Pred**	**Default**	**Meas**	**Pred**
ALS	30.876	34.447	25.271	36.337	41.479	26.984	86.214	67.102	26.236
LR	36.33	25.618	26.197	45.932	31.292	35.896	1516.179	178.573	56.771
RF	32.467	21.659	22.324	36.500	24.652	24.199	32.864	29.892	25.477
PCA	48.382	37.861	32.252	93.429	62.576	62.232	322.852	185.411	77.172
SVD	38.896	29.723	28.213	71.889	74.284	27.525	337.659	350.852	30.762
WC	31.790	20.810	26.273	31.792	24.825	29.725	62.492	36.551	36.470

Table 5.4: Comparing default, best measured and best predicted execution time for validating benchmarks

The figures that follow visualize the above results. The figure for the tiny datasize shows that for every benchmark we have achieved reduction of its execution time except obviously for ALS that could not be well predicted by our models, according to table 5.3. As we have stated, our autotuning framework can be as accurate as the model allows it to be, so the ALS results are never to be trusted. In all other cases, however, where the models had a good enough accuracy, we have accomplished a satisfying improvement.

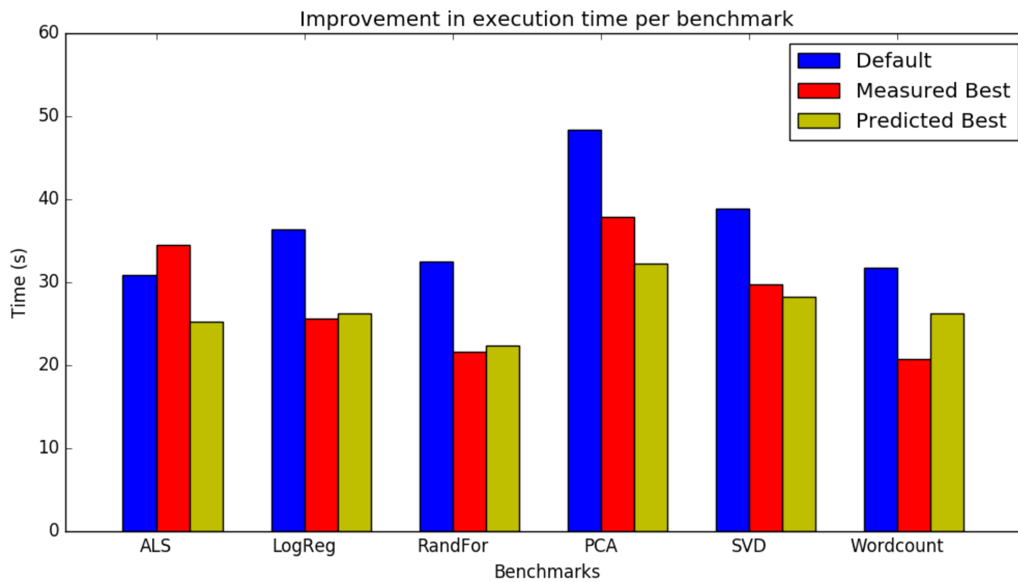


Figure 5.6: Comparing default, best measured and best predicted execution time for validating benchmarks and tiny size of input data

For small input datasize again we have achieved reduction of its execution time except obviously for ALS and SVD that could not be well predicted by our models, according to table 5.3. The predictions are, however, accurate enough for the rest of the cases, as expected. Once again we notice that as the input data size increases the improvements tend to be more significant, which begins to show especially with the PCA benchmark.

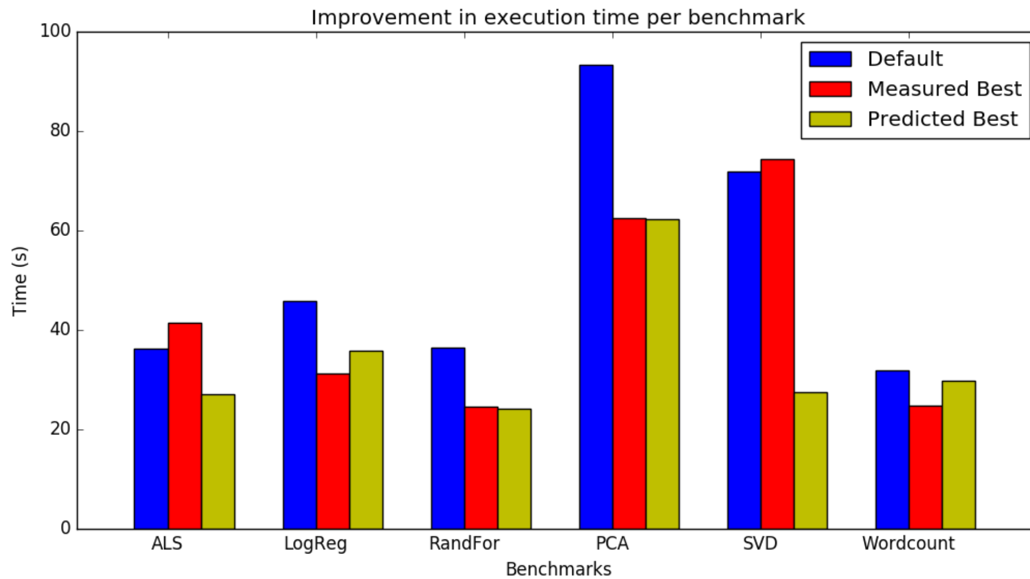
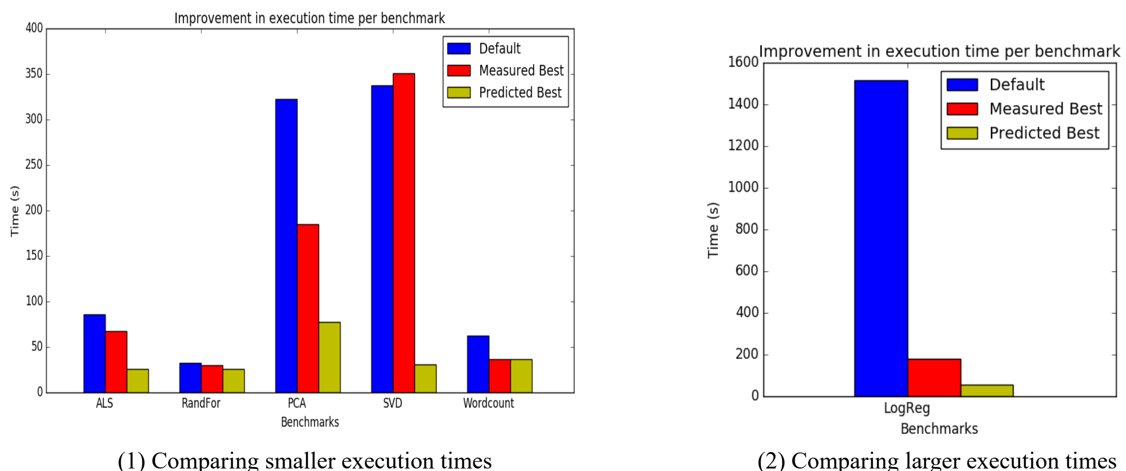


Figure 5.7: Comparing default, best measured and best predicted execution time for validating benchmarks and small size of input data

For large datasets there are presented two distinct figures because of the different scaling in the time axis. We notice this time that the only benchmark that did not improve is the SVD. However, the improvement of PCA and ALS is not to be trusted because, as the table 5.3 shows it has not been accomplished with the help of our models but out of luck. It is obvious, however, that because of its data intensive character PCA can afford to be greatly improved. The RF benchmark does not accomplish a good enough reduction of the execution time, even though the improvement becomes greater as the input data become larger, probably because of it not being data intensive and the memory-related parameters having low importance.



(1) Comparing smaller execution times (2) Comparing larger execution times

Figure 5.8: Comparing default, best measured and best predicted execution time for validating benchmarks and large size of input data

In order to visualize better the improvement shown in the above three diagrams we present

below a speedup diagram that shows what fraction of the default execution time is the best that we achieved, per benchmark and input datasize. The diagrams contain only the program-input pairs which our models could predict accurately and not those that were improved by chance.

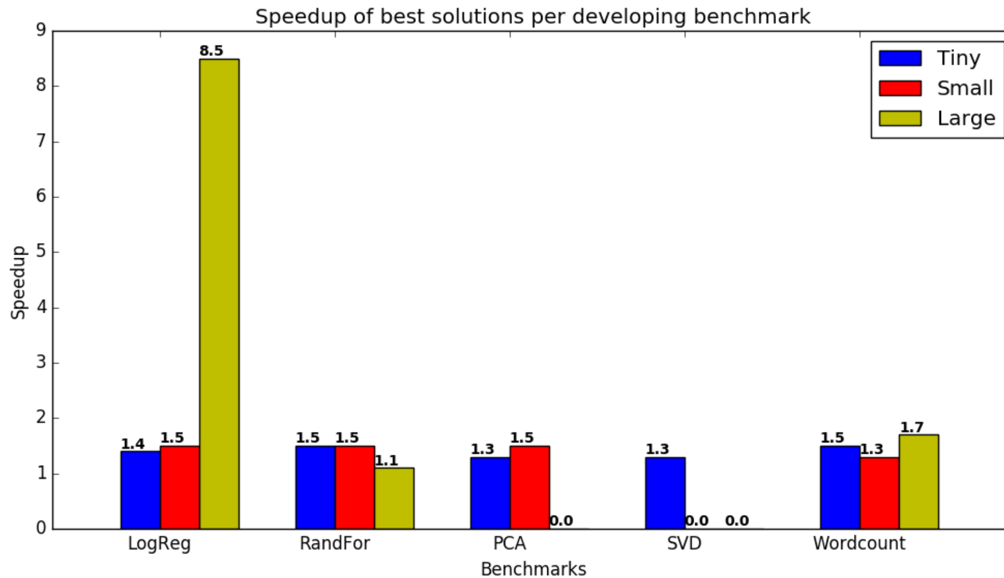


Figure 5.9: Comparing speedups for all the validating benchmarks and inputs with no negative scores

We notice that the execution times of all program-data pairs with predicting R^2 score of our models greater than zero, when running with our optimal configurations are shorter than those of the same program-pairs running with the default configurations. Also, our framework improves the performance significantly for all the validating benchmarks. The maximum speedup achieves 8.5× when LR runs with 8 GB of data. The LR benchmark, as well as the representative of the cluster benchmarks Kmeans and SVM we commented before, present a remarkable improvement. It is an application both compute and data intensive so there are plenty parameters that affect significantly its performance end therefore it gets so much improved. The average speedup for all the program-input pairs is 2.01×. Finally, the performance improvement for a benchmark made by our framework generally increases when the input data size of that benchmark increases.

5.3 Comparison with a Model Free Optimization Process

The insertion of the model in our optimization process has been made, as it has already been explained, so that our optimizer can evaluate candidate solutions with negligible delays. In this section we will try to apply the optimizer directly on spark, testing 200 different configurations, and check the time overhead and the number of different configurations that need to be tested for each program-input pair to find a solution as good as ours. The results are shown in table 5.5.

We notice that sometimes the optimizer that evaluates the candidate solutions directly on spark can produce results as good as ours with a small time overhead of less than ten minutes. In many cases though, this could take a much greater amount of time, such as nearly an hour or even several hours. This difference has undoubtedly a lot to do with the model accuracy that each benchmark-input pair can achieve. However, in the general case, our modeling technique saves a lot of time

	Small		Large	
Benchmark	**No configs**	**Time**	**No configs**	**Time**
Bayes	35	1645s	4	292s
Kmeans	4	450s	12	2831s
SVM	> 200	> 12763s	4	442s
Aggr	32	2658s	8	656s
Join	8	905s	36	3356s
Scan	4	321s	4	364s
PR	4	156s	36	5096s
Linear	8	611s	8	818s
GBT	60	4403s	8	3020s
Sort	4	177s	4	160s
LDA	4	889s	4	2419s
TS	> 200	> 9289s	60	4750s
ALS	4	213s	4	406s
LR	134	6210s	104	23533s
RF	8	303s	28	1314s
PCA	76	6461s	4	1038s
SVD	4	325s	4	1135s
WC	4	133s	4	218s

Table 5.5: Comparing our optimization process with the one applied directly on spark

and makes our optimizer actually useful for a programmer who does not have time to spare, so it should most definitely be preferred over the on spark optimizer.

Chapter 6

Conclusion and Future Work

In this thesis, we first investigate the impact that the different Spark parameters have on the performance of Spark workloads. We find that 23 of them affect the performance significantly and, thus, reconfiguring them can lead to remarkable differences in execution time.

However, manually configuring Spark workloads without in-depth knowledge of it is extremely challenging because of the massive number of configuration parameters that exist and their wide range of possible values. To address this issue, we propose and implement an automated framework for auto-tuning of spark applications. It first collects basic architectural characteristics of a workload and uses them to cluster different applications and construct accurate performance models for each cluster. Subsequently, our framework employs genetic algorithm to search the optimum configuration for each workload by taking the performance predicted by its performance model and the configuration parameter values as inputs. This pioneer idea to cluster applications before building models for them, offers the chance to use the framework even on unseen applications that have not been used to train none of the different performance models, and achieve in most cases satisfying results. We use six representative unseen Spark workloads, each with three input data sets to evaluate our framework. The results show that it can speed up most of these 18 program-input pairs up to 8.5× and with an average of 2.01×, achieving generally more speedup when the sizes of workload input data increase, which is a very nice property for big data analytics.

In the future it is worth further experimenting with the use of our framework on different machine setups. Primarily, there is need to experiment in a distributed way, on a multi-node cluster. In this way we will also test the cluster deploy mode of yarn cluster manager, or even change the cluster manager to kubernetes and hopefully even notice greater speedups.

Also, as aforementioned, a complete study of our problem on an energy level would be very interesting. It is worth finding the right architectural characteristics needed to be taken into account when building the models in order to make them accurate enough. Then it will be possible to modify, or to be more accurate, to make an extra version of our wrapper script that sets its objective to both minimize time and energy. These values, of course, will once again come from the performance and energy models respectively.

References

[1] Zhibin Yu, Zhendong Bei and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. In Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18). ACM, New York, NY, USA, 14 pages.
`https://doi.org/http://dx.doi.org/10.1145/3173162.3173187`

[2] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, S. Feng, Configuring in-memory cluster computing using random forest, Future Generation Computer Systems (2017),
`http://dx.doi.org/10.1016/j.future.2017.08.011`

[3] Apache Spark. Tuning Spark,
`http://spark.apache.org/docs/latest/tuning.html`

[4] Liang Bao, Xin Liu and Weizhao Chen. 2018. Learning-based Automatic Parameter Tuning for Big Data Analytics Frameworks. https://arxiv.org/abs/1808.06008 [cs.SE]

[5] T. Chiba and T. Onodera, "Workload characterization and optimization of TPC-H queries on Apache Spark," 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Uppsala, 2016, pp. 112-121, doi: 10.1109/ISPASS.2016.7482079.

[6] A. Fekry, L. Carata, T. Pasquier, A. Rice and A. Hopper, "Towards Seamless Configuration Tuning of Big Data Analytics," 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Dallas, TX, USA, 2019, pp. 1912-1919, doi: 10.1109/ICDCS.2019.00189.

[7] K. Wang, M. Maifi Hasan Khan, N. Nguyen and S. Gokhale, "A Model Driven Approach Towards Improving the Performance of Apache Spark Applications," 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Madison, WI, USA, 2019, pp. 233-242, doi: 10.1109/ISPASS.2019.00036.

[8] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In Proceedings of the 12nd USENIX Symposium on Networked Systems Design and Implementation (NSDI) (NSDI'15). USENIX Association, Oakland, CA, 293–307.

[9] Y. Zhao, F. Hu and H. Chen, "An adaptive tuning strategy on spark based on in-memory computation characteristics," 2016 18th International Conference on Advanced Communication Technology (ICACT), Pyeongchang, 2016, pp. 484-488, doi: 10.1109/ICACT.2016.7423442.

[10] G. Wang, J. Xu and B. He, "A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning," 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE

2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, 2016, pp. 586-593, doi: 10.1109/HPCC-SmartCity-DSS.2016.0088.

[11] HiBench Benchmark Suite,
https://github.com/Intel-bigdata/HiBench

[12] J. Ansel et al., ”OpenTuner: An extensible framework for program autotuning,” 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), Edmonton, AB, 2014, pp. 303-315, doi: 10.1145/2628071.2628092.

[13] OpenTuner,
https://github.com/jansel/opentuner

[14] S. Chae and T. Chung, ”DSMM: A Dynamic Setting for Memory Management in Apache Spark,” 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Madison, WI, USA, 2019, pp. 143-144, doi: 10.1109/ISPASS.2019.00024.

[15] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) (HotCloud’10). USENIX Association, Boston, MA, 1–8.

[16] Apache Spark - Lightning-fast unified analytics engine,
https://spark.apache.org/

[17] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012.

[18] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. Journal of Machine Learning Research (JMLR), 2016.

[19] Kruskal-Wallis Test,
https://en.wikipedia.org/wiki/Kruskal-Wallis_one-way_analysis_of_variance

[20] Genetic Algorithms,
https://en.wikipedia.org/wiki/Genetic_algorithm

[21] Random Forest Algorithm,
https://en.wikipedia.org/wiki/Random_forest

[22] Apache Hadoop YARN,
https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[23] HDFS Architecture Guide,
https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[24] Juwei Shi, Yunjie Qiu, Umar F. Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, Fatma Ozcan, Clash of the titans: Mapreduce vs. spark for large scale data analytics, Proceedings of VLDB Endowment, 2015,
https://doi.org/10.14778/2831360.2831365

[25] K. Li, X. Tang, B. Veeravalli and K. Li, ”Scheduling Precedence Constrained Stochastic Tasks on Heterogeneous Cluster Systems,” in IEEE Transactions on Computers, vol. 64, no. 1, pp. 191-204, Jan. 2015, doi: 10.1109/TC.2013.205.

[26] Herodotos Herodotou. 2011. Hadoop Performance Models.
https://arxiv.org/abs/1106.0940

[27] Herodotou Herodotos, Babu Shivnath. (2011). Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. Proceedings of the VLDB Endowment. 4. 1111-1122. 10.14778/3402707.3402746.

[28] Herodotou Herodotos, Lim Harold, Luo Gang, Borisov Nedyalko, Dong Liang, Cetin Fatma, Babu Shivnath. (2011). Starfish: A Self-tuning System for Big Data Analytics. CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings. 261-272.

[29] Adem E. Gencer, David Bindel, Emin G. Sirer, Robbert van Renesse. Configuring distributed computations using response surfaces. Proceedings of the 16th annual Middleware conference, 2015, pp. 235–246.
https://doi.org/10.1145/2814576.2814730

[30] Apache Mesos,
http://mesos.apache.org/

[31] Kubernetes (K8s),
https://kubernetes.io/

[32] Spark Configuration,
https://spark.apache.org/docs/latest/configuration.html

[33] Processor Counter Monitor (PCM),
https://github.com/opcm/pcm

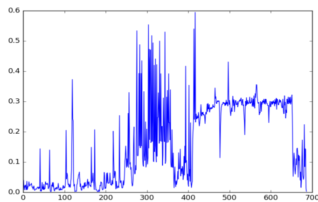
[34] Romain Tavenard, Johann Faouzi, Gilles Vandewiele, Felix Divo, Guillaume Androz, Chester Holtz, Marie Payne, Roman Yurchak, Marc Rußwurm, Kushal Kolar, Eli Woods. Tslearn, A Machine Learning Toolkit for Time Series Data. Journal of Machine Learning Research. 21(118): 1−6, 2020.
http://jmlr.org/papers/v21/20-091.html

[35] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 12(85):2825−2830, 2011.
http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

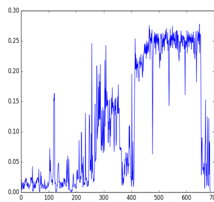
Appendices

Appendix A

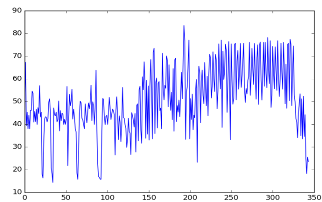
Low level metric plots to characterize benchmarks



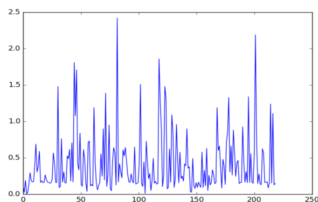
(1) Read Bayes



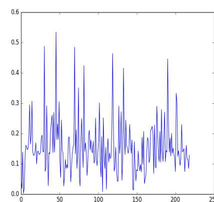
(2) Write Bayes



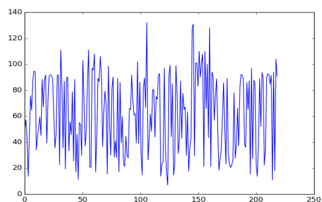
(3) IPC Bayes



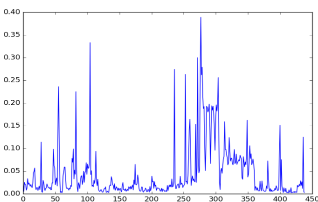
(4) Read Kmeans



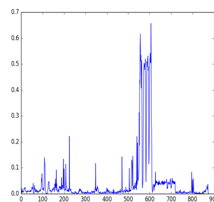
(5) Write Kmeans



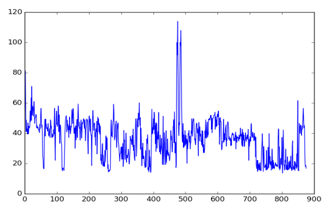
(6) IPC Kmeans



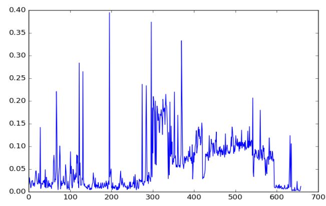
(7) Read Aggr

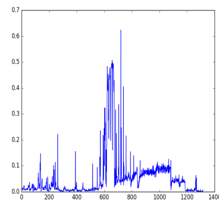


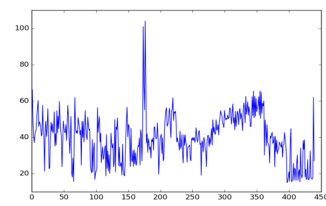
(8) Write Aggr

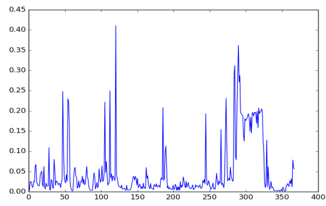


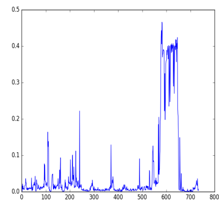
(9) IPC Aggr

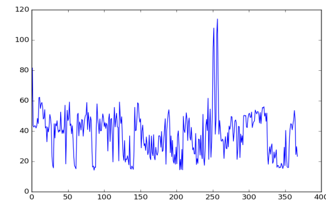

(10) Read Join

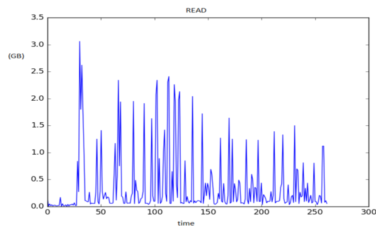

(11) Write Join


(12) IPC Join

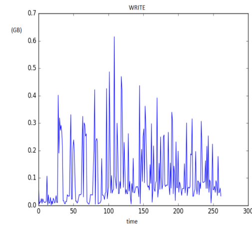

(13) Read Scan


(14) Write Scan

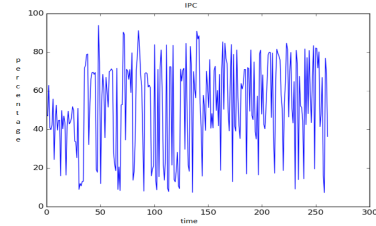

(15) IPC Scan



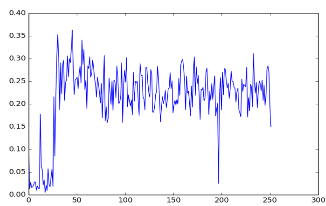
(16) Read PR

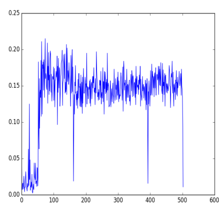


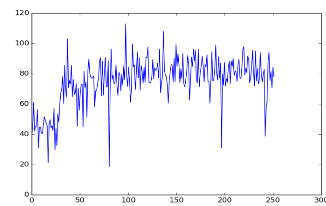
(17) Write PR

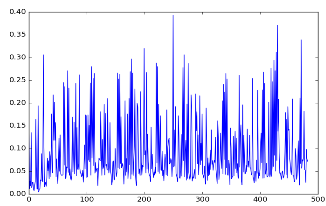


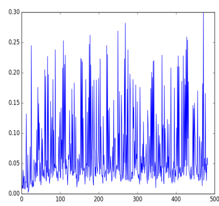
(18) IPC PR

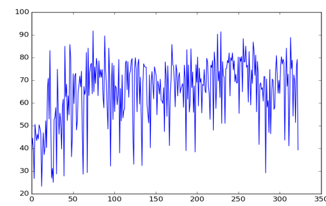

(19) Read Linear

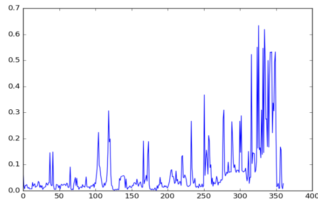

(20) Write Linear

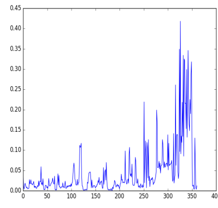

(21) IPC Linear

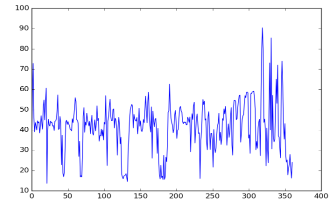

(22) Read GBT

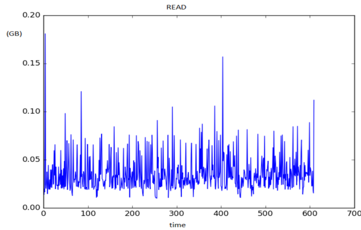

(23) Write GBT


(24) IPC GBT

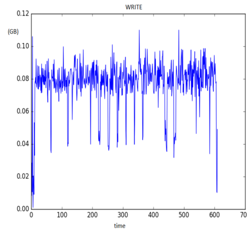

(25) Read Sort


(26) Write Sort

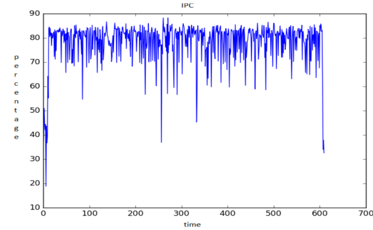

(27) IPC Sort



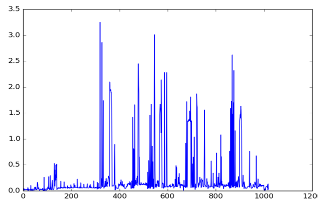
(28) Read LDA

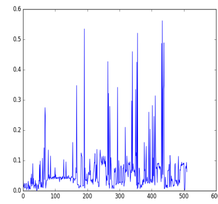


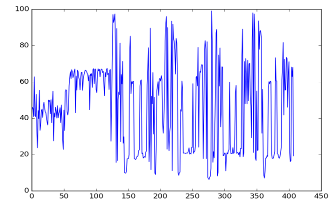
(29) Write LDA

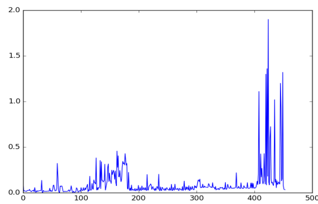


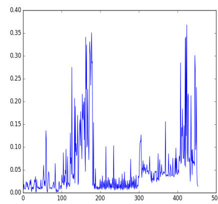
(30) IPC LDA

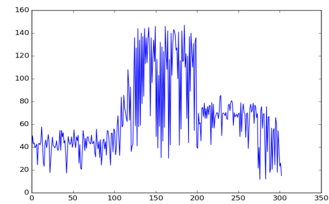

(31) Read TS

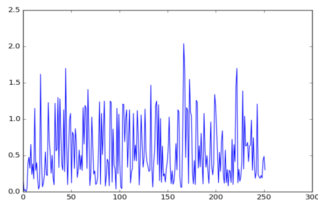

(32) Write TS

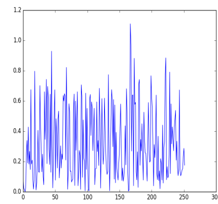

(33) IPC TS

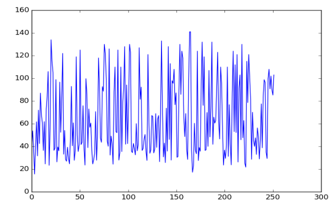

(34) Read ALS

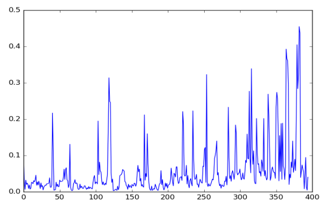

(35) Write ALS


(36) IPC ALS

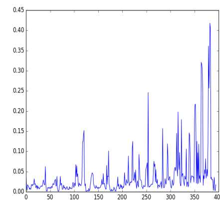

(37) Read LR


(38) Write LR

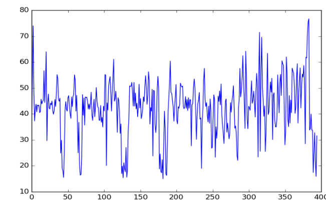

(39) IPC LR



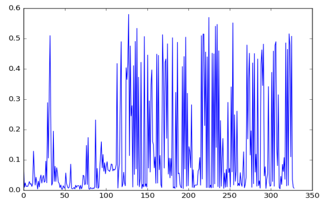
(40) Read RF



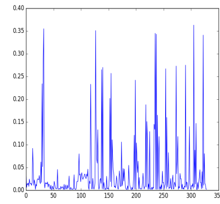
(41) Write RF



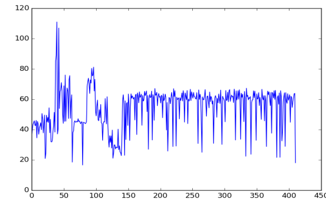
(42) IPC RF



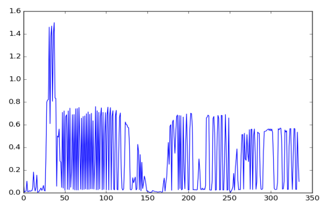
(43) Read PCA



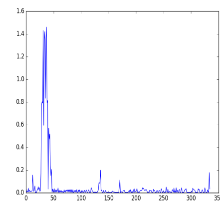
(44) Write PCA



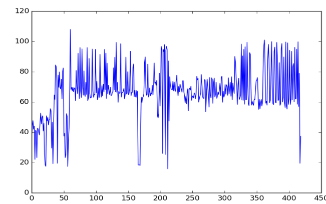
(45) IPC PCA



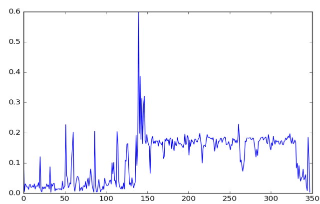
(46) Read SVD



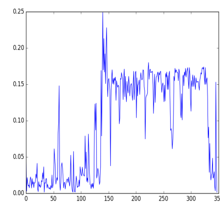
(47) Write SVD



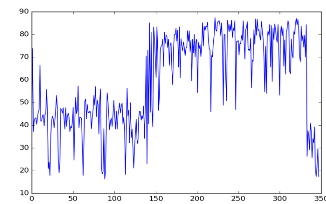
(48) IPC SVD



(49) Read WC



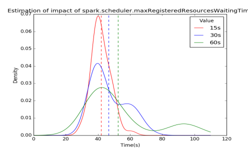
(50) Write WC



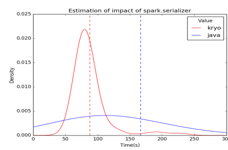
(51) IPC WC

Appendix B

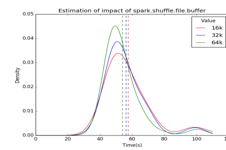
Density plots showing the significance of all spark parameters



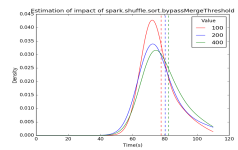
(1) spark.scheduler.maxRegisteredResourcesWaitingTime



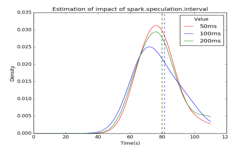
(2) spark.serializer



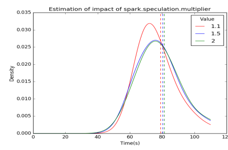
(3) spark.shuffle.file.buffer



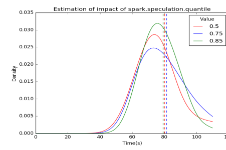
(4) spark.shuffle.sort.bypassMergeThreshold



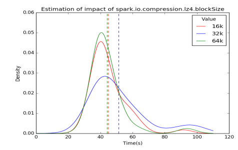
(5) spark.speculation.interval



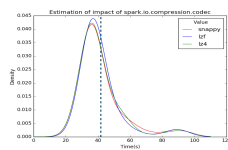
(6) spark.speculation.multiplier



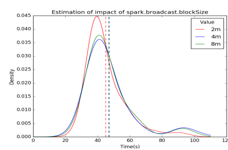
(7) spark.speculation.quantile



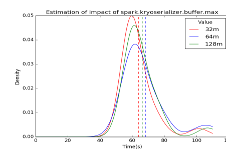
(8) spark.io.compression.lz4.blockSize



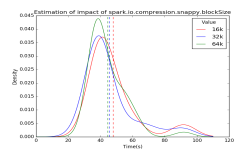
(9) spark.io.compression.codec



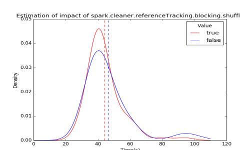
(10) spark.broadcast.blockSize



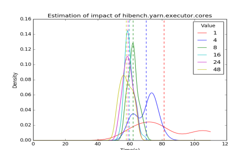
(11) spark.kryoserializer.buffer.max



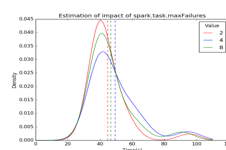
(12) spark.io.compression.snappy.blockSize



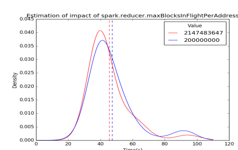
(13) spark.cleaner.referenceTracking.blocking.shuffle



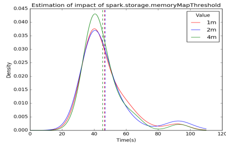
(14) spark.executor.cores

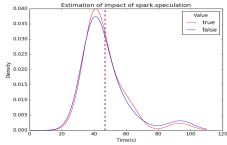


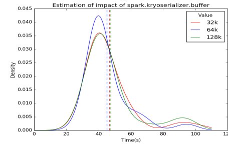
(15) spark.task.maxFailures

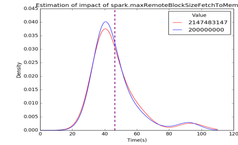


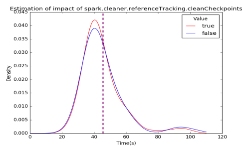
(16) spark.reducer.maxBlocksInFlightPerAddress

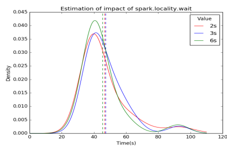

(17) spark.storage.memoryMapThreshold

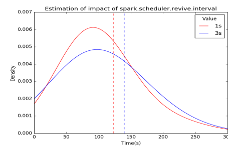

(18) spark.speculation

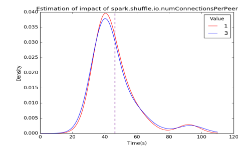

(19) spark.kryoserializer.buffer

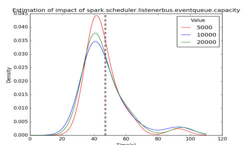

(20) spark.maxRemoteBlockSizeFetchToMem

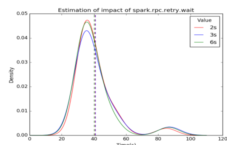

(21) spark.cleaner.referenceTracking.cleanCheckpoints

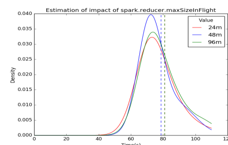

(22) spark.locality.wait

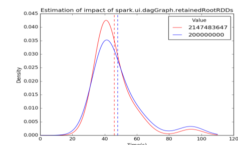

(23) spark.scheduler.revive.interval

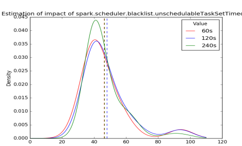

(24) spark.shuffle.io.numConnectionsPerPeer

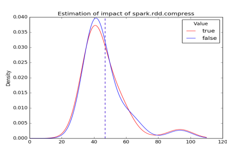

(25) spark.scheduler.listenerbus.eventqueue.capacity

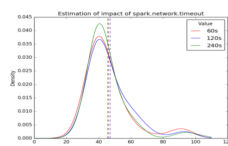

(26) spark.rpc.retry.wait

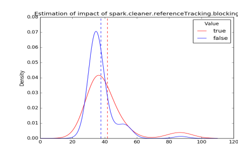

(27) spark.reducer.maxSizeInFlight

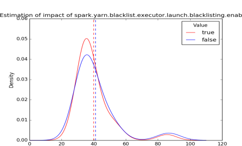

(28) spark.ui.dagGraph.retainedRootRDDs

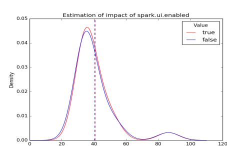

(29) spark.scheduler.blacklist.unschedulableTaskSetTimeout

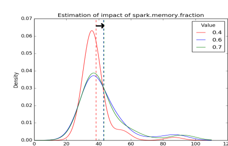

(30) spark.rdd.compress

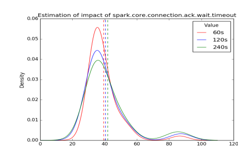

(31) spark.network.timeout

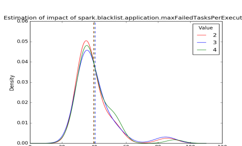

(32) spark.cleaner.referenceTracking.blocking

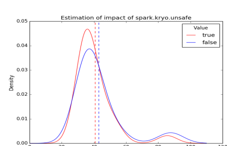

(33) spark.yarn.blacklist.executor.launch.blacklisting.enabled

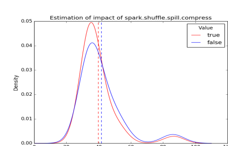

(34) spark.ui.enabled

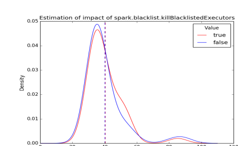

(35) spark.memory.fraction

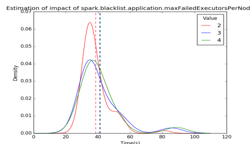

(36) spark.core.connection.ack.wait.timeout

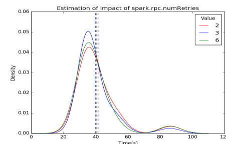

(37) spark.blacklist.application.maxFailedTasksPerExecutor

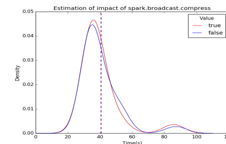

(38) spark.kryo.unsafe

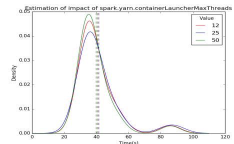

(39) spark.shuffle.spill.compress

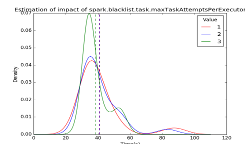

(40) spark.blacklist.killBlacklistedExecutors

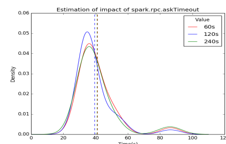

(41) spark.blacklist.application.maxFailedExecutorsPerNode

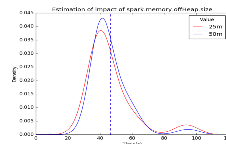

(42) spark.rpc.numRetries

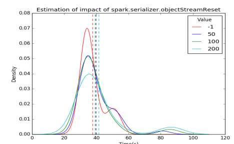

(43) spark.broadcast.compress

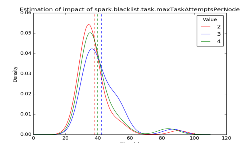

(44) spark.yarn.containerLauncherMaxThreads

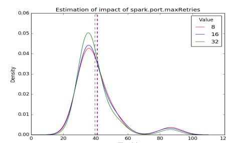

(45) spark.blacklist.task.maxTaskAttemptsPerExecutor

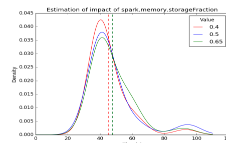

(46) spark.rpc.askTimeout

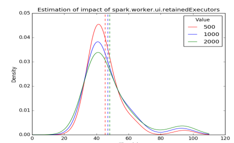

(47) spark.memory.offHeap.size

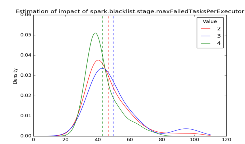

(48) spark.serializer.objectStreamReset

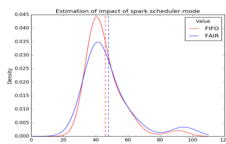

(49) spark.blacklist.task.maxTaskAttemptsPerNode

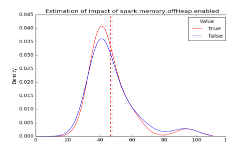

(50) spark.port.maxRetries

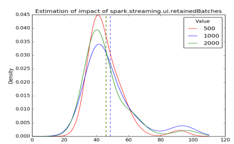

(51) spark.memory.storageFraction

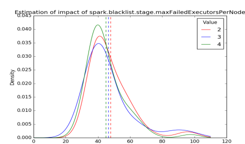

(52) spark.worker.ui.retainedExecutors

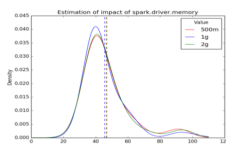

(53) spark.blacklist.stage.maxFailedTasksPerExecutor

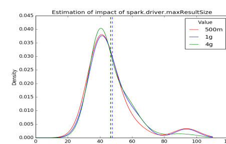

(54) spark.scheduler.mode

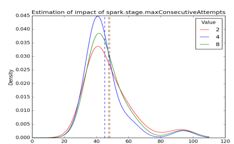

(55) spark.memory.offHeap.enabled

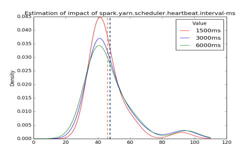

(56) spark.streaming.ui.retainedBatches

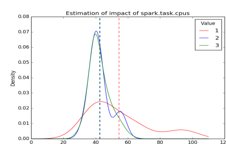

(57) spark.blacklist.stage.maxFailedExecutorsPerNode

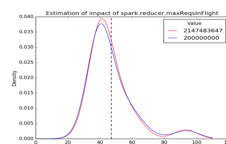

(58) spark.driver.memory

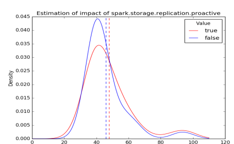

(59) spark.driver.maxResultSize

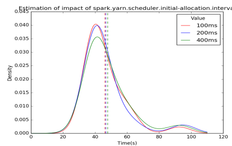

(60) spark.stage.maxConsecutiveAttempts


(61) spark.yarn.scheduler.heartbeat.interval-ms

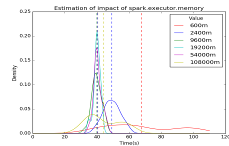

(62) spark.task.cpus


(63) spark.reducer.maxReqsInFlight

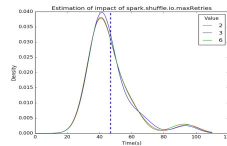

(64) spark.storage.replication.proactive



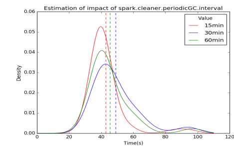
(65) spark.yarn.scheduler.initial-allocation.interval



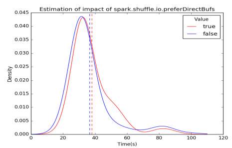
(66) spark.executor.memory



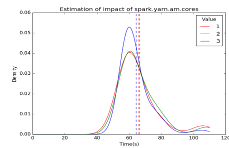
(67) spark.shuffle.io.maxRetries



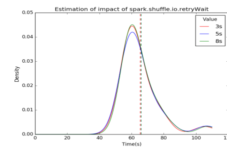
(68) spark.cleaner.periodicGC.interval



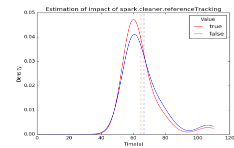
(69) spark.shuffle.io.preferDirectBufs



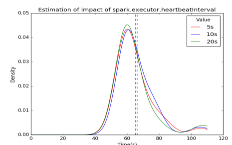
(70) spark.yarn.am.cores



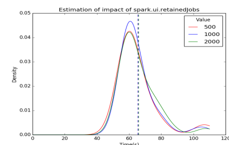
(71) spark.shuffle.io.retryWait



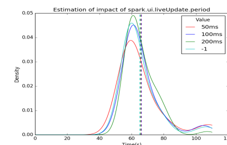
(72) spark.cleaner.referenceTracking



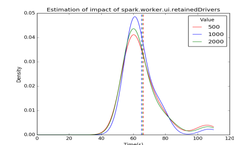
(73) spark.executor.heartbeatInterval



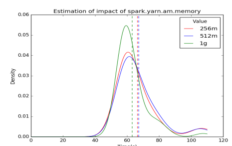
(74) spark.ui.retainedJobs



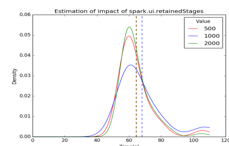
(75) spark.ui.liveUpdate.period



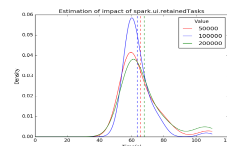
(76) spark.worker.ui.retainedDrivers



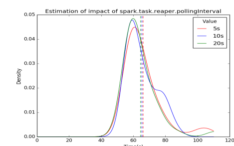
(77) spark.yarn.am.memory



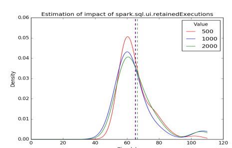
(78) spark.ui.retainedStages



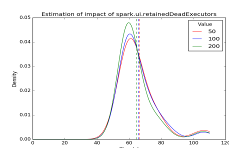
(79) spark.ui.retainedTasks



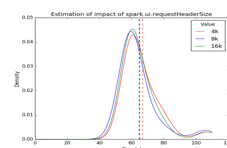
(80) spark.task.reaper.pollingInterval



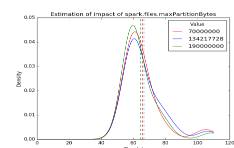
(81) spark.sql.ui.retainedExecutions



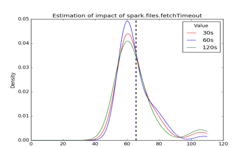
(82) spark.ui.retainedDeadExecutors



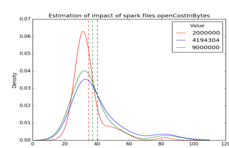
(83) spark.ui.requestHeaderSize



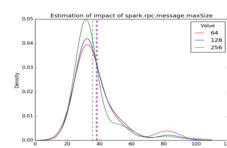
(84) spark.files.maxPartitionBytes



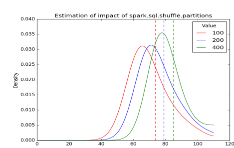
(85) spark.files.fetchTimeout



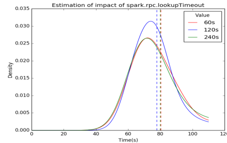
(86) spark.files.openCostInBytes



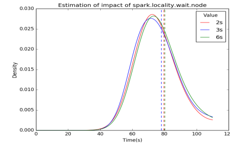
(87) spark.rpc.message.maxSize



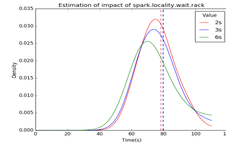
(88) spark.sql.shuffle.partitions



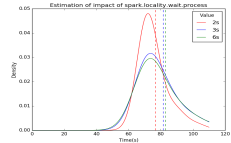
(89)
spark.rpc.lookupTimeout



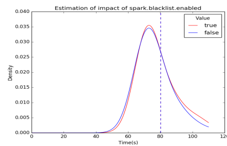
(90)
spark.locality.wait.node



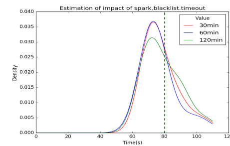
(91)
spark.locality.wait.rack



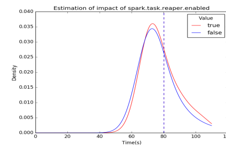
(92)
spark.locality.wait.process



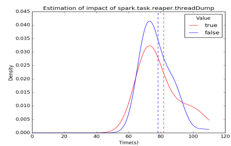
(93)
spark.blacklist.enabled



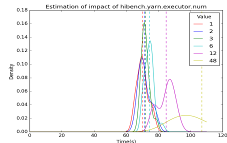
(94)
spark.blacklist.timeout



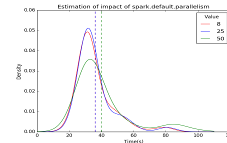
(95)
spark.task.reaper.enabled



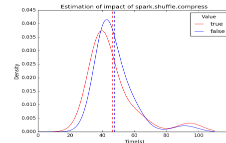
(96)
spark.task.reaper.threadDump



(97)
spark.executor.instances



(98)
spark.default.parallelism



(99)
spark.shuffle.compress