ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών

Εκτέλεση και Βελτιστοποίηση ροών εργασιών Big Data σε περιβάλλοντα πολλαπλών μηχανών

Διπλωματική Εργασία

Ιωάννης Γ. Μηλιός

Επιβλέπων Καθηγητής

Νεκτάριος Κοζύρης

Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών

Εκτέλεση και Βελτιστοποίηση ροών εργασιών Big Data σε περιβάλλοντα πολλαπλών μηχανών

Διπλωματική Εργασία

Ιωάννης Γ. Μηλιός

Επιβλέπων Καθηγητής

Νεκτάριος Κοζύρης

Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $10^{η}$ Νοεμβρίου 2017

………………..	………………..	………………..
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Δημήτριος Τσουμάκος
Καθηγητής Ε.Μ.Π.	Επίκ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ιονίου Παν

Αθήνα, Νοέμβριος 2017

Περίληψη

Τα τελευταία χρόνια παρατηρείται μια έκρηξη δεδομένων στο διαδίκτυο. Η ποικιλία των κοινωνικών δικτύων και η ευκολία με την οποία παρέχεται η πρόσβαση σε αυτά, οδηγεί τους χρήστες στη μεταμόρφωση τεράστιου όγκου δεδομένων σε ημερήσια βάση. Σε αυτό συμβάλλει και η πτώση της τιμής του υλικού, με αποτέλεσμα μοναδικοί χρήστες να χρησιμοποιούν πολλαπλές συσκευές για την παραγωγή εικόνων, κειμένων, video κ.τ.λ.

Η πληροφορία που υπάρχει σε αυτά τα δεδομένα και η ανάγκη που προκύπτει για την εξόρυξή της οδήγησε, από τις αρχές της προηγούμενης δεκαετίας, στη δημιουργία κατανεμημένων συστημάτων όπου με ειδικούς αλγορίθμους επεξεργάζονται τα δεδομένα και ανακτούν την απαραίτητη πληροφορία. Η διαφορά στον τρόπο αποθήκευσης και οι ποικίλοι τρόποι επεξεργασίας όμως έχουν οδηγήσει στην δημιουργία πολλαπλών τέτοιων μηχανών. Λόγω αυτής της ποικιλίας και κυρίως της διαφορετικότητας τους, οι χρήστες φαίνεται να είναι εγκλωβισμένοι στη χρησιμοποίηση μόνο λίγων εξ' αυτών με αποτέλεσμα να βάλλεται τόσο η επεκτασιμότητα όσο και η λειτουργία των εφαρμογών τους.

Τη λύση στο παραπάνω πρόβλημα έρχονται να δώσουν συστήματα τα οποία θα αδιαφορούν για την πλατφόρμα εκτέλεσης των ροών εργασιών, θα αντιμετωπίζουν τους τελεστές των εφαρμογών σαν μαύρα-κουτιά και θα αναθέτουν τις προς εκτέλεση εργασίες στα αντίστοιχα βέλτιστα συστήματα προς εκτέλεση.

Στην παρούσα διπλωματική εργασία παρουσιάζουμε την δημιουργία και εκτέλεση ροών εργασιών (workflows) σε δεδομένα μεγάλου όγκου, σε δύο διαφορετικά συστήματα, το IReS και το Rheem. Σκοπός μας είναι η σύγκρισή τους όσον αφορά στην ευκολία δημιουργίας των workflows , στην βελτιστοποίηση του πλάνου εκτέλεσης και στην επιλογή των ελάχιστων υπολογιστικών πόρων για τη μείωση του κόστους.

Για την επιλογή των εκάστοτε συστημάτων επεξεργασίας το IReS και το Rheem χρησιμοποιούν μοντέλα, που έχουν δημιουργηθεί στη φάση της εκπαίδευσης. Η δημιουργία αυτών των μοντέλων μέσω αλγορίθμων μηχανικής μάθησης, βασίζεται σε δεδομένα όπως ο χρόνος εκτέλεσης, ο αριθμός των πυρήνων που χρησιμοποιήθηκαν, το μέγεθος της κύριας μνήμης κ.τ.λ.

Μελετώντας διαφορετικά σενάρια χρήσης, οδηγούμαστε τέλος σε συμπεράσματα που αφορούν κυρίως τη διαφορετική προσέγγιση των συστημάτων απέναντι στους τελεστές των ροών εργασιών, την ύπαρξη ή μη βέλτιστης πολιτικής χρήσης των διαθέσιμων υπολογιστικών πόρων και την διαδικασία δημιουργίας και εκτέλεσης του βέλτιστου πλάνου.

Λέξεις Κλειδιά

IReS, Rheem, Hadoop, Spark, ροή εργασιών, τελεστής, περιβάλλον πολλαπλών μηχανών

Abstract

In recent years a data outburst through the internet is observed. The variety of the social media available and the ease with which access is provided, leads towards creation of big data on daily basis. A big contribution to this is the price drop of hardware and as a result a single user has access to multiple devices regarding image, text and video production among others.

Information available in this kind of data and the necessity of mining that stems from it, has lead since the early 00s in the creation of distributed systems which employ specific algorithms in order to process the data and retrieve the information required. However, the difference in data storage and the various ways of processing, have introduced multiple engines capable of such operations. The culmination of such diversity in these systems is that the users appear to be tied to limited number of them which results that both the extensibility and the usability of the applications are compromised.

In order to overcome the obstacles stated above, certain systems have been developed which are not bound to a specific execution platform of the workflow. These systems treat the application operators as black-boxes and assign the tasks in the equivalent optimized systems to be executed.

In this diploma thesis, the creation and execution of big data workflows in IReS and Rheem systems is presented. The primary objective is their comparison regarding the ease of creating such workflows, the optimization of the execution plan and the allocation of the minimum resources needed so as to reduce the cost.

In order to select the optimal execution plan, IReS and Rheem use models that have previously trained. With the assistance of machine learning algorithms, the generated models are based on meta data such as execution time, CPU load, size of main memory etc.

Finally, through studying various usage scenarios arise three main conclusions. First of all, both IReS and Rheem approach the workflows differently as far as the operators are concerned. Furthermore, IReS supports an optimal policy on allocation of resources available in contrast with Rheem. Lastly, a comparison of the two systems is displayed regarding the implementation and execution of the optimal plan.

Key-Words

IReS, Rheem, Hadoop, Spark, workflow, operator, multi-engine platform

Ευχαριστίες

Για την πραγματοποίηση της παρούσας διπλωματικής, θα ήθελα να ευχαριστήσω την οικογένειά μου, τους φίλους μου και την Ελπίδα, που ήταν πάντα δίπλα μου σε όλο αυτό το ταξίδι και με υποστήριζαν…

Περιεχόμενα

Πίνακας Εικόνων

1 Εισαγωγή

Με την πάροδο των χρόνων, η τεχνολογία εξελίχθηκε σε τέτοιο βαθμό, ώστε το διαδίκτυο να καταστεί πλέον το πιο διαδεδομένο μέσο επικοινωνίας και πληροφόρησης. Στις μέρες μας, ο κάθε χρήστης έχει τη δυνατότητα αλλά και την ευελιξία να χειρίζεται μέσω πολλαπλών συσκευών το ίντερνετ, παράγοντας ταυτόχρονα τεράστιο όγκο δεδομένων.

Τα μέσα κοινωνικής δικτύωσης αποτελούν την κύρια πηγή ενασχόλησης, ψυχαγωγίας, επικοινωνίας και πληροφόρησης του μέσου χρήστη. Πλέον όμως εκτός από την ανάκτηση της πληροφορίας οι χρήστες αλληλεπιδρούν και μεταμορφώνουν χιλιάδες κείμενα, εικόνες και βίντεο κάθε δευτερόλεπτο. Σύμφωνα με τις επίσημες μετρήσεις τον Ιούνιο του 2017 χρησιμοποίησαν ενεργά το Facebook πάνω από δύο δισεκατομμύρια μοναδικοί χρήστες , με τα νούμερα να κυμαίνονται στην τάξη των εκατοντάδων εκατομμυρίων χρηστών μηνιαίως και για τα υπόλοιπα κοινωνικά δίκτυα όπως το twitter, το Linkedin και το Instagram.

Σύμφωνα με μελέτες το 90% των δεδομένων που υπάρχουν σήμερα στο διαδίκτυο δημιουργήθηκαν τα τελευταία 2 χρόνια, καθώς επίσης καθημερινώς παράγονται 2.5 πεντάκις εκατομμύρια bytes δεδομένων (2.5×10^{18}) [1]. Στην ραγδαία αυτή αύξηση των δεδομένων προφανώς έχει συμβάλλει και το hardware. Η πτώση της τιμής του υλικού δίνει τη δυνατότητα στον κάθε χρήστη ξεχωριστά να διαχειρίζεται πολλαπλές συσκευές και να ανακτά αλλά και να παράγει συνεχώς πληροφορία.

Όπως εύκολα καταλαβαίνει κανείς η εξόρυξη της πληροφορίας που υπάρχει μέσα σε αυτόν τον τεράστιο όγκο δεδομένων, είναι πολλή σημαντική, αλλά ταυτόχρονα απαιτεί αρκετό χρόνο και είναι κοστοβόρα.

Από τις αρχές της προηγούμενης δεκαετίας άρχισε να παρατηρείται η δυσκολία στο να αποθηκεύονται στοιχεία σε μεμονωμένες βάσεις δεδομένων και η ανάγκη για τη δημιουργία κατανεμημένων βάσεων και αλγορίθμων για την επεξεργασία τους. Συγκεκριμένα η google στα τέλη του 2004 βλέποντας αυτή τη συμφόρηση παρουσίασε τον αλγόριθμο του map-reduce όπου 'σπάει' τα δεδομένα σε υποκατηγορίες , τα αναλύει με ξεχωριστούς υπολογιστικούς πόρους και μετά ενώνει τα αποτελέσματα ώστε να οδηγηθεί στην εξόρυξη της πληροφορίας. Αυτός ο αλγόριθμος οδήγησε στην δημιουργία του Hadoop [2] , ενός open-source project που χρησιμοποιείται για την κατανεμημένη αποθήκευση και επεξεργασία δεδομένων μεγάλου όγκου.

Από τη γέννηση του Hadoop μέχρι σήμερα, τα εργαλεία ανάλυσης δεδομένων μεγάλου όγκου έχουν γίνει αναπόσπαστο κομμάτι των επιχειρήσεων παγκοσμίως. Μεγάλες εταιρίες όπως η google, Facebook, amazon κ.τ.λ. για να εντοπίσουν το τι πραγματικά χρειάζονται οι πελάτες τους και να τους κατηγοριοποιήσουν ανάλογα με τις προτιμήσεις και τα ενδιαφέροντα τους , χρησιμοποιούν τα παραπάνω εργαλεία.

Φυσικά η ανάγκη για ανάλυση τέτοιου όγκου δεδομένων δεν περιορίζεται μόνο στα κοινωνικά δίκτυα αλλά εμφανίζεται επίσης σε διάφορους τομείς. Για παράδειγμα μια εταιρεία εξόρυξης πετρελαίου μπορεί να παράγει περισσότερο από 1.5 TB διαφορετικά δεδομένα ημερησίως. Τα δεδομένα αυτά μπορεί να προέρχονται από ετερογενείς πηγές, όπως αισθητήρες, συσκευές GPS και άλλες συσκευές μέτρησης.

Επιπροσθέτως ο κατανεμημένος τρόπος επεξεργασίας των δεδομένων έχει οδηγήσει στην εκτόξευση της χρήσης αλγορίθμων μηχανικής μάθησης. Παλιότερα για την ανάλυση μεγάλου όγκου δεδομένων ο ερευνητής περιοριζόταν στην μνήμη του υπολογιστή του με αποτέλεσμα την δημιουργία μοντέλων χρησιμοποιώντας ένα κομμάτι των πραγματικών δεδομένων. Πλέον του παρέχεται η δυνατότητα να χρησιμοποιεί το σύνολο των δεδομένων έτσι ώστε να παράγει καλύτερα και πιο ακριβή μοντέλα.

Τα τελευταία χρόνια έχουν δημιουργηθεί πολλαπλά εργαλεία για την ανάλυση δεδομένων σε κατανεμημένα περιβάλλοντα. Το πιο διαδεδομένο από αυτά είναι το Spark [3]. Το Spark δημιουργήθηκε το 2012 και χρησιμοποιείται κατά κόρον από το 2014 και μετά. Σε αντίθεση με to Hadoop δεν προσφέρει το δικό του σύστημα διαχείρισης αρχείων, οπότε είναι απαραίτητο τα προς επεξεργασία αρχεία να βρίσκονται είτε στο HDFS(Hadoop Distributed File System) [4] είτε σε κάποιο άλλο cloud-based σύστημα. Η διαφορά του με το Hadoop εντοπίζεται κυρίως στην επεξεργασία, καθώς το Spark εκτελεί τις εφαρμογές 100 φορές γρηγορότερα στη μνήμη και 10 φορές γρηγορότερα στο δίσκο από το Hadoop. Αυτό το πετυχαίνει μειώνοντας το διάβασμα-γράψιμο στο δίσκο και αποθηκεύοντας τα ενδιάμεσα αποτελέσματα στη μνήμη, σε αντίθεση με το Hadoop όπου μετά από κάθε μεμονωμένη ενέργεια αποθηκεύει τα αποτελέσματα στο cluster και μετά ξανά διαβάζει από εκεί. Το Spark επίσης είναι το πλέον κατάλληλο για εφαρμογές που τρέχουν σε πραγματικό χρόνο καθώς επίσης και για αλγορίθμους μηχανικής μάθησης οι οποίοι εκτελούν πολλαπλές λειτουργίες.

Βλέπουμε λοιπόν πως ενώ αρχικά φαίνεται να λύθηκε το πρόβλημα της επεξεργασίας των δεδομένων μεγάλου όγκου, σιγά σιγά έχει αρχίσει να αναδύεται στην επιφάνεια ένα καινούριο. Ενώ οι τωρινές πλατφόρμες ανάλυσης είναι πετυχημένες στην αξιοποίηση πολλαπλών πτυχών των δεδομένων, ταυτόχρονα φαίνεται η αποτελεσματικότητά τους να είναι δεσμευμένη από ένα μόνο υπολογιστικό μοντέλο, το οποίο συνήθως εξαρτάται από κάποιο ιδιόκτητο σύστημα. Φυσικά κανένα σύστημα δεν είναι κατάλληλο για όλων των τύπων υπολογισμούς και κανένα σύστημα αποθήκευσης δεν είναι κατάλληλο για όλα τα είδη των δεδομένων. Αυτό πρακτικά σημαίνει πως για παράδειγμα αν ένας οργανισμός θέλει να μεταπηδήσει από το Hadoop στο Spark θα πρέπει να υλοποιήσει εξαρχής όλες τις εφαρμογές κατάλληλα ώστε να υπακούν στο νέο σύστημα επεξεργασίας. Το γεγονός αυτό αποθαρρύνει την επέκταση των εφαρμογών και ταυτόχρονα αναγκάζει τους αναλυτές να είναι ‘δεμένοι’ με ένα σύστημα επεξεργασίας και αποθήκευσης, με σκοπό να αποφεύγουν τις δυσκολίες της μετάβασης από το ένα σύστημα στο άλλο.

Για το λόγο αυτόν έχουν παρουσιαστεί τρία συστήματα διαχείρισης πολλαπλών μηχανών (multi-engine platforms), το IReS [5], το Rheem [6] και το Musketeer [7] . Σκοπός των παραπάνω συστημάτων είναι η ανεξαρτησία τους από τις πλατφόρμες επεξεργασίας δεδομένων μεγάλου όγκου της αγοράς. Για να επιτύχουν αυτό το σκοπό χρησιμοποιούν έναν μετά-χρονοδρομολογητή που έχει την ευθύνη της κατανομής κάθε υπό-εργασίας στο αντίστοιχο προς εκτέλεση σύστημα.

Σκοπός της παρούσας διπλωματικής εργασίας είναι η πολυεπίπεδη σύγκριση του IReS και του Rheem κατά την διαδικασία της ανάλυσης δεδομένων μεγάλου όγκου.

Στο 2ο κεφάλαιο θα αναλύσουμε το θεωρητικό υπόβαθρο των δύο συστημάτων , δίνοντας βάση στην αρχιτεκτονική και στον τρόπο σχεδίασης και λειτουργίας τους.

Στο 3ο κεφάλαιο θα παρουσιάσουμε στοχευμένα workflows για την ανάδειξη των πλεονεκτημάτων και των μειονεκτημάτων της κάθε προσέγγισης

Στο 4ο κεφάλαιο θα ασχοληθούμε με την αξιολόγηση και τα τελικά συμπεράσματα που προέκυψαν από τις δοκιμές στις οποίες τα υποβάλλαμε.

2 Θεωρητικό Υπόβαθρο IReS – Rheem

2.1 IReS

Στις μέρες μας, η ροή εργασιών(workflow) που ακολουθείται για την ανάλυση δεδομένων σε διάφορα προβλήματα, χαρακτηρίζεται από μεγάλη πολυπλοκότητα αλλά και από την ύπαρξη πολλών διαφορετικών λειτουργιών που πρέπει να υλοποιηθούν για την επίτευξη της. Ειδικότερα τέτοιου είδους εργασίες μπορεί να αποτελούνται από πολλαπλούς τύπους δεδομένων (π.χ. σχεσιακοί, γράφοι) οι οποίοι να παράγονται από διαφορετικές πηγές. Επιπροσθέτως δύναται να εκτελούνται κάτω από διαφορετικές παραμέτρους, όπως έμφαση στη μείωση του κόστους σε μνήμη, κατανάλωσης υπολογιστικής ισχύος αλλά και αδιαφορία για τα παραπάνω με σκοπό την βελτιστοποίηση της απόδοσης όσον αφορά το χρόνο εκτέλεσης. Κατόπιν διαφέρουν ακόμα και στην απλότητά της φύσης τους όπως είναι τα απλά joins και η μεταφορά δεδομένων, σε σχέση με το πολύπλοκο NLP[8] και τα περισσότερα προβλήματα επεξεργασίας γράφων. Όπως εύκολα καταλαβαίνει κανείς δεν υπάρχει ένα μοναδικό σύστημα που να μπορεί να εξυπηρετήσει τα παραπάνω και οι όποιες προσπάθειες έχουν γίνει για την επίτευξη αυτής της συγχώνευσης, στερούνται έναν μέτα-χρονοδρομολογητή (meta-scheduler) ο οποίος θα είναι υπεύθυνος για την ανάθεση της κάθε εργασίας-λειτουργίας στο κατάλληλο σύστημα.

Το IReS (Intelligent Multi-Engine Resource Scheduler) είναι ένα open-source σύστημα που μπορεί να διαχειρίζεται, να εκτελεί και να παρακολουθεί πολύπλοκες ροές εργασιών(workflows) όπως οι προαναφερθείσες. Προσεγγίζει τους εκάστοτε τελεστές ως μαύρα -κουτιά, γεγονός που διευκολύνει το χειρισμό οποιουδήποτε έργου που μπορεί να κυμαίνεται από απλές λειτουργίες (π.χ. ένωση, ταξινόμηση) αλλά και αρκετά πιο πολύπλοκες όπως αλγορίθμους μηχανικής μάθησης και επεξεργασία γράφων. Η εκτέλεσή τους μπορεί να γίνει σε οποιοδήποτε είτε συγκεντρωτικό είτε κατανεμημένο σύστημα. Λόγω του ότι αντιμετωπίζει αγνωστικιστικά τα διάφορα συστήματα καθιστά πολλή εύκολη την προσθήκη καινούριων τελεστών και συστημάτων.

Το μόνο που χρειάζεται το IReS είναι η περιγραφή του έργου προς ανάλυση αλλά και των δεδομένων μέσω μιας επεκτάσιμης δομής, καθώς επίσης και ένα μοντέλο με τα χαρακτηριστικά του κόστους και της επίδοσης του έργου στα διαθέσιμα συστήματα που θα τους ανατεθεί η εκτέλεση. Στη συνέχεια χρησιμοποιώντας τον σχεδιαστή(planner) του, που βασίζεται σε αλγορίθμους δυναμικού προγραμματισμού, αναθέτει τον κάθε τελεστή της ροής εργασιών στο αντίστοιχο βέλτιστο σύστημα που είναι διαθέσιμο και ταυτόχρονα επιλέγει την ακριβή ποσότητα πόρων που θα χρειαστούν για την εκτέλεση [9].

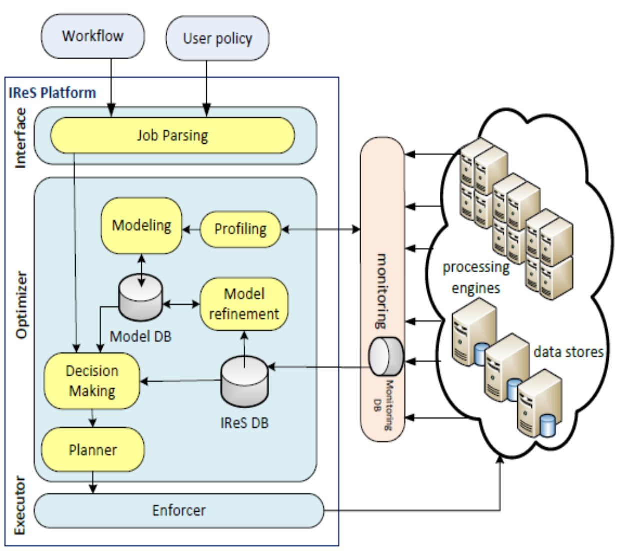
Φυσικά για να έχει τη δυνατότητα ο planner να επιλέγει το βέλτιστο τρόπο κατανομής των εργασιών είναι υλοποιημένο στο IReS ένα μοντέλο που συλλέγει πληροφορίες από πραγματικές εκτελέσεις των τελεστών. Η συλλογή αυτή λαμβάνει μέρος τόσο κατά τη διάρκεια της εκπαίδευσης του μοντέλου, όσο και κατά τη φάση της πραγματικής εκτέλεσης της ροής εργασιών.

Παρακάτω ακολουθεί η λεπτομερής ανάλυση της αρχιτεκτονικής του IReS αλλά και των υπό-επιπέδων που αποτελείται.

2.1.1 Αρχιτεκτονική

Το IReS επικεντρώνεται στην υψηλή αποτελεσματικότητα και στην προσαρμογή της εκτέλεσης των εργασιών ανάλυσης που παρέχεται από τον εκάστοτε χρήστη. Αυτό γίνεται δυνατόν μέσω της δημιουργίας συμπαγών μοντέλων, της επίβλεψης αλλά και της χρονοδρομολόγησης που αφορούν τα διαφορετικά συστήματα εκτέλεσης και αποθήκευσης των δεδομένων της ροής εργασιών. Παρέχει τη δυνατότητα τα διαφορετικά κομμάτια της εκτέλεσης να αναθέτονται σε πάνω από μια μηχανή ανάλογα με τις απαιτήσεις του χρήστη ως προς τις παραμέτρους της βελτιστοποίησης του έργου. Ακόμα και στην περίπτωση της επιλογής ενός μόνο συστήματος το IReS θα επιτύχει την βελτιστοποίηση της εργασίας αποφασίζοντας τον ακριβή αριθμό πόρων που θα χρειαστούν. Στην περίπτωση δε των πολλαπλών συστημάτων επιλογής θα ληφθεί υπόψιν τόσο η ζήτηση σε πόρους όσο και η επίτευξη της τελικής βελτιστοποίησης που επιθυμεί ο χρήστης.

Η κεντρική ιδέα πίσω από το IReS είναι η δημιουργία λεπτομερών μοντέλων με βάση το κόστος και την επίδοση, δεδομένα που λαμβάνονται κατά την εκτέλεση. Ύστερα αυτά τα μοντέλα χρησιμοποιούνται για την επιλογή του κατάλληλου γράφου εκτέλεσης με κοινό γνώμονα πάντοτε την ικανοποίηση των προδιαγραφών που θέτει ο χρήστης πριν την εκτέλεση. Η αρχιτεκτονική του IReS φαίνεται στο Σχήμα 1. Αποτελείται από τρία επίπεδα : το κομμάτι της διεπαφής (interface) , το κομμάτι του βελτιστοποιητή (optimizer) και το κομμάτι της εκτέλεσης (execution).



Εικόνα 1 : Η Αρχιτεκτονική του IReS

2.1.2 Τα διαφορετικά επίπεδα του IReS

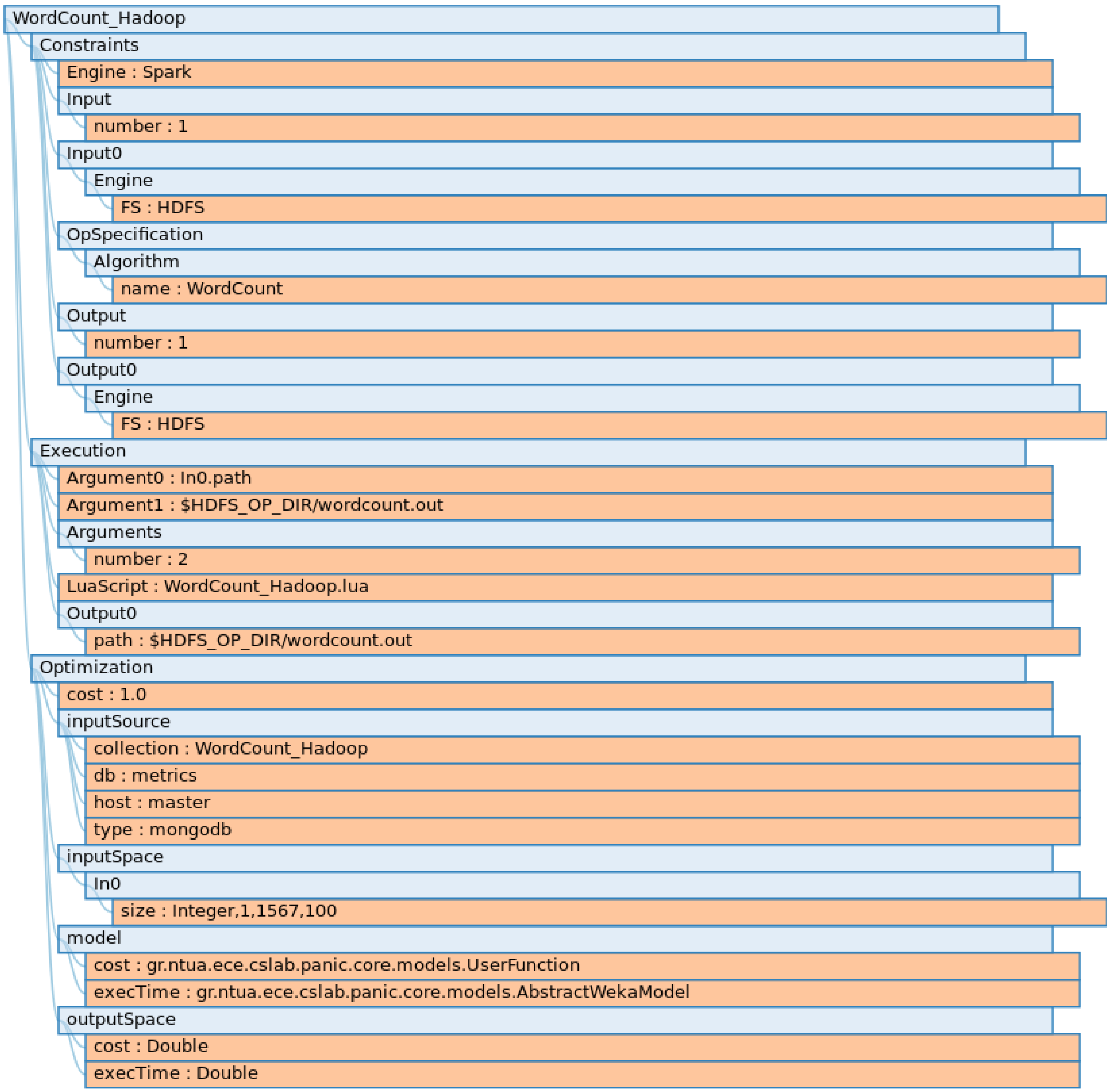
2.1.2.1 Interface Layer

Το επίπεδο της διεπαφής (interface) είναι ουσιαστικά υπεύθυνο για την επικοινωνία με το περιβάλλον του χρήστη με σκοπό να λάβει την είσοδο που απαιτείται για να εκτελεστούν οι λειτουργίες. Αποτελείται από τον αναλυτή της εργασίας (job parser) ο οποίος αναγνωρίζει τα προς εκτέλεση αντικείμενα όπως οι τελεστές, τα δεδομένα τις εξαρτήσεις τους αλλά και τα μετά-δεδομένα από τα οποία συνοδεύονται. Επιπρόσθετα επικυρώνει την πολιτική του χρήστη, η οποία πρέπει να είναι αυστηρά ορισμένη και δομημένη σε έναν ειδικό γράφο εξαρτήσεων.

Ο χρήστης καλείται να δηλώσει τα αντικείμενα της εκτέλεσης όπως οι τελεστές , τα δεδομένα και ολόκληρη την ροή εργασιών που θα ακολουθηθεί, μαζί με τις εσωτερικές εξαρτήσεις τους και τους περιορισμούς τους χρησιμοποιώντας μια κοινή δομή περιγραφής μετά-δεδομένων. Μέσω αυτής της αφηρημένης δομής παρέχεται η δυνατότητα στο χρήστη να επιλέξει τα συστήματα που επιθυμεί να εκτελέσει την εφαρμογή του, καθώς επίσης και όλους τους μεμονωμένους τελεστές που θα χρειαστεί όπως και τις παραμέτρους ως προς τις οποίες θα γίνει η βελτιστοποίηση. Κατόπιν το IReS είναι υπεύθυνο να αφαιρέσει τα αφηρημένα κομμάτια για να δρομολογηθεί το πλάνο εκτέλεσης.

Οι κύριες οντότητες της παραπάνω δομής είναι τα δεδομένα και οι τελεστές όπου θα πρέπει να συνοδεύονται από τα μετά-δεδομένα, τις ιδιότητες δηλαδή που τα περιγράφουν. Τα δεδομένα και οι τελεστές μπορούν να είναι είτε αφηρημένα (abstract) είτε υλοποιημένα (materialized) . Οι αφηρημένοι τελεστές και δεδομένα χρησιμοποιούνται για να περιγράψουν μια αφηρημένη ροή εργασιών (abstract workflow) προς εκτέλεση. Αντίθετα οι υλοποιημένοι τελεστές περιέχουν μέσα τους ειδικές πληροφορίες για τους ίδιους και θα πρέπει να συνοδεύονται από τα προγράμματα στα οποία αναφέρονται.

Πιο συγκεκριμένα στους υλοποιημένους τελεστές ο χρήστης έχει τη δυνατότητα να ορίσει διάφορες παραμέτρους , όπως ο αλγόριθμος εκτέλεσης, το σύστημα υποστήριξης, τον τύπο των δεδομένων εισόδου αλλά και την επιθυμητή βελτιστοποίηση ως προς κάποια παράμετρο που αφορά είτε τον αλγόριθμο είτε το σύστημα αποθήκευσης. Επίσης σε αυτό το σημείο έχει τη δυνατότητα να διαλέξει τη χρήση ή όχι των μοντέλων που έχουν δημιουργηθεί καθώς και να φτιάξει δικιά του συνάρτηση βελτιστοποίησης. Όλες οι παραπάνω πληροφορίες οργανώνονται και αποτυπώνονται σε δεντρική δομή, όπως φαίνεται στην Εικόνα 2. Για να υπάρχει ευελιξία μόνο τα πρώτα επίπεδα του δέντρου των μετά-δεδομένων είναι προκαθορισμένα. Είναι υποχρεωτικό να παρέχονται από το χρήστη ιδιότητες που αφορούν τους τελεστές και τα δεδομένα, ενώ είναι προαιρετικά αυτά που αφορούν τη βελτιστοποίηση και την επιλογή των μοντέλων. Στους αφηρημένους τελεστές αντίθετα δεν ισχύουν τα παραπάνω. Συγκεκριμένα ο χρήστης θα πρέπει να ορίσει τα πεδία που αφορούν τους περιορισμούς (constraints) και την εκτέλεση (execution) , ενώ τα πεδία της βελτιστοποίησης (optimization) είναι προαιρετικά.



Εικόνα 2 : Δεντρική αναπαράσταση του τελεστή WordCount_Hadoop

Στο υπό-δέντρο των περιορισμών περιέχονται όλες οι πληροφορίες που χρειάζονται έτσι ώστε οι αφηρημένοι τελεστές να αντιστοιχηθούν με τους υλοποιημένους αλλά και με τα σωστά δεδομένα. Τα υποχρεωτικά πεδία αφορούν διασαφήνιση του συστήματος από το οποίο θα προέλθουν τα δεδομένα αλλά και θα αποθηκευτούν, τον αλγόριθμο και τα συστήματα επεξεργασίας που θα χρησιμοποιηθούν και οτιδήποτε άλλο κρίνεται απαραίτητο για να επιτευχθεί η αντιστοίχιση των τελεστών.

Το υπό-δέντρο της εκτέλεσης παρέχει τις απαραίτητες παραμέτρους για την εκτέλεση του υλοποιημένου τελεστή. Εδώ δηλώνεται το πραγματικό μονοπάτι που αντιστοιχεί στο σύστημα που βρίσκονται τα δεδομένα (π.χ. HDFS) καθώς και τα arguments που απαιτούνται για να εκτελεστεί το script του τελεστή.

Όπως προαναφέρθηκε τα πεδία της βελτιστοποίησης (optimization) είναι προαιρετικά. Σε αυτά ο χρήστης μπορεί να επιλέξει αν χρειάζεται να χρησιμοποιήσει μοντέλα που έχουν δημιουργηθεί προηγουμένως αλλά και να ορίσει δικές του συναρτήσεις βελτιστοποίησης ως προς κάποια παράμετρο που έχει μετρηθεί κατά τη φάση της εκπαίδευσης. Στην Εικόνα 2 φαίνεται η επιλογή της βελτιστοποίησης ως προς το μέγεθος του αρχείου εισόδου.

2.1.2.2 Optimization Layer

Το επίπεδο του βελτιστοποιητή (optimizer) είναι υπεύθυνο για την βελτιστοποίηση εν τέλει της εκτέλεσης σεβόμενο τις απαιτήσεις που έχουν οριστεί από το χρήστη. Το κύριο συστατικό του βελτιστοποιητή είναι το κομμάτι της επιλογής (planner), όπου αποφασίζει το βέλτιστο πλάνο εκτέλεσης σε πραγματικό χρόνο. Αυτό ουσιαστικά σημαίνει πως όλες οι αποφάσεις για το που θα εκτελεστούν οι ξεχωριστοί τελεστές αλλά και το πόσοι πόροι θα χρειαστούν, που αναφέρθηκε παραπάνω, λαμβάνονται σε αυτό το στάδιο. Επίσης σε αυτό το σημείο αποφασίζεται το βέλτιστο πλάνο όσον αφορά την ενδιάμεση μεταφορά των δεδομένων-αποτελεσμάτων κάθε μεμονωμένης εργασίας ώστε να καταλήξει στον τελικό προορισμό της εξόδου. Οι αποφάσεις αυτές βασίζονται στα μοντέλα που έχουν δημιουργηθεί από προηγούμενες εκτελέσεις από τον profiler και αλληλεπιδρούν ανάμεσα στους φυσικούς διαθέσιμους πόρους και στο επίπεδο παρακολούθησης (monitoring) . Επιπροσθέτως κατά τη διάρκεια της εκτέλεσης της ροής εργασιών τα μοντέλα ουσιαστικά εκκαθαρίζονται, λόγω των καινούριων δεδομένων που λαμβάνονται, και είναι δυνατόν να επαναπροσδιοριστούν σε πραγματικό χρόνο έτσι ώστε να προσφέρουν το νέο βέλτιστο πλάνο εκτέλεσης. Υπεύθυνο για αυτή τη λειτουργία είναι το μοντέλο εκκαθάρισης (Refinement Model), το οποίο λαμβάνει και αποθηκεύει στη βάση δεδομένων του IReS στοιχεία που αφορούν το χρόνο εκτέλεσης, τη χρήση πυρήνων και μνήμης καθώς και άλλων μετρήσεων, τα οποία μπορεί να χρησιμοποιηθούν σε πραγματικό χρόνο δυναμικά έτσι ώστε να υπάρχει πλήρης ενημέρωση ως προς τη βελτιστοποίηση. Τα παραπάνω μοντέλα παράγονται από το ειδικό τμήμα του βελτιστοποιητή (Modeling Module) και αποθηκεύονται στην ειδική βάση δεδομένων (Model DB).

Πιο αναλυτικά, το εργαλείο που χρησιμοποιεί το IReS για τη δημιουργία του προφίλ των εργασιών (profiler) διαφέρει από κάποιο αντίστοιχο που μπορεί να υπάρχει σε μια σχεσιακή βάση δεδομένων, καθώς κάτι τέτοιο δεν μπορεί να χρησιμοποιηθεί σε πολύπλοκες εφαρμογές όπως σε τελεστές υλοποίησης αλγορίθμων μηχανικής μάθησης ή ανάλυσης γράφων. Γι' αυτό το λόγο αδιαφορεί πλήρως για την πλατφόρμα εκτέλεσης και συμπεριφέρεται στον υλοποιημένο τελεστή ως μαύρο-κουτί και δημιουργεί το μοντέλο βάσει μετρήσεων που συγκεντρώνει σε offline εκτελέσεις καθώς και χρησιμοποιώντας αλγορίθμους μηχανικής μάθησης σε εκτελέσεις πραγματικού χρόνου. Ο μηχανισμός αυτός έχει υιοθετήσει κομμάτια προηγούμενης δουλειάς [10]. Οι παράμετροι εισόδου που δέχεται ανήκουν σε τρεις κατηγορίες : α) πληροφορίες που αφορούν τα ίδια τα δεδομένα όπως ο τύπος τους ή το μέγεθός τους, β) πληροφορίες που έχουν να κάνουν με τους τελεστές και τους αλγορίθμους που τους υλοποιούν και τέλος γ) πληροφορίες που αφορούν τους πόρους που θα χρησιμοποιηθούν όπως για παράδειγμα το μέγεθος της συστοιχίας (cluster size) ή το μέγεθος της κύριας μνήμης. Η έξοδος κάθε εκτέλεσης αποτελεί το προφίλ του τελεστή όσον αφορά την επίδοσή του και το κόστος του, υπολογίζοντας όλους τους συνδυασμούς που αφορούν τις εισόδους που έχει επιλέξει ο χρήστης κατά τη διαδικασία της περιγραφής της ροής των εργασιών. Οι μετρήσεις που μαζεύονται εν τέλει χρησιμοποιούνται για να δημιουργήσουν μοντέλα πρόβλεψης, κάνοντας χρήση νευρωνικών δικτύων, SVM [11] , παρεμβολής και τεχνικών κυρτότητας για τον κάθε τελεστή που εκτελείται σε κάποιο συγκεκριμένο σύστημα.

Ένα ακόμη κομμάτι του βελτιστοποιητή είναι το μοντέλο εκκαθάρισης (Refinement Model). Κατά τη διάρκεια της εκτέλεσης μιας ροής εργασιών το κομμάτι αυτό προσφέρει πληροφορίες στο υπάρχων μοντέλο με σκοπό να το ξανά ορίσει και να επιβλέπει για πιθανές αλλαγές στις υποδομές π.χ. αναβάθμιση του υπάρχοντος hardware από την τελευταία μέτρηση κτλ. Ο μηχανισμός αυτός προσφέρει ευελιξία στο IReS, βελτιώνοντας την ακρίβεια των μοντέλων κατά τη λειτουργία της πλατφόρμας.

Ο planner όπως αναφέραμε είναι το πιο σημαντικό κομμάτι του βελτιστοποιητή. Το κομμάτι αυτό διερευνά όλα τα διαθέσιμα πλάνα εκτέλεσης και ανακαλύπτει το βέλτιστο που όμως πρωτίστως ικανοποιεί τα χαρακτηριστικά προς βελτιστοποίηση που έχει επιλέξει ο χρήστης.

Αρχικά ο αλγόριθμος δέχεται ως είσοδο τον γράφο της αφηρημένης ροής εργασιών (abstract workflow) εκφραζόμενο ως κατευθυνόμενο ακυκλικό γράφο (DAG) με κόμβους τους τελεστές και τα δεδομένα. Διατηρεί έναν πίνακα δυναμικού προγραμματισμού που είναι υπεύθυνος για την αποθήκευση του καλύτερου πλάνου εκτέλεσης για κάθε διαφορετική μορφή των δεδομένων (π.χ. csv, json κτλ.). Κατόπιν ο planner επεξεργάζεται όλους τους αφηρημένους τελεστές που απαρτίζουν το προς εκτέλεση έργο χρησιμοποιώντας τον αλγόριθμο αναζήτησης κατά βάθους (DFS) σε γράφο. Αυτή η διαδικασία εξασφαλίζει πως κατά την επεξεργασία ενός τελεστή, όλοι οι πρόγονοί του στον γράφο έχουν ήδη επεξεργασθεί και γι'αυτό ο πίνακας περιέχει μέσα τη βέλτιστη λύση ανά είσοδο.

Για κάθε αφηρημένο τελεστή εξερευνάται η βιβλιοθήκη του IReS με σκοπό να βρεθούν όλοι οι αντίστοιχοι υλοποιημένοι τελεστές. Για την επίσπευση της ανωτέρω διαδικασίας ειδικές δομές μετά-δεδομένων σε δεντρική μορφή χρησιμοποιούνται, οι οποίες επιτρέπουν για χάρη της απόδοσης μία δεντρική αντιστοίχιση. Αυτό έχει ως επακόλουθο η πολυπλοκότητα της αντιστοίχισης δύο δέντρων μετά-δεδομένων με έως το πολύ m κόμβους να είναι O(m). Ακόμα μεγαλύτερη αποδοτικότητα επιτυγχάνεται αντιστοιχώντας ειδικά μετά-δεδομένα αρκετά χαρακτηριστικά όπως το όνομα του αλγορίθμου της εκτέλεσης. Έτσι επιλέγονται μόνο οι τελεστές με το σωστό χαρακτηριστικό για να χρησιμοποιηθούν στον παραπάνω αλγόριθμο.

Όταν ανακαλυφθούν όλες οι αντιστοιχίσεις των τελεστών, η επεξεργασία συμβουλεύεται τις λεπτομέρειες εισόδου και εξόδου και προσθέτει τους απαραίτητους τελεστές για ενδιάμεση μεταφορά ή μετατροπή. Αυτοί οι τελεστές χρειάζονται για να ενώσουν τελεστές διαφορετικών μηχανών και συστημάτων διαβάσματος της εισόδου ή/και αποθήκευσης της εξόδου.

Τέλος για να γίνει η εκτίμηση της απόδοσης των μετρήσεων (π.χ. κόστος , χρόνος εκτέλεσης) ο planner συμβουλεύεται τα μοντέλα εκτίμησης για καθέναν από τους υλοποιημένους τελεστές. Αφού εκτιμηθεί το κόστος του τελεστή προσθέτουμε στον πίνακα του δυναμικού προγραμματισμού όλα τα δεδομένα εξόδου. Όταν τελειώσει η επεξεργασία όλων των αφηρημένων τελεστών το βέλτιστο κόστος των δεδομένων που μας ενδιαφέρουν επιστρέφεται χρησιμοποιώντας την αντίστοιχη εγγραφή του πίνακα.

Για να μελετήσουμε την πολυπλοκότητα του παραπάνω αλγορίθμου ας υποθέσουμε ότι μια ροή εργασιών αποτελείται από op αφηρημένους τελεστές με το πολύ m υλοποιημένους τελεστές να αντιστοιχούν σε έναν αφηρημένο. Επιπλέον ας υποθέσουμε πως κάθε τελεστής έχει k εισόδους το πολύ. Για κάθε ενδιάμεσο σύνολο δεδομένων ο πίνακας του δυναμικού προγραμματισμού θα περιέχει το πολύ m εγγραφές κάθε μία παραγόμενη από έναν από τους m υλοποιημένους τελεστές που αντιστοιχίζονται σε έναν αφηρημένο. Έτσι λοιπόν η εσωτερική επανάληψη του αλγορίθμου θα τρέξει το πολύ m φορές, με αποτέλεσμα να παίρνουμε πολυπλοκότητα χειρότερης περίπτωσης για τον βελτιστοποιητή $O(op \times m^2 \times k)$. Παρακάτω στην Εικόνα 3 παρουσιάζεται ο αλγόριθμος.

ALGORITHM 1: *Optimizer*

```
//G(Datasets, Operators) : abstract workflow graph
//Datasets : set of datasets
//Operators : set of abstract operators
//target : target dataset
for d ∈ Datasets do
    //initialize dpTable
    if d.isMaterialized() then
        if d == target then
            return 0;
        dpTable[d].insert(d, 0);
for o ∈ Operators following DAG topological ordering do
    MOperators = findMaterializedOperators(o);
    for mo ∈ MOperators do
        inputCost = 0;
        for in ∈ mo.getInputs() do
            minCost = ∞;
            for tin ∈ dpTable[in] do
                if tin.matchWithOperatorInput(mo)
                 then
                    if tin.getCost < minCost then
                        minCost = tin.getCost;
                else
                    if tin.checkMove(mo) then
                        moveCost = tin.getCost +
                         tin.moveCost(mo);
                        if moveCost < minCost then
                            minCost = moveCost;
            inputCost+ = minCost;
        operatorCost = estimateCost(mo);
        cost = inputCost + operatorCost;
        for out ∈ o.getOutputs() do
            tout = outputFor(mo, out);
            dpTable[out].insert(tout, cost);
return dpTable[target].getMinCost();
```

Εικόνα 3 : Αλγόριθμος επιλογής βέλτιστου πλάνου

Εκτός όμως από την ορθή επιλογή του βέλτιστου συστήματος που θα εκτελεστεί κάθε τελεστής, ο planner του IReS προβλέπει και την συνολική ποσότητα των πόρων που απαιτείται για κάθε ολοκληρωμένη ροή εργασιών ενώ παράλληλα συμμορφώνεται στις απαιτήσεις που έχει ορίσει ο χρήστης πριν από τη διαδικασία της εκτέλεσης. Αυτή η πολιτική μπορεί να αφορά το χρόνο εκτέλεσης ή οποιαδήποτε συνάρτηση κόστους έχει οριστεί από τον χρήστη. Η πρόβλεψη των πόρων επεξεργασίας επεκτείνει την δομή MOEA [12] και βασίζεται στον γενετικό αλγόριθμο NSGA-II [13]

2.1.2.3 Executor Layer

Τέλος στο IReS υπάρχει και το επίπεδο της εκτέλεσης (executor layer) το οποίο εφαρμόζει το βέλτιστο επιλεγμένο πλάνο πάνω στις φυσικές υποδομές που υπάρχουν διαθέσιμες (Hadoop, Spark, Hive, Weka κτλ). Στο επίπεδο αυτό περιέχονται εργαλεία που μεταφράζουν τις εντολές υψηλού επιπέδου, όπως οι πόροι του επιλεγμένου συστήματος και οι μεταφορά ενδιάμεσων αποτελεσμάτων που ίσως χρειαστεί, σε αρχέγονες εντολές όπως αναμένονται από τα συστήματα επεξεργασίας και αποθήκευσης της προς εκτέλεσης ροής εργασιών.

Το παρόν πρωτότυπο εργασίας του IReS βασίζεται στο YARN [14] από τα πιο διαδεδομένα εργαλεία διαχείρισης υπολογιστικών συστοιχιών (clusters) το οποίο είναι υπεύθυνο για την παροχή υπολογιστικών πόρων καθώς και για την χρονοδρομολόγηση διαφόρων σκελετών επεξεργασίας. Επίσης το IReS επεκτείνει το Cloudera Kitten [15] ένα σετ από εργαλεία που χρησιμοποιούνται για τη διαμόρφωση και την εκκίνηση των containers του YARN καθώς επίσης και για το τρέξιμο εφαρμογών μέσα σε αυτά, με σκοπό να υποστηρίξει την εκτέλεση ολόκληρου του κατευθυνόμενου ακυκλικού γράφου των τελεστών και όχι μόνο ενός εξ αυτών.

Η παρακολούθηση της εκτέλεσης των εργασιών συλλαμβάνει σφάλματα και αστοχίες που συμβαίνουν σε πραγματικό χρόνο. Έτσι εξασφαλίζει τη στιβαρή σχεδίαση και διαθεσιμότητα του συστήματος χρησιμοποιώντας δύο μηχανισμούς. Ο πρώτος μηχανισμός είναι υπεύθυνος για την παρακολούθηση της 'υγείας' του συστήματος, εκτελώντας ανά περιόδους ειδικά scripts σε όλους τους κόμβους της υπολογιστικής συστοιχίας. Τα αποτελέσματα αυτά αναφέρονται πίσω στον IReS server. Ο δεύτερος μηχανισμός όπως φαίνεται και από την Εικόνα 4 ελέγχει τη διαθεσιμότητα των υπόλοιπων συστημάτων είτε επεξεργασίας (π.χ. Spark) είτε αποθήκευσης (π.χ. HDFS) που είναι απαραίτητα για το πλάνο εκτέλεσης.

Service	Status	Action
MLLib	true	stop
Spark	true	stop
WEKA	true	stop
MapReduce	true	stop
Python	true	stop

Εικόνα 4 : Έλεγχος διαθεσιμότητας συστημάτων στο γραφικό περιβάλλον του IReS

Αυτές οι πληροφορίες παρέχονται τόσο κατά τη διάρκεια της δημιουργίας του πλάνου εκτέλεσης, όσο και κατά την ίδια την εκτέλεση κάθε ροής εργασιών. Κατά τη φάση δημιουργίας του βέλτιστου πλάνου , οι μη διαθέσιμες μηχανές απορρίπτονται και οι πόροι που θα χρησιμοποιηθούν προβλέπονται λαμβάνοντας υπόψιν μόνον αυτούς που είναι διαθέσιμοι τη δεδομένη στιγμή. Σε περίπτωση αστοχίας σε πραγματικό χρόνο κατά τη διάρκεια της εκτέλεσης των εργασιών, το υπολοιπόμενο πλάνο ξανά προγραμματίζεται προς εκτέλεση χωρίς όμως το σύστημα να αποβάλλει τα ενδιάμεσα αποτελέσματα που είχαν παραχθεί μέχρι τη στιγμή της αστοχίας.

2.2 RHEEM

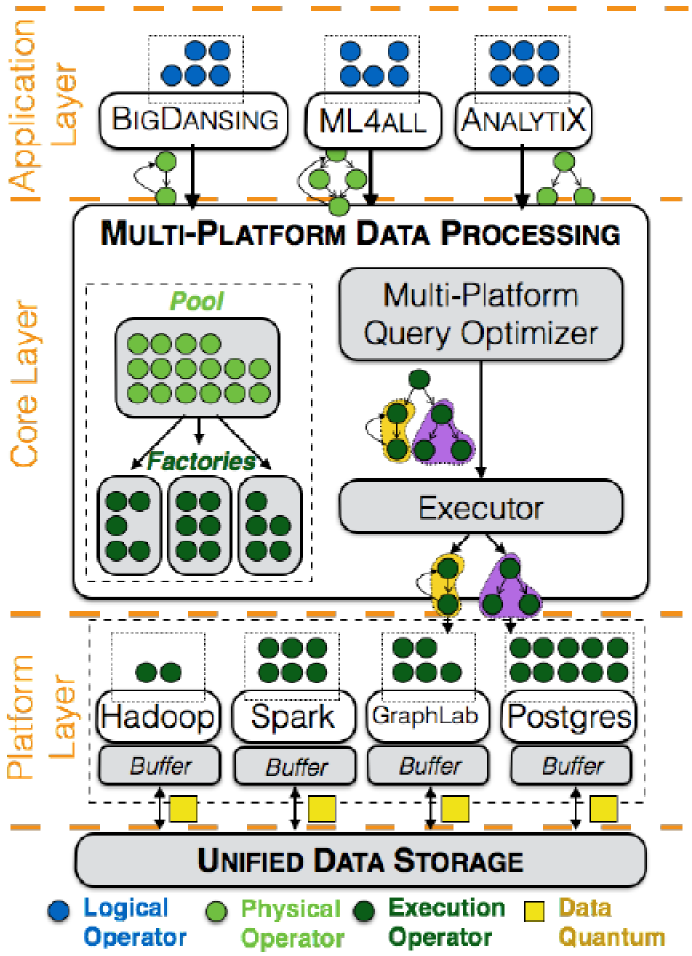
Στο παρόν υπό-κεφάλαιο θα αναλύσουμε το θεωρητικό υπόβαθρο του Rheem το οποίο υλοποιήθηκε για να επιλύσει τα προβλήματα που έχουν προαναφερθεί και αφορούν τις πολλές επιλογές που υπάρχουν πλέον όσον αφορά την αποθήκευση και την επεξεργασία δεδομένων μεγάλου όγκου. Το Rheem παρέχει τρία επίπεδα επεξεργασίας δεδομένων και αποθήκευσης, με σκοπό να επιτύχει τόσο την ανεξαρτησία των συστημάτων όσο και τη διαλειτουργικότητα μεταξύ των ποικίλων πλατφορμών που κυριαρχούν στην αγορά. Το Rheem καλείται να επιλύσει δύο βασικά προβλήματα που συναντώνται στον τομέα της ανάλυσης των δεδομένων.

Το πρώτο από αυτά είναι ότι οι εφαρμογές εξαρτώνται στενά από συγκεκριμένα συστήματα επεξεργασίας , κάνοντας τη μεταφορά τους σε καινούρια και πιο αποδοτικά συστήματα ένα πολύ δύσκολο και κοστοβόρο έργο. Επίσης συνήθως οι πιο πολύπλοκες εφαρμογές απαιτούν τα διάφορα κομμάτια τους να υλοποιούνται σε διαφορετικές μηχανές, με αποτέλεσμα ο χρήστης να πρέπει μετά να συνδέσει και να αναλύσει τα αποτελέσματα χειροκίνητα, έτσι ώστε να καταλήξει στο τελικό αποτέλεσμα [16]. Τέλος πολλές φορές κρίνεται απαραίτητο να υλοποιηθούν εξαρχής ολόκληρες οι εφαρμογές σε καινούρια συστήματα που είναι αποδεδειγμένα πιο γρήγορα όπως για παράδειγμα σε Spark αντί του Hadoop.

Το δεύτερο κύριο πρόβλημα είναι πως τα σύνολα δεδομένων συνήθως παράγονται από διαφορετικές πηγές και αυτό έχει ως επακόλουθο να αποθηκεύονται και σε διαφορετικά συστήματα. Αυτό οδηγεί τους χρήστες να πρέπει να βρίσκουν τρόπους να τα μεταφέρουν σε ένα κοινό, να αλλάζουν τη μορφή τους για να γίνονται αποδεκτά από τις διαφορετικές μηχανές αποθήκευσης και εν τέλει να μπορούν να τα επεξεργαστούν. Προφανώς η παραπάνω διαδικασία απαιτεί πολύ χρόνο και δεν έχει να προσφέρει τίποτα θετικό ως προς την τελική ανάλυση και εξόρυξη των αποτελεσμάτων.

Αυτοί είναι οι δύο κυριότεροι λόγοι που οδήγησαν στη δημιουργία του Rheem. Η αφηρημένη επεξεργασία που το διέπει επιτρέπει στους χρήστες να επικεντρώνονται μόνο στη λογική των προγραμμάτων τους, αλλά ταυτόχρονα επιτρέπει και στις ίδιες τις εφαρμογές να μην εξαρτώνται από συγκεκριμένα συστήματα ανάλυσης και επεξεργασίας. Αυτό έχει ως αποτέλεσμα το Rheem να μπορεί να επιλέξει να εκτελέσει μια εργασία ταυτόχρονα σε διαφορετικές πλατφόρμες έτσι ώστε να βελτιστοποιήσει την απόδοση.

Στην Εικόνα 5 παρακάτω παρουσιάζονται τα 3 επίπεδα πάνω στα οποία είναι δομημένο το Rheem. Το πρώτο επίπεδο, αυτό της εφαρμογής (application layer) είναι υπεύθυνο για τη μοντελοποίηση της λογικής που παρέχει ο χρήστης. Το κυρίως ή κεντρικό επίπεδο (core layer) παρέχει την ενδιάμεση αναπαράσταση μεταξύ των εφαρμογών και των συστημάτων επεξεργασίας. Το τρίτο και τελευταίο επίπεδο της πλατφόρμας (platform layer) συγκεντρώνει όλα τα παραπάνω συστήματα επεξεργασίας.



Εικόνα 5 : Η Αρχιτεκτονική του Rheem

Σε αντίθεση με τα κοινά συστήματα διαχείρισης βάσεων δεδομένων το Rheem διαχωρίζει το φυσικό επίπεδο και το επίπεδο εκτέλεσης. Αυτός ο διαχωρισμός επιτρέπει στις εφαρμογές να εκφράζουν το φυσικό πλάνο τους με βάση τις αλγοριθμικές τους ανάγκες χωρίς να είναι δεμένες με κάποιο συγκεκριμένο σύστημα επεξεργασίας.

Η επικοινωνία ανάμεσα σε αυτά τα 3 επίπεδα γίνεται με συναρτήσεις ορισμένες από το χρήστη (UDFs). Σκοπός του Rheem είναι να λαμβάνει πολύπλοκα αναλυτικά έργα, να τα διαιρεί άψογα σε υπό-έργα, να χρονοδρομολογεί καθένα από αυτά στη βέλτιστη πλατφόρμα επεξεργασίας, να παρακολουθεί την εκτέλεσή τους και τέλος να συναθροίζει τα αποτελέσματα.

Κατά τη συγγραφή της παρούσας εργασίας το Rheem παρέχει τη δυνατότητα της αφηρημένης λογικής που αναλύσαμε παραπάνω μόνο όσον αφορά την επεξεργασία των δεδομένων, ενώ γίνονται προσπάθειες να επιτευχθεί η ίδια λογική όσον αφορά και τα συστήματα αποθήκευσης του εκάστοτε συνόλου δεδομένων .

2.2.1 Αφηρημένη επεξεργασία δεδομένων στο Rheem

Το Rheem παρέχει τριών ειδών τελεστές, λογικούς, φυσικούς τελεστές και τελεστές εκτέλεσης. Συγκεκριμένα η είσοδος στο επίπεδο της εφαρμογής είναι οι λογικοί τελεστές που παρέχονται από το χρήστη και η έξοδος είναι το φυσικό πλάνο (physical plan). Το φυσικό πλάνο περνιέται ως είσοδος στο κύριο επίπεδο (core layer) όπου λαμβάνουν χώρα βελτιστοποιήσεις σε πολύ-συστηματικό επίπεδο έτσι ώστε να παραχθεί το πλάνο εκτέλεσης της εφαρμογής (execution plan).

2.2.1.1 Application Layer

Ένας λογικός τελεστής δεν είναι παρά μια συνάρτηση που παρέχεται από το χρήστη (UDF) που συμπεριφέρεται σαν μια ειδική μονάδα της εφαρμογής της επεξεργασίας των δεδομένων. Μπορεί κανείς να ορίσει τον λογικό τελεστή ως ένα ειδικό πρότυπο (template) όπου οι χρήστες παρέχουν όλες τις λογικές διαδικασίες των εργασιών τους. Αυτή η αφηρημένη λογική ενεργοποιεί τόσο την ευκολία στη χρήση, αφού κρύβει ουσιαστικά δυσνόητες λεπτομέρειες της υλοποίησης, όσο και την επίτευξη υψηλής επίδοσης επιτρέποντας διάφορες βελτιστοποιήσεις.

Οι λογικοί τελεστές δουλεύουν πάνω σε data quanta που είναι οι μικρότερες χαρακτηριστικές μονάδες ενός συνόλου δεδομένων. Για παράδειγμα σε περίπτωση που το σύνολο δεδομένων αφορά πολλαπλές εγγραφές σε μία βάση δεδομένων , data quanta θεωρείται η μία εγγραφή. Η υποδιαίρεση σε τόσο μικρά κομμάτια δεδομένων επιτρέπει στο Rheem να εφαρμόζει τους τελεστές σε παραλληλισμό και έτσι να επιτυγχάνει καλύτερη επίδοση.

Σε αυτό το σημείο αξίζει να αναφέρουμε ένα παράδειγμα για την καλύτερη κατανόηση του παρόντος επιπέδου του Rheem. Ας υποθέσουμε ότι θέλουμε να προσφέρουμε στους χρήστες λογικούς τελεστές ώστε να υλοποιήσουν αλγορίθμους μηχανική μάθησης. Ο προγραμματιστής λοιπόν θα πρέπει να προσφέρει τρεις βασικούς τελεστές. Έναν για αρχικοποίηση των ειδικών παραμέτρων του αλγορίθμου, έναν για την επεξεργασία των υπολογισμών που απαιτούνται από τον αλγόριθμο και τέλος έναν τελεστή επανάληψης που να ικανοποιεί τις προϋποθέσεις τερματισμού. Έτσι χρησιμοποιώντας τους παραπάνω τελεστές οι χρήστες μπορούν να υλοποιήσουν αλγορίθμους όπως οι SVM , K-Means [17].

2.2.1.2 Core Layer

Το κεντρικό ή κυρίως επίπεδο είναι η 'καρδιά' του Rheem. Εκθέτει μία γκάμα από φυσικούς τελεστές, όπου ο καθένας αντιπροσωπεύει μια αλγοριθμική απόφαση για την εκτέλεση μιας αναλυτικής εργασίας. Οι φυσικοί τελεστές είναι η αγνωστικιστική, όσον αφορά την πλατφόρμα, υλοποίηση των λογικών τελεστών. Αυτοί οι τελεστές παρέχονται στον προγραμματιστή έτσι ώστε να αναπτύξει μια εφαρμογή πάνω στο Rheem. Στο παραπάνω παράδειγμα ο βελτιστοποιητής του Rheem αντιστοιχεί τον λογικό τελεστή της αρχικοποίησης σε έναν Map φυσικό τελεστή και τον τελεστή επεξεργασίας σε έναν GroupBy φυσικό τελεστή. Το Rheem παρέχει δύο διαφορετικές υλοποιήσεις για τον GroupBy τελεστή, επομένως ο βελτιστοποιητής του κυρίως επιπέδου θα πρέπει να διαλέξει αυτόν που ικανοποιεί την εφαρμογή περισσότερο.

Όταν η εφαρμογή έχει παράγει το φυσικό πλάνο για το δοσμένο έργο προς εκτέλεση, το Rheem διαιρεί το πλάνο σε υπό-εργασίες που τις ονομάζουμε μονάδες της εκτέλεσης. Κάθε τέτοια μονάδα θα δρομολογηθεί από το Rheem να εκτελεστεί σε μία μοναδική πλατφόρμα επεξεργασίας. Ύστερα μεταφράζει όλα τα υπό-έργα σε ένα εκτελεστικό πλάνο βελτιστοποιώντας το καθένα ξεχωριστά ανάλογα με το προς εκτέλεση σύστημα που έχει δρομολογηθεί. Τέλος προγραμματίζει την εκτέλεση της κάθε υπό-εργασίας στο εκάστοτε σύστημα. Από τα παραπάνω φαίνεται πως αντίθετα με τα κλασικά συστήματα διαχείρισης βάσεων δεδομένων το Rheem μπορεί να παράγει εκτελεστικό πλάνο που να πρέπει να εκτελεστεί σε πολλαπλές πλατφόρμες και να μην δεσμεύεται από μία μοναδική.

2.2.1.3 Platform Layer

Σε αυτό το επίπεδο, οι τελεστές εκτέλεσης (execution operators) ορίζουν πως θα εκτελεστεί κάθε εργασία και κάτω από ποιο συγκεκριμένο σύστημα. Με άλλα λόγια οι τελεστές εκτέλεσης είναι ουσιαστικά η υλοποίηση των φυσικών τελεστών με εξάρτηση όμως από κάποιο σύστημα επεξεργασίας. Στο παράδειγμά μας δηλαδή οι εκτελεστικοί τελεστές MapPartitions και ReduceByKey του Spark είναι ένας τρόπος έκφρασης του λογικού τελεστή αρχικοποίησης και επεξεργασίας αντιστοίχως.

Σε αντίθεση με τους λογικούς τελεστές, οι τελεστές εκτέλεσης δουλεύουν πάνω σε πολλαπλές μονάδες δεδομένων, γεγονός που τους κάνει να ενεργοποιούνται με μία κλήση συνάρτησης .

2.2.1.4 Αντιστοιχία μεταξύ των τελεστών (Operator Mappings)

Ο ορισμός αντιστοιχίας (mapping) μεταξύ των τελεστών εκτέλεσης και των φυσικών τελεστών είναι αρμοδιότητα του εκάστοτε προγραμματιστή, όποτε μια καινούρια μηχανή συνδέεται στον πυρήνα του Rheem. Στόχος είναι οι αντιστοιχίσεις να βασίζονται σε ένα δομημένο μοντέλο όπου οι τελεστές θα χαρακτηρίζονται από συγκεκριμένες πληροφορίες περιεχομένου. Αυτές οι πληροφορίες θα βοηθούν ουσιαστικά το Rheem να διαλέγει μεταξύ διαφορετικών υλοποιήσεων για τον ίδιο τελεστή, ανάλογα με τις απαιτήσεις της εκάστοτε εφαρμογής. Στο παράδειγμά μας παραπάνω ο λογικός τελεστής της επεξεργασίας αντιστοιχίζεται σε δύο διαφορετικούς φυσικούς τελεστές (SortGroupBy και HashGroupBy). Σε αυτήν την περίπτωση ο προγραμματιστής μέσω των πληροφοριών, θα παρέχει την απαραίτητη βοήθεια στον βελτιστοποιητή ώστε να διαλέξει τον κατάλληλο φυσικό τελεστή ανάλογα με την περίσταση. Τέλος ο προγραμματιστής θα είναι υπεύθυνος για την δημιουργία αυτών των τελεστών στη φυσική τους υπόσταση (physical operators) και το σύστημα θα αναλαμβάνει μετά να τους μεταφράσει σε τελεστές εκτέλεσης.

2.2.2 Αλληλεπίδραση με το χρήστη

Οι χρήστες διαφοροποιούνται σε δύο κατηγορίες, στους τελικούς χρήστες (end users) οι οποίοι αλληλεπιδρούν με τις εφαρμογές και στους προγραμματιστές που αλληλεπιδρούν με το σύστημα και τα τρία του επίπεδα. Παρακάτω, αναλύουμε τον τρόπο με τον οποίο οι προγραμματιστές ορίζουν τις συναρτήσεις περιγραφής σε καθένα από τα τρία επίπεδα.

2.2.2.1 Application Layer

Σε αυτό το επίπεδο οι προγραμματιστές μοντελοποιούν μια εφαρμογή ανάλυσης δεδομένων προσδιορίζοντας ένα σύνολο αφηρημένων λογικών τελεστών. Οι τελικοί χρήστες υλοποιούν αυτούς τους τελεστές με σκοπό να εκφράσουν το ειδικό δικό τους έργο ανάλυσης. Το Rheem παρέχει έναν ειδικό αφηρημένο λογικό τελεστή (LogicalOperator) ο οποίος ορίζει τη μέθοδο applyOp. Οι λογικοί τελεστές οποιασδήποτε εφαρμογής επεκτείνουν αυτόν τον λογικό τελεστή και παρέχουν μια υλοποίηση της μεθόδου applyOp. Το Rheem επικαλείται αυτή την παραγόμενη μέθοδο σε πραγματικό χρόνο εκτέλεσης για να εφαρμόσει κάποιον λογικό τελεστή. Εκτός όμως από τους λογικούς τελεστές ο προγραμματιστής μπορεί να εκθέσει επίσης μια ειδική γλώσσα δηλώσεων με σκοπό οι χρήστες να μπορούν να ορίσουν καλύτερα τις εργασίες τους (π.χ. queries). Η εφαρμογή είναι έπειτα υπεύθυνη να μεταφράσει τις παραπάνω δηλώσεις σε ένα λογικό πλάνο και ύστερα ο βελτιστοποιητής της εφαρμογής να μεταφράσει το λογικό πλάνο σε ένα φυσικό πλάνο.

2.2.2.2 Core Layer

Το Rheem παρέχει μια μεγάλη γκάμα από φυσικούς τελεστές που βοηθούν τις εκάστοτε εφαρμογές να παράγουν το φυσικό τους πλάνο. Για να υπάρχει επεκτασιμότητα το σύστημα παρέχει τον αφηρημένο φυσικό τελεστή (PhysicalOperator) με την αφηρημένη μέθοδο applyOp, έτσι ώστε οι προγραμματιστές να μπορούν να ορίσουν τους δικούς τους φυσικούς τελεστές. Ο ορισμός ενός καινούριου φυσικού τελεστή έχει σκοπό να ικανοποιήσει δύο διαφορετικές ανάγκες.

Αρχικά ο προγραμματιστής ορίζει έναν τελεστή περιτύλιξης (wrapper operator) έτσι ώστε να εκτελέσει τον λογικό τελεστή μαζί με κάποιες φυσικές λεπτομέρειες, όπως αλγοριθμικές αποφάσεις. Αυτός ο τελεστής ακολουθεί την υπογραφή του λογικού τελεστή. Κατά δεύτερον επειδή υπάρχει περίπτωση η έξοδος κάποιου τελεστή να μην μπορεί να χρησιμοποιηθεί ως είσοδος σε κάποιον μεταγενέστερο τελεστή ο προγραμματιστής ορίζει έναν ενισχυτικό τελεστή (enhancer operator) που συμπληρώνει τα πιθανά κενά μεταξύ των wrapper τελεστών.

Για παράδειγμα ας υποθέσουμε ότι θέλουμε να τρέξουμε τον K-Means αλγόριθμο. Υπάρχει περίπτωση ο αλγόριθμός μας να χρησιμοποιεί δύο λογικούς τελεστές τον GetCentroid που δίνει το κοντινότερο κέντρο μιας μονάδας δεδομένων, και τον SetCentroids που υπολογίζει τα νέα κέντρα. Το πρόβλημα εντοπίζεται στο ότι ο πρώτος τελεστής βγάζει στην έξοδο ένα σημείο και το κοντινότερό του κέντρο, ενώ ο δεύτερος τελεστής χρειάζεται ως είσοδο ένα κέντρο και όλα τα κοντινά του σημεία. Το πρόβλημα αυτό επιλύεται με την παροχή από τον προγραμματιστή ενός GroupBy ενισχυτικού τελεστή (enhancer operator) ανάμεσα στον GetCentroid και στον SetCentroid.

2.2.2.3 Platform Layer

Για τη μοντελοποίηση μιας πλατφόρμας επεξεργασίας δεδομένων, οι προγραμματιστές επεκτείνουν τον αφηρημένο τελεστή εκτέλεσης (ExecutionOperator) και υλοποιούν τη μέθοδό του applyOp. Εδώ υπάρχουν δύο σενάρια που μπορεί να προκύψουν.

Αν έχει οριστεί ένας καινούριος φυσικός τελεστής, επειδή για παράδειγμα ο προγραμματιστής θέλει να εισάγει μια καινούρια εφαρμογή, τότε θα πρέπει ο φυσικός τελεστής να ακολουθηθεί από έναν καινούριο τελεστή εκτέλεσης στο πραγματικό σύστημα που θα πραγματοποιηθεί η εκτέλεση. Στο δεύτερο σενάριο ο προγραμματιστής θέλει να εισάγει μια καινούρια πλατφόρμα στο επίπεδο εκτέλεσης. Σε αυτήν την περίπτωση κάθε φυσικός τελεστής θα πρέπει να υποστηρίζεται από τον αντίστοιχο τελεστή εκτέλεσης στον καινούριο σύστημα.

Το Rheem χρησιμοποιεί αυτούς τους τελεστές εκτέλεσης για να παράγει το εκτελεστικό πλάνο και για να μεταφέρει τις λεπτομέρειες της εκτέλεσης στην αντίστοιχη πλατφόρμα, όπως είναι η διανομή των δεδομένων, η παράλληλη εκτέλεση και η αποθήκευση. Τέλος η επιλεγμένη πλατφόρμα εκτέλεσης απλά εκτελεί το πλάνο εκτέλεσης με τα δικά της δεδομένα και το δικό της επεξεργαστικό μοντέλο.

2.2.3 Πολυεπίπεδη βελτιστοποίηση

Σε αντίθεση με τα συνηθισμένα συστήματα διαχείρισης δεδομένων, τα οποία είναι δεμένα με ένα συγκεκριμένο σύστημα επεξεργασίας, σκοπός του Rheem είναι όχι μόνο η ανεξαρτησία όσον αφορά τις πλατφόρμες αλλά και η πολύ-πλατφορμική εκτέλεση (multi-platform execution). Για την επίτευξη και των δύο παραπάνω είναι αναγκαίο να γίνονται βελτιστοποιήσεις σε όλα τα επίπεδα του Rheem. Στο επίπεδο της εφαρμογής επικυρώνεται η εργασία που παρέχεται στην είσοδο, μεταφράζεται στο λογικό πλάνο και μετά παράγεται το βέλτιστο φυσικό πλάνο. Στο κύριο επίπεδο το φυσικό πλάνο, μεταφράζεται σε εκτελεστικό πλάνο, και υποδιαιρεί τις εργασίες σε μικρές μονάδες εκτέλεσης. Τέλος στο επίπεδο της πλατφόρμας (platform layer) συμβαίνει μια ακόμα πιο λεπτή υποδιαίρεση των παραπάνω μονάδων, βασιζόμενη στο πραγματικό σύστημα που θα αναλάβει εν τέλει την εκτέλεση.

Ο βελτιστοποιητής είναι υπεύθυνος για τη μετάφραση ενός δοσμένου αφηρημένου Rheem-πλάνου στο πιο αποδοτικό πλάνο εκτέλεσης. Το πλάνο εκτέλεσης που προκύπτει αποτελείται από ένα σύνολο από υπό-πλάνα που αφορούν λεπτομέρειες του κάθε συστήματος. Αξίζει να σημειωθεί εδώ πως το πρόβλημα της βελτιστοποίησης είναι αρκετά διαφορετικό από το αντίστοιχο στις κλασικές βάσεις δεδομένων. Αρχικά το Rheem από τη φύση του είναι επεκτάσιμο τόσο στο πεδίο των τελεστών που παρέχει, όσο και στο ποιες πλατφόρμες υποστηρίζει. Αυτό έχει ως επακόλουθο την ανάγκη ο βελτιστοποιητής να είναι κι αυτός επεκτάσιμος όσον αφορά τη μετάφραση Rheem-πλάνων σε πραγματικά πλάνα εκτέλεσης σε κάποιο σύστημα. Τέλος η αφηρημένη ανάλυση δεδομένων που υποστηρίζει το Rheem βασίζεται σε συναρτήσεις οριζόμενες από το χρήστη (UDFs), με αποτέλεσμα οι τελεστές να εμφανίζονται στον βελτιστοποιητή ως μαύρα-κουτιά, γεγονός που κάνει τη δημιουργία συναρτήσεων κόστους αρκετά δυσκολότερη.

Αρχικά το Rheem για να αντιμετωπίσει τα παραπάνω προβλήματα, εφαρμόζει ένα επεκτάσιμο σύνολο από μετασχηματισμούς γράφων στο Rheem-πλάνο, έτσι ώστε να βρει διαφορετικά πλάνα εκτέλεσης. Ύστερα συγκρίνει τα διαφορετικά πλάνα που βρήκε χρησιμοποιώντας τις συναρτήσεις κόστους του τελεστή εκτέλεσης (ExecutionOperator). Οι παραπάνω συναρτήσεις μπορούν είτε να δίνονται είτε να είναι προϊόντα εκπαίδευσης και παραμετροποιούνται με σεβασμό πάντα προς τους υπάρχοντες πόρους του υλικού που υπάρχει (π.χ. αριθμός των υπολογιστικών κόμβων για κατανεμημένους τελεστές). Λόγω της αντιμετώπισης των τελεστών ως μαύρα-κουτιά από τον βελτιστοποιητή, ειδικές βελτιστοποιήσεις που αφορούν συγκεκριμένους τομείς πρέπει να γίνουν σε συνεργασία με την εκάστοτε εφαρμογή. Το Rheem επιτρέπει στις εφαρμογές να εκθέτουν σημασιολογικές παραμέτρους που αφορούν τις συναρτήσεις τους, καθώς επίσης παρέχει κάποιες υποδείξεις (π.χ. αριθμός επαναλήψεων), περιορισμούς και εναλλακτικά πλάνα.

Όταν ο βελτιστοποιητής αποφασίσει τελικά για κάποιο πλάνο εκτέλεσης η μηχανή το εκτελεί χρονοδρομολογώντας τα διαφορετικά υπό-πλάνα, ενορχηστρώνοντας τη μεταφορά δεδομένων ανάμεσα στις πλατφόρμες και μαζεύοντας στατιστικά στοιχεία της εκτέλεσης για την περεταίρω καλυτέρευση του βελτιστοποιητή.

3 Αξιολόγηση και Σύγκριση IReS – Rheem

3.1 Δημιουργία Workflow στο IReS

Για τη δημιουργία ενός workflow στο IReS πρέπει να ακολουθηθεί μια σειρά από βήματα. Σε πρώτη φάση θα παρουσιαστεί το WordCount workflow που αποτελείται από έναν μόνο αφηρημένο τελεστή που αντιστοιχίζεται σε τρεις υλοποιημένους, έναν σε Hadoop , έναν σε Spark κι έναν σε Java. Σκοπός είναι να εκπαιδεύσουμε και τους 3 τελεστές με παράμετρο το μέγεθος του κειμένου για διάφορα μεγέθη και σε επόμενο στάδιο να δούμε τις επιλογές του IReS ανάλογα με την είσοδο που του δίνουμε. Φυσικά αναμένουμε στις μικρές εισόδους να διαλέγει την υλοποίηση της Java και σε μεγαλύτερα κείμενα μία εκ των Hadoop ή Spark.

Για τη δημιουργία του WordCount_Hadoop τελεστή αρχικά δημιουργούμε έναν φάκελο με το όνομα του τελεστή στο μονοπάτι των τελεστών του IReS server. Αυτός ο φάκελος θα πρέπει να περιέχει τελικά όλα τα αρχεία που θα χρειαστεί το IReS για να εκτελέσει το workflow στην περίπτωση που επιλεγεί από το σύστημα ως το βέλτιστο πλάνο.

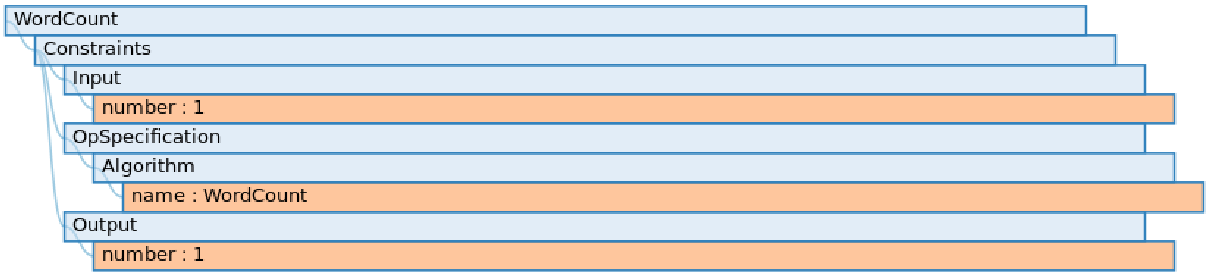
Πρώτο μας μέλημα είναι λοιπόν να δώσουμε την περιγραφή του τελεστή σε ένα ειδικό αρχείο που ονομάζεται description. Αρχικά πρέπει να ορίσουμε το σύστημα εκτέλεσης , που σε αυτήν την περίπτωση είναι το Hadoop. Έπειτα ορίζουμε τον αριθμό των arguments των εισόδων και τον αριθμό των εξόδων, και το όνομα του αλγορίθμου εκτέλεσης. Κατόπιν πρέπει να ορίσουμε και τις παραμέτρους της εκτέλεσης δηλαδή το μονοπάτι στο HDFS που είναι αποθηκευμένο το dataset μας, τον αριθμό των arguments που χρησιμοποιούμε και τέλος το όνομα του script που συνδέει το IReS με το YARN μέσω του Cloudera Kitten . Τέλος εδώ δηλώνουμε και οποιαδήποτε βελτιστοποίηση θέλουμε να κάνουμε, στη δικιά μας περίπτωση όπως προαναφέραμε η παράμετρός μας θα είναι το μέγεθος του αρχείου. Στο αρχείο αυτό ουσιαστικά δηλώνουμε όλα τα μετά-δεδομένα που θα χρειαστούν για την εκτέλεση του τελεστή.

Το δεύτερο αρχείο που χρειάζεται να δημιουργήσουμε είναι αυτό που συνδέει το αρχείο του Cloudera Kitten που θα πρέπει να έχει την κατάληξη .lua , κι εκεί θα πρέπει να δηλώσουμε όλα τα υπόλοιπα αρχεία και scripts καθώς και τα ολοκληρωμένα μονοπάτια τους, που χρειάζεται η εφαρμογή μας. Στο συγκεκριμένο παράδειγμα χρειαζόμαστε το jar αρχείο του map-reduce προγράμματός μας καθώς και το όνομα του script που θα περιέχει τις εντολές προς εκτέλεση.

Το τελευταίο βήμα για τη δημιουργία του τελεστή είναι το script που προαναφέρθηκε.

Φυσικά αυτό που δημιουργήσαμε παραπάνω είναι ο υλοποιημένος τελεστής του WordCount που τρέχει στο Hadoop και προς το παρόν δεν έχει αντιστοιχηθεί στον αφηρημένο. Επομένως για τη δημιουργία του αφηρημένου οδηγούμαστε στο φάκελο abstractOperators και φτιάχνουμε ένα αρχείο που θα πρέπει να έχει ως όνομα αλγορίθμου το ίδιο όνομα που χρησιμοποιήσαμε παραπάνω καθώς επίσης και τον ίδιο αριθμό των εισόδων και εξόδων. Στο γραφικό περιβάλλον του IReS μπορούμε να δούμε την δεντρική δομή του αφηρημένου τελεστή WordCount στην Εικόνα 6.

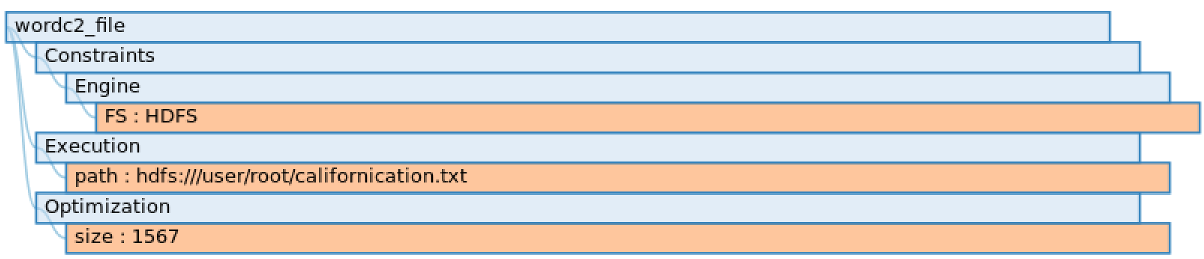
Abstract Operator: WordCount



Εικόνα 6 : Αφηρημένος τελεστής WordCount

Στη συνέχεια οδηγούμαστε στο φάκελο datasets του IReS server και δημιουργούμε το αρχείο που θα χρησιμοποιηθεί για να ληφθούν οι πληροφορίες της εισόδου του τελεστή. Το αρχείο αυτό θα πρέπει μα περιέχει υποχρεωτικά το μονοπάτι που οδηγεί στο κατανεμημένο σύστημα αποθήκευσης που θέλουμε να χρησιμοποιηθεί. Στην Εικόνα 7 φαίνεται η δεντρική δομή του αρχείου αυτού στο γραφικό περιβάλλον του IReS. Είναι σημαντικό να τονιστεί πως έχουμε δηλώσει και το μέγεθος του αρχείου που θα χρησιμοποιηθεί.

Dataset: wordc2_file



Εικόνα 7 : Αναπαράσταση συνόλου δεδομένων εισόδου στο IReS

Τέλος απομένει η δημιουργία της αφηρημένης ροής εργασιών (abstract workflow) και γι' αυτό το σκοπό οδηγούμαστε στο φάκελο abstrackWorkflows. Ο φάκελος αυτός πρέπει να περιέχει το αρχείο του γράφου που ορίζει την είσοδο και την έξοδο κάθε τελεστή καθώς και τον τελικό προορισμό της εξόδου. Ο γράφος στη δική μας περίπτωση είναι ο παρακάτω :

wordc2_file,WordCount

WordCount,d1

d1,$$target

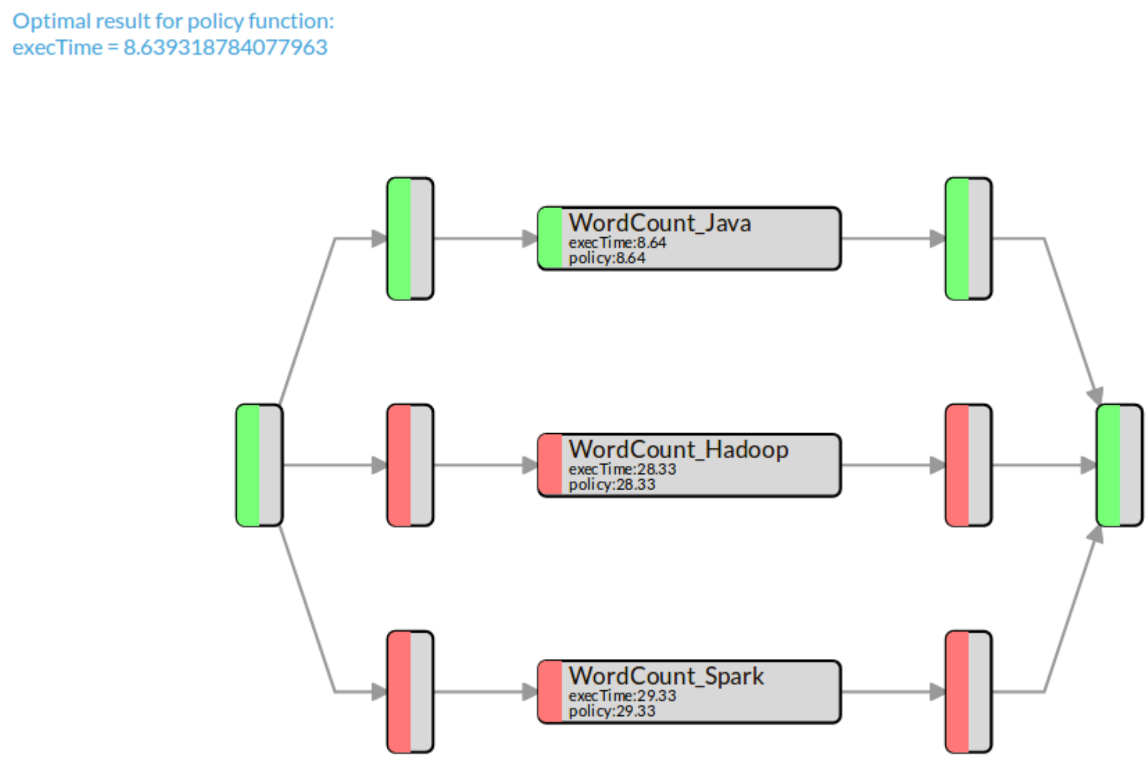
Ουσιαστικά αυτό που δηλώνουμε εδώ είναι πως το dataset που θα χρησιμοποιήσουμε που φαίνεται και στην Εικόνα 6 θα δοθεί ως είσοδος στον αφηρημένο τελεστή WordCount που δημιουργήσαμε προηγουμένως. Ύστερα ο τελεστής WordCount θα εκτελεστεί, επιλέγοντας κάποιον από τους υλοποιημένους τελεστές που του αντιστοιχούν, και θα γράψει τα αποτελέσματά του στο ενδιάμεσο αρχείο d1, και αφού δεν υπάρχει κάποιος άλλος τελεστής μετέπειτα το αρχείο d1 θα γράψει στην έξοδο, που έχει οριστεί στο description αρχείο του εκάστοτε υλοποιημένου τελεστή. Τέλος δημιουργούμε ακόμα δύο φακέλους με ονόματα datasets και operators όπου θα πρέπει να περιέχουν μέσα τα αρχεία των δεδομένων εισόδου και τους αφηρημένους τελεστές που θα χρησιμοποιήσουμε αντίστοιχα.

3.2 Εκπαίδευση και Δοκιμές στο IReS

Το IReS για να δημιουργήσει το μοντέλο του εκάστοτε τελεστή χρειάζεται να μαζέψει στοιχεία μέσω του ganglia [18] τουλάχιστον από δύο εκτελέσεις τα οποία και αποθηκεύει τοπικά σε μία mongo DB. Έτσι λοιπόν εκπαιδεύουμε τον τελεστή που δημιουργήσαμε με διαφορετικά αρχεία ως προς το μέγεθος.

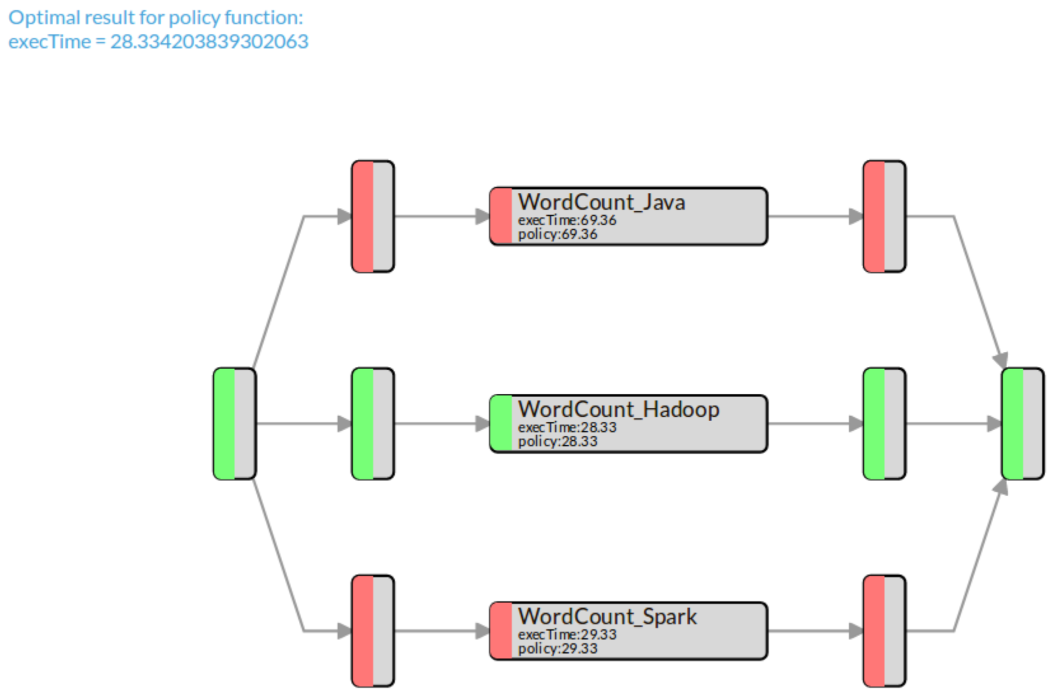
Ακολουθούμε την παραπάνω διαδικασία, από τη φάση της δημιουργίας έως και τη φάση της εκπαίδευσης και για τους άλλους δύο materialized operators και αφού ολοκληρωθεί αυτή η διαδικασία δοκιμάζουμε εκτελέσεις με διαφορετικού μεγέθους αρχεία στην είσοδο.

Στην Εικόνα 8 παρακάτω φαίνεται η επιλογή του IReS όσον αφορά αρχείο εισόδου μεγέθους 10MB . Όπως ήταν αναμενόμενο το IReS επιλέγει να γίνει η εκτέλεση με τον τελεστή της Java καθώς είναι ασύμφορη η ενεργοποίηση τόσο του Hadoop όσο και του Spark για τόσο μικρή είσοδο.



Εικόνα 8 : Η επιλογή του IReS σε αρχείο εισόδου 10MB

Στην επόμενη εκτέλεση αυξάνουμε το μέγεθος του αρχείου στα 800MB και αναμένουμε την επιλογή ανάμεσα στο Hadoop και το Spark. Πράγματι βασιζόμενο στα μοντέλα που έχει δημιουργήσει το IReS επιλέγει να εκτελέσει το workflow με τον τελεστή του Hadoop, όπως φαίνεται στην Εικόνα 9.



Εικόνα 9 : Η επιλογή του IReS σε αρχείο εισόδου 800MB

3.3 Δημιουργία workflow στο Rheem

Για να μπορέσουμε να συγκρίνουμε τα δύο συστήματα προσπαθήσαμε να δημιουργήσουμε τα ίδια workflows και στο IReS και στο Rheem. Το Rheem όμως ακολουθεί διαφορετική προσέγγιση όσον αφορά τη δημιουργία ενός workflow.

Αρχικά το Rheem είναι περισσότερο μια προγραμματιστική βιβλιοθήκη παρά ένα σύστημα όπως το IReS. Για τη δημιουργία ακόμα και του ευκολότερου τελεστή όπως του WordCount το Rheem απαιτεί βαθιά κατανόηση της διαδικασίας , αφού στην πραγματικότητα ο χρήστης θα πρέπει να δημιουργήσει εξαρχής το πρόγραμμα χρησιμοποιώντας τις βιβλιοθήκες του Rheem. Πιο συγκεκριμένα στην περίπτωση που ο τελεστής δεν είναι ήδη υλοποιημένος από την ομάδα προγραμματιστών του Rheem , ο χρήστης θα πρέπει να ακολουθήσει τρία στάδια για να φτάσει σε σημείο να μπορεί ο κώδικάς του να ικανοποιήσει την εφαρμογή του.

Ο χρήστης λοιπόν θα πρέπει πρωτίστως να δημιουργήσει το φυσικό τελεστή (physical operator) της εφαρμογής του επεκτείνοντας μία από τις κλάσεις του αντίστοιχου αφηρημένου τελεστή UnaryToUnaryOperator, BinaryToUnaryOperator κτλ. Στη συνέχεια θα πρέπει να υλοποιήσει τον αντίστοιχο τελεστή εκτέλεσης για καθένα από τα συστήματα επεξεργασίας που σκοπεύει να χρησιμοποιήσει και τέλος να δημιουργήσει τις απαραίτητες αντιστοιχίσεις μεταξύ του φυσικού και του εκτελεστικού τελεστή.

Τελικά λοιπόν ο χρήστης αφού επιβεβαιώσει πως υπάρχουν ήδη υλοποιημένοι οι τελεστές που σκοπεύει να χρησιμοποιήσει, καλείται να δημιουργήσει την εφαρμογή του εξαρχής χρησιμοποιώντας μόνο Rheem κώδικα.

Στο παράδειγμα του WordCount λοιπόν καλούμαστε να χρησιμοποιήσουμε τους Rheem τελεστές flatmap, filter, map, reduceByKey και collect μαζί με τις απαραίτητες ιδιότητες τους. Το Rheem μετά είναι υπεύθυνο να αντιστοιχίσει αυτούς τους φυσικούς τελεστές σε όσους αντίστοιχους τελεστές εκτέλεσης υπάρχουν, να καταστρώσει το πιο αποδοτικό πλάνο και εν τέλει να εκτελέσει την εφαρμογή.

Στο σημείο αυτό , αν και θα αναλυθεί συγκεντρωτικά παρακάτω, αξίζει να αναφερθεί πως εδώ βλέπουμε μία αδυναμία που παρουσιάζει το Rheem σε σχέση με το IReS. Για την δημιουργία του ίδιου απλού workflow στο IReS αρκεί να χρησιμοποιηθεί η υλοποίηση του τελεστή στην κάθε πλατφόρμα που στοχεύουμε να χρησιμοποιήσουμε, ενώ το Rheem απαιτεί τα προγράμματα να γράφονται στη δική του ‘γλώσσα’ ώστε να αναλάβει μετά τη μετάφραση του Rheem κώδικα στο σύστημα που θα δρομολογηθεί η εκτέλεση.

Εκτελώντας την εφαρμογή WordCount για το αρχείο των 800MB παρατηρούμε ότι το Rheem ενώ έχει τη δυνατότητα να διαλέξει ανάμεσα σε Spark και Java Streams, επιλέγει να τρέξει το workflow σε Java Streams σε αντίθεση με το IReS που επέλεξε το Hadoop. Παρακάτω φαίνεται το πλάνο που χτίστηκε για την εκτέλεση.

```
>>> ExecutionStage[T[JavaTextFileSource[Load file]]]:
>    JavaTextFileSource[Load file] => StreamChannel => JavaFlatMap[Split words]
>    JavaFlatMap[Split words] => StreamChannel => JavaFilter[Filter empty words]
>    JavaFilter[Filter empty words] => StreamChannel => JavaMap[To lower case, add        counter]
>    JavaMap[To lower case, add counter] => StreamChannel => JavaReduceBy[Add counters]
> Out JavaReduceBy[Add counters] => CollectionChannel
>>> ExecutionStage[T[JavaLocalCallbackSink[collect()]]]:
> In  CollectionChannel => JavaLocalCallbackSink[collect()]
```

Από τον παραπάνω γράφο διακρίνουμε πως το Rheem επέλεξε να τρέξει όλους τους τελεστές σε Java Streams. Επίσης σε αυτό το στάδιο είναι σημαντικό να αναφέρουμε ακόμα μια διαφορά σε σχέση με το IReS. Το Rheem λόγω του τρόπου που απαιτεί την υλοποίηση των εφαρμογών έχει τη δυνατότητα να μπορεί να διαλέξει μέσα στην ίδια την εφαρμογή να εκτελέσει κάποιους τελεστές σε ένα σύστημα και κάποιους άλλους σε διαφορετικό για να επιτύχει τη μέγιστη απόδοση. Το IReS από την άλλη στο ίδιο παράδειγμα χρησιμοποιεί τον υλοποιημένο τελεστή WordCount , όπου εσωτερικά στον κώδικά του φυσικά και υπάρχουν οι ίδιες λειτουργίες του flatmap, filter κτλ αλλά το IReS τον θεωρεί ενιαίο , που σημαίνει πως δεν μπορεί να εκτελέσει κάποιες λειτουργίες σε Java και κάποιες σε Hadoop.

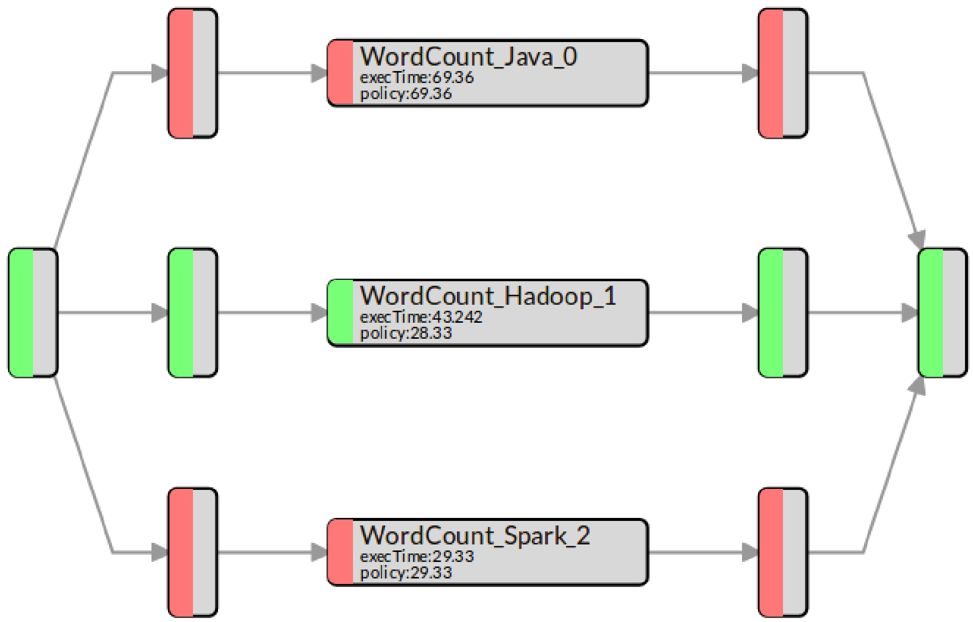
Παρακάτω βλέπουμε κάποιες μετρήσεις του Rheem στην έξοδο και τον τελικό χρόνο εκτέλεσης.

```
* Optimization                              - 0:00:06.643
  * Prepare                                 - 0:00:00.290
    * Prune&Isolate                         - 0:00:00.034
    * Transformations                       - 0:00:00.256
    * Sanity                                - 0:00:00.000
  * Cardinality&Load Estimation             - 0:00:06.083
    * Create OptimizationContext            - 0:00:00.021
    * Create CardinalityEstimationManager   - 0:00:00.002
    * Push Estimation                       - 0:00:06.060
      * Estimate source cardinalities       - 0:00:05.843
  * Create Initial Execution Plan           - 0:00:00.270
    * Enumerate                             - 0:00:00.186
      * Concatenation                       - 0:00:00.109
        * Channel Conversion                - 0:00:00.089
      * Prune                               - 0:00:00.040
    * Pick Best Plan                        - 0:00:00.015
    * Split Stages                          - 0:00:00.032
* Execution                                 - 0:00:45.444
  * Execution 0                             - 0:00:45.444
    * Execute                               - 0:00:45.437
* Post-processing                           - 0:00:00.107
  * Log measurements                        - 0:00:00.107
  * Release Resources                       - 0:00:00.000
```

Στην Εικόνα 9 παρακάτω βλέπουμε την αντίστοιχη έξοδο του IReS.
Παρατηρούμε πως το IReS επιλέγοντας να εκτελέσει το workflow στο Hadoop, μια επιλογή που το Rheem δεν προσφέρει, ολοκληρώνει την εκτέλεση κατά 2s γρηγορότερα. Επίσης θα πρέπει να συνυπολογίσουμε πως η παραπάνω έξοδος του Rheem γίνεται στο shell, ενώ το IReS γράφει την έξοδο στο HDFS δημιουργώντας τα απαραίτητα αρχεία σε αυτό με βάση τις επιλογές του χρήστη.

Tracking URL: http://master:8088/proxy/application_1509477022310_0047/

State: FINISHED SUCCEEDED



Εικόνα 10 : Μετά το πέρας της εκτέλεσης του WordCount workflow

3.4 Συμπεράσματα για το WordCount workflow

Είδαμε παραπάνω τη διαφορετική αντιμετώπιση των δύο συστημάτων όσον αφορά την εκτέλεση ενός workflow. Το IReS ουσιαστικά κατάφερε να υπερνικήσει το Rheem στην ανωτέρω διαδικασία κυρίως λόγω της επεκτασιμότητάς του. Αν και οι δύο εκτελέσεις που παρουσιάζουμε δεν μπορούν να γενικευθούν καθώς σίγουρα θα υπάρχουν αντίστοιχα workflows όπου το Rheem μπορεί να αποδώσει καλύτερα, δεν γίνεται να παραληφθεί το γεγονός πως το Rheem βασίζεται μόνο πάνω σε δύο πλατφόρμες. Αντίθετα το IReS υποστηρίζοντας ουσιαστικά οποιοδήποτε κατανεμημένο σύστημα επεξεργασίας φάνηκε να αντιμετωπίζει με περισσότερες εναλλακτικές το ανωτέρω παράδειγμα.

Φυσικά μια σημαντική ακόμα παρατήρηση αφορά την διαφορά φιλοσοφίας που φάνηκε να υπάρχει μεταξύ των δύο συστημάτων από την πρώτη κιόλας σύγκρισή τους. Η διαφορά αυτή εντοπίζεται στον τρόπο που μεταχειρίζονται τους τελεστές.

Το Rheem για την εκτέλεση του WordCount workflow ουσιαστικά χώρισε την εφαρμογή σε πολλούς διαφορετικούς αρχέγονους τελεστές (readTextFile.flatmap,filter,map,reduceByKey,collect) και για το σύνολό τους ξεκίνησε να ψάχνει το πιο αποδοτικό πλάνο εκτέλεσης. Μπορεί στο δικό μας παράδειγμα, λόγω της απλότητάς του, το Rheem να επέλεξε την εκτέλεση όλων των τελεστών με Java Streams, αλλά πιθανότατα σε διαφορετικές εφαρμογές μπορεί να κατανέμει κάποιους τελεστές στο Spark και κάποιους σε Java. Για παράδειγμα σε αλγορίθμους μηχανικής μάθησης όπου εμφανίζεται η ανάγκη για επανάληψη , που αφορά όμως μικρά σύνολα δεδομένων που χρησιμεύουν κυρίως στην εκπαίδευση των μοντέλων, το Rheem εκτελεί αυτούς τους τελεστές σε Java Streams ενώ τους υπόλοιπους σε Spark, με αποτέλεσμα να βελτιώνει αισθητά την απόδοση της εφαρμογής.

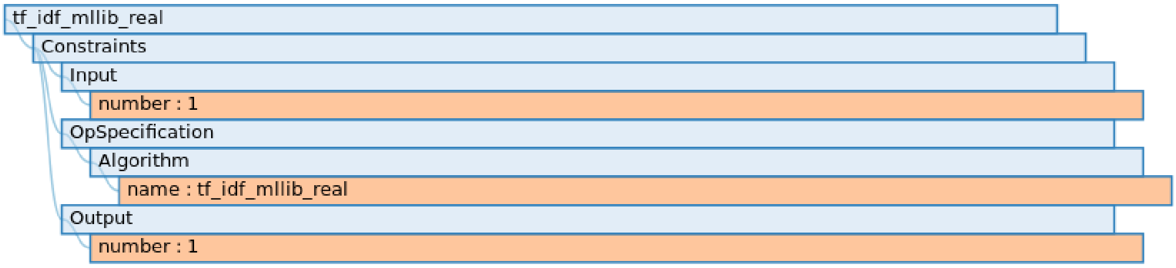
Το IReS αντίθετα στο παραπάνω παράδειγμα θεωρεί την υλοποίηση του WordCount σε Hadoop σαν έναν τελεστή. Αυτός είναι και ο λόγος που για την επιλογή του καλύτερου πλάνου ανάλογα με το μέγεθος του αρχείου που του δίνουμε στην είσοδο , αποφασίζει πολύ γρηγορότερα από το Rheem για τον ποιον τελεστή θα χρησιμοποιήσει. Επίσης αυτός ο χειρισμός κάνει το σύστημα πολύ πιο φιλικό στο χρήστη, καθώς δεν του επιβάλλει να έχει περεταίρω γνώσεις για το ίδιο το σύστημα. Το μόνο που απαιτείται από το IReS είναι να μπορεί να δηλώσει ο χρήστης τα αρχεία που θα χρησιμοποιήσει και τις εξαρτήσεις τους από τα αντίστοιχα συστήματα επεξεργασίας.

3.5 Δημιουργία TF-IDF – Kmeans Workflow

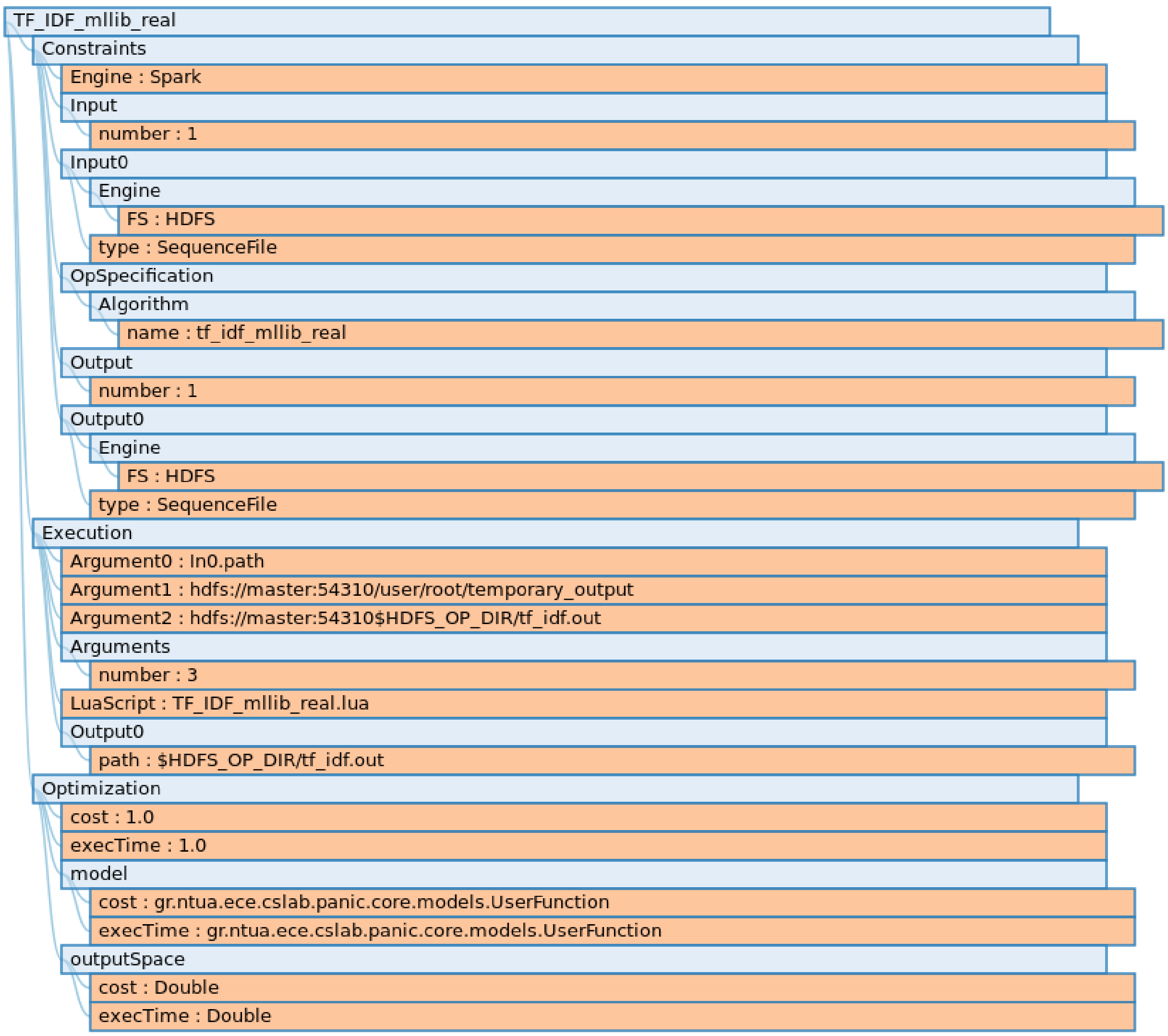
Σε αυτήν την ενότητα θα περιγράψουμε την διαδικασία που ακολουθήθηκε για τη δημιουργία ενός workflow όπου θα διαβάζει αρχεία κειμένου από κάποιον φάκελο στο HDFS και θα εφαρμόζει τον αλγόριθμο TF-IDF (term frequency – inverse document frequency) και μετέπειτα στα αποτελέσματα αυτά θα εφαρμόζει τον clustering αλγόριθμο K-means.

Ακολουθούμε παρόμοια διαδικασία με το προηγούμενο workflow για να υλοποιήσουμε τον τελεστή TF-IDF. Αφού φτιάξουμε και αποθηκεύσουμε στο IRes τον abstractOperator TF-IDF χρησιμοποιούμε την υλοποίηση του Spark και συγκεκριμένα της mllib βιβλιοθήκης του για να δημιουργήσουμε τον πρώτο materialized τελεστή. Η συγκεκριμένη υλοποίηση δεν παρέχει αντιστροφή της διαδικασίας , που σημαίνει πως δεν μπορούμε να δούμε τις τιμές των συχνοτήτων για τις λέξεις που υπολογίστηκαν. Γι'αυτό χρειάστηκε να την επεκτείνουμε δημιουργώντας ειδικό λεξικό ώστε μετά το πέρας της εκτέλεσης να παρέχεται η δυνατότητα της αντιστροφής. Στις Εικόνες 10 και 11 βλέπουμε τη δεντρική αναπαράσταση του αφηρημένου τελεστή TF-IDF και του υλοποιημένου τελεστή σε Spark αντίστοιχα.

Abstract Operator: tf_idf_mllib_real



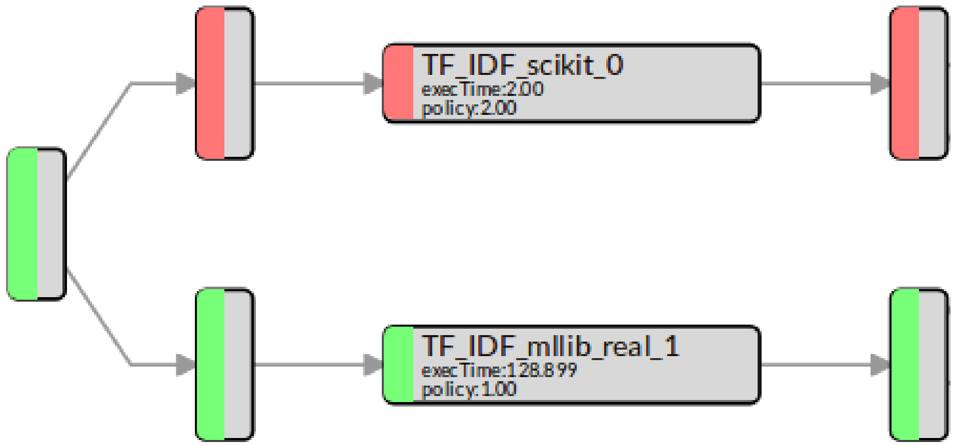
Εικόνα 11 : Αφηρημένος τελεστής tf-idf



Εικόνα 12 : Υλοποιημένος τελεστής tf-idf

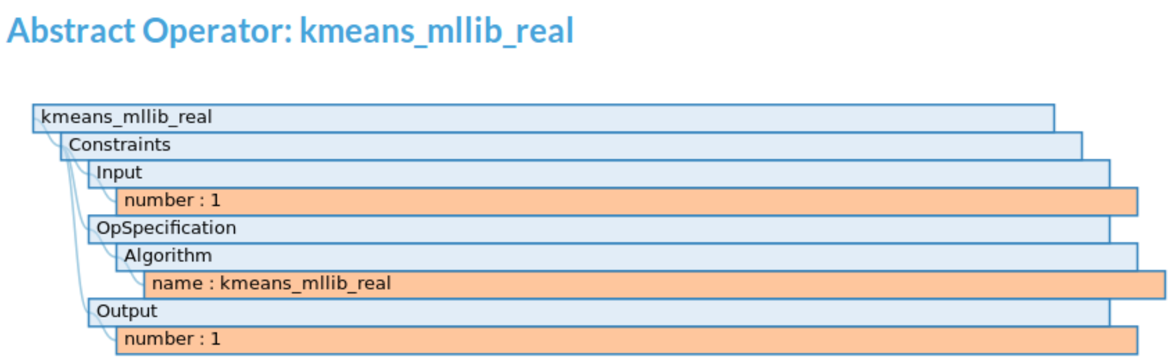
Ο δεύτερος materialized τελεστής που θα χρησιμοποιηθεί είναι η υλοποίηση του αλγορίθμου με τη βασική βιβλιοθήκη της python scikit. Ουσιαστικά αυτή η υλοποίηση αντιστοιχίζεται με την υλοποίηση στο προηγούμενο παράδειγμα της java.

Στην Εικόνα 12 φαίνεται το πρώτο στάδιο δημιουργίας του workflow.



Εικόνα 13 : Πρώτο στάδιο tf-idf – Kmeans workflow

Οδηγούμαστε τώρα στο επόμενο στάδιο που είναι οι δημιουργία των τελεστών K-Means. Η πρώτη υλοποίηση σε αντιστοιχία με το TF-IDF προέρχεται από την βιβλιοθήκη mllib του Spark και η δεύτερη από τη scikit βιβλιοθήκη της python. Στην Εικόνα 13 βλέπουμε την αφηρημένη υλοποίηση του K-Means



Εικόνα 14 : Αφηρημένος τελεστής KMeans

Σκοπός της παρουσίασης του συγκεκριμένου workflow είναι η ανάδειξη κάποιων πλεονεκτημάτων του IReS. Βλέπουμε ότι μέχρι το σημείο αυτό ουσιαστικά έχουμε δημιουργήσει δύο ανεξάρτητα workflows. Πρόκειται για δύο πολύ διαδεδομένους αλγορίθμους στην ανάλυση δεδομένων μεγάλου όγκου και αρκετές φορές είναι σύνηθες να απαιτείται ο αλγόριθμος K-Means να πρέπει να τρέξει με είσοδο τα δεδομένα της εξόδου του αλγορίθμου TF-IDF. Για να το πετύχουμε λοιπόν αυτό το μόνο που χρειάζεται από τη μεριά του χρήστη είναι η τροποποίηση του γράφου της ροής εργασιών. Έτσι λοιπόν στο IReS αλλάζουμε τον γράφο της εκτέλεσης στον παρακάτω :

input_file,tf_idf

tfidf,d1

d1,K-Means

K-Means,d2

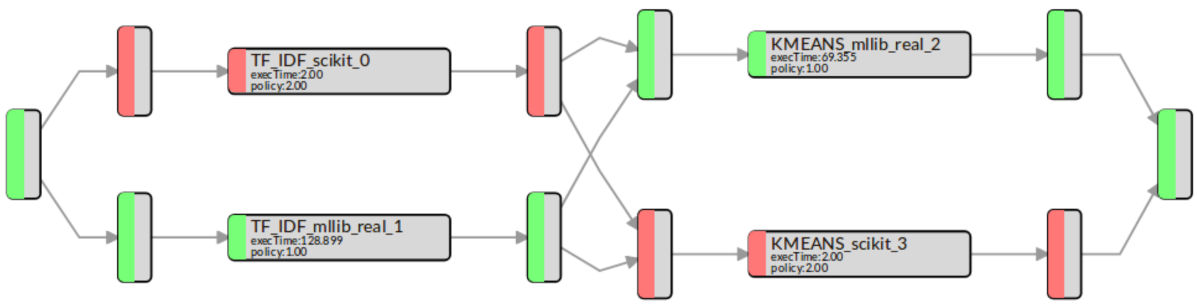
d2,$$target

Ουσιαστικά με αυτόν τον τρόπο ενημερώνουμε το IReS πως σε αντίθεση με πριν το προσωρινό αρχείο d1, θα γράψει στο HDFS την έξοδο του αλγορίθμου TF-IDF και ταυτόχρονα θα είναι και το αρχείο εισόδου του αλγορίθμου K-Means. Έτσι λοιπόν ολοκληρώθηκε το workflow που θέλαμε, με την εκτέλεσή του να φαίνεται παρακάτω στην Εικόνα 14 . Εδώ αξίζει να σημειωθεί πως δεν χρειαζόμαστε ενδιάμεσους τελεστές για την τροποποίηση της εξόδου ώστε να μπορεί να εφαρμοσθεί στην είσοδο του επόμενου τελεστή. Ο λόγος είναι πως έχουμε φροντίσει μέσω του κώδικα η κάθε έξοδος που προκύπτει να ικανοποιεί τις απαιτήσεις εισόδου του επόμενου εν σειρά τελεστή.

Αυτή η τεχνική που χρησιμοποιήθηκε μειονεκτεί ως προς την επεκτασιμότητα. Στην περίπτωση για παράδειγμα ενός ακόμα τελεστή που υλοποιεί το K-Means λ.χ. σε Hadoop και θέλει διαφορετική μορφή στην είσοδο, θα πρέπει να αλλαχθεί ο κώδικας του TF-IDF αλγορίθμου έτσι ώστε να μπορεί να εναρμονίζεται με τις απαιτήσεις του κάθε επόμενου τελεστή. Μια διαφορετική προσέγγιση θα μπορούσε να είναι, να κρατήσουμε την έξοδο του TF-IDF ως έχει (SparseVector) και να δημιουργήσουμε ενδιάμεσους τελεστές που θα αναλαμβάνουν να την μεταφράζουν στην επιθυμητή είσοδο.

Tracking URL: http://master:8088/proxy/application_1509550310571_0001/

State: FINISHED SUCCEEDED



Εικόνα 15 : Εκτέλεση του tf-idf – Kmeans Workflow

Το Rheem δεν προσφέρει υλοποιημένο τον τελεστή TF-IDF αλλά μόνο το K-Means. Η δυσκολία που παρουσιάστηκε τόσο στην υλοποίηση του τελεστή αλλά και στην ενοποίηση του με τον υπάρχων K-Means είναι από τα μειονεκτήματα της χρήσης του Rheem. Ενώ στο IReS η εκτέλεση ενός workflow εξαρτάται αποκλειστικά από την ευχέρεια του χρήστη στα διάφορα συστήματα της αγοράς, όπως το Hadoop ή το Spark , στο Rheem είναι απαραίτητη η δημιουργία κάθε workflow στη δική του γλώσσα. Λόγω του περιορισμένου αριθμού τελεστών που παρέχει μέχρι στιγμής υλοποιήσαμε μόνο το K-Means σε αρχείο παρόμοιου μεγέθους με το workflow του IReS .

Ένα ακόμα μειονέκτημα που συναντήσαμε σε αυτήν την υλοποίηση είναι πως ο συγκεκριμένος αλγόριθμος δέχεται ως είσοδο μόνο csv αρχεία, που αποτελούν τη σύνηθες είσοδο σε τέτοιου είδους αλγορίθμους αλλά όχι και τη μοναδική.

Παρακάτω παρουσιάζουμε μέρος από το πλάνο που επέλεξε το Rheem και βλέπουμε πως λόγω του ότι το αρχείο της εισόδου που δώθηκε είναι αρκετά μεγάλο το Rheem αποφασίζει να εκτελέσει ένα μέρος του σε Spark κι ένα μέρος του σε Java.

```
>>> ExecutionStage[T[SparkTextFileSource[Read file]]]:
>     SparkTextFileSource[Read file] => RddChannel => SparkCollect[convert output@SparkTextFileSource[Read file]]
> Out SparkCollect[convert output@SparkTextFileSource[Read file]] => CollectionChannel
>>> ExecutionStage[T[JavaRepeat[Loop]]]:
```

> In CollectionChannel => JavaRepeat[Loop]

> In CollectionChannel => JavaRepeat[Loop]

> Out JavaRepeat[Loop] => StreamChannel

> Out JavaRepeat[Loop] => StreamChannel

>>> ExecutionStage[T[JavaMap[Create points]]]:

> In CollectionChannel => JavaMap[Create points]

>   JavaMap[Create points] => StreamChannel => JavaCollect[convert out@JavaMap[Create points]]

>>> ExecutionStage[T[SparkCollectionSource[convert out@JavaMap[Average points]]]]:

.

.

.

> In CollectionChannel => SparkCollectionSource[convert out@JavaMap[Average points]]

> Out SparkCollectionSource[convert out@JavaMap[Average points]] => RddChannel

>>> ExecutionStage[T[SparkCollectionSource[convert out@JavaGlobalReduce[Count centroids (b)]]]]:

> In CollectionChannel => SparkCollectionSource[convert out@JavaGlobalReduce[Count centroids (b)]]

>   SparkCollectionSource[convert out@JavaGlobalReduce[Count centroids (b)]] => RddChannel => SparkFlatMap[Resurrect centroids]

>   SparkFlatMap[Resurrect centroids] => RddChannel => SparkUnionAll[New+resurrected centroids]

>   SparkUnionAll[New+resurrected centroids] => RddChannel => SparkCollect[convert out@SparkUnionAll[New+resurrected centroids]]

> Out SparkCollect[convert out@SparkUnionAll[New+resurrected centroids]] => CollectionChanne

Συγκεκριμένα βλέπουμε πως αρχικά το Rheem διαβάζει την είσοδο χρησιμοποιώντας το Spark και συνεχίζει τον αλγόριθμο αλλάζοντας πλατφόρμα για το κομμάτι του loop του αλγορίθμου. Μετά από το πέρας της επανάληψης και κάποιων ακόμα υπολογισμών , επιστρέφει στο Spark όπου χρησιμοποιεί τον τελεστή UnionAll και Collect για τα αποτελέσματα.

4 Συνολικά Συμπεράσματα

4.1 Planning

Όπως αποδείχθηκε η διαδικασία της εκτίμησης του πιο αποδοτικού πλάνου που θα ακολουθηθεί είναι εξέχουσας σημασίας και στα δύο συστήματα. Πολλές φορές όπως είδαμε και στα παραπάνω παραδείγματα η ίδια διαδικασία μπορεί να χρειαστεί να εκτελεστεί με διαφορετικές παραμέτρους κάτω από διαφορετικές συνθήκες. Έτσι λοιπόν θα ήταν παράλογο να ακολουθούσαμε τυφλά πάντα την ίδια υλοποίηση. Συχνά για μικρότερα, σχετικά, σύνολα δεδομένων είναι προς το συμφέρον του χρήστη το σύστημα να επιλέγει την μη κατανεμημένη υλοποίηση με σκοπό την καλύτερη επίδοση όσον αφορά το χρόνο εκτέλεσης αλλά και την προφύλαξη των πόρων των υπολογιστικών συστοιχιών, που ίσως χρειάζονται για την περάτωση παράλληλα άλλων εφαρμογών.

Το IReS για την δημιουργία μοντέλων εκτίμησης χρησιμοποιεί το ganglia την ώρα της εκτέλεσης του τελεστή και συγκεντρώνει διάφορα δεδομένα εκ των οποίων είναι ο χρόνος εκτέλεσης, η μνήμη που χρησιμοποιήθηκε, η χρήση του επεξεργαστή κτλ. Για τη δημιουργία του μοντέλου απαιτούνται δύο εκτελέσεις του ίδιου τελεστή. Μετά από αυτές τις δύο εκπαιδεύσεις το IReS κατά την εκκίνηση του server ζητάει τα δεδομένα από τη mongo DB που έχουν αποθηκευτεί και μέσω του panic [10] υλοποιεί το μοντέλο. Το Rheem αντίστοιχα κατά την εκτέλεση της εφαρμογής φροντίζει να κρατάει τις δικές του μετρήσεις για παρόμοιες παραμέτρους με το IReS σε json αρχεία. Η μόνη μικρή διαφορά είναι πως το Rheem έρχεται με κάποιους έτοιμους υπολογισμούς για τους τελεστές που είναι ήδη υλοποιημένοι σε αυτό και παράλληλα ενημερώνεται όπως και to IReS μετά από κάθε εκτέλεση.

Η διαφορά όμως που παρουσιάζεται σε αυτό το στάδιο έχει να κάνει με τον διαφορετικό τρόπο που αντιμετωπίζουν το IReS και το Rheem την έννοια των τελεστών . Το IReS όπως είδαμε έχει τη δυνατότητα να αντιμετωπίζει τους τελεστές πιο καθολικά απ'ότι το Rheem με αποτέλεσμα οι μετρήσεις που θα συλλέγει να μην αφορούν αρχέγονους τύπους όπως το flatmap του Rheem λόγου χάρη. Η διαφορά αυτή πρακτικά σημαίνει πως το Rheem μπορεί να παράγει αρκετά περισσότερα πλάνα εκτέλεσης από το IReS , αλλά παρ' όλ' αυτά στην εξίσωση πρέπει να μπει και το κόστος μετακίνησης των δεδομένων από το ένα σύστημα αποθήκευσης στο άλλο. Συνήθως, επειδή μιλάμε για μεγάλο όγκο δεδομένων, η μετακίνηση αυτή συμφέρει να γίνει μία έως πολύ λίγες φορές κατά την εκτέλεση ενός wokflow οπότε είναι κάτι που και το IReS μπορεί να το προβλέψει και να ελιχθεί.

4.2 Execution

4.2.1 Platform Independence

Μία ακόμη διαφορά μεταξύ των δύο συστημάτων που συγκρίναμε στην παρούσα διπλωματική είναι και το κατά πόσο είναι εύκολο να προσθέσεις μια καινούρια μηχανή. Το IReS είναι ένα αγνωστικιστικό σύστημα όσον αφορά τη μηχανή εκτέλεσης. Μέσω των script που παρέχει ο χρήστης ουσιαστικά μπορεί να εκτελέσει το workflow σε οποιοδήποτε σύστημα επεξεργασίας. Είδαμε παραπάνω στο πρώτο παράδειγμα πως η χρησιμοποίηση του Hadoop ήταν καθοριστικός παράγοντας και έκανε τη διαφορά σε σχέση με το Rheem που παρέχει μόνο υλοποιήσεις σε Spark και σε Java. Σύμφωνα με τους προγραμματιστές του Rheem στο μέλλον θα υπάρξει και υποστήριξη του Hadoop. Ουσιαστικά όμως αυτό δεν λύνει το πρόβλημα για το Rheem καθώς θα συνεχίσει να είναι δεμένο με τα συστήματα επεξεργασίας που υποστηρίζει. Αυτή η εξάρτηση που παρουσιάζει είναι πιθανό να αποθαρρύνει τους χρήστες από τη χρήση του καθώς απαιτεί για κάθε καινούργια πλατφόρμα που θα μπορεί να χρησιμοποιήσει, την δημιουργία τελεστών τόσο εκτέλεσης όσο και φυσικών καθώς και τη σύνδεση μεταξύ τους.

4.2.2 Fault tolerance

Σε αυτού του είδους τα συστήματα είναι πολύ σημαντικό να υπάρχει πρόβλεψη για την αστοχία των μηχανών επεξεργασίας. Είναι πολύ σύνηθες να συμβαίνουν αστοχίες όσον αφορά τη ζητούμενη μνήμη κάθε προγράμματος ή ακόμα και αστοχίες που έχουν να κάνουν με κάποιο εσωτερικό πρόβλημα της έκδοσης που παρέχεται και ενδέχεται να διορθωθεί σε κάποια επόμενη ενημέρωση. Γεγονός είναι πως αυτά τα προβλήματα τις περισσότερες φορές συμβαίνουν ανεξάρτητα από τον κώδικα της εκάστοτε εφαρμογής και κατά το μεγαλύτερο ποσοστό τους λύνονται με την επαναχρονοδρομολόγηση της εφαρμογής.

Το IReS για να αποφύγει τέτοιου είδους καταστάσεις δρα σε δύο διαφορετικά σημεία. Αρχικά πριν από την εκτέλεση ελέγχει τη διαθεσιμότητα των συστημάτων επεξεργασίας που μπορούν να ικανοποιήσουν τις απαιτήσεις του χρήστη και αποβάλλει από το εκτελεστικό του πλάνο αυτά που δεν είναι σε υγιή κατάσταση λειτουργιάς. Κατόπιν σε πραγματικό χρόνο ελέγχει για τυχόν αστοχίες και αν εντοπίσει κάποια επαναπρογραμματίζει την εργασία προς εκτέλεση. Αξίζει να σημειωθεί εδώ πως όσα δεδομένα έχουν προκύψει μέχρι εκείνη τη στιγμή δεν χάνονται αλλά αντίθετα χρησιμοποιούνται στην επαναδρομολόγηση του workflow για να μειώσουν το χρόνο εκτέλεσης.

Το Rheem αντίθετα δεν διαθέτει κανέναν τέτοιο μηχανισμό. Στο παραπάνω παράδειγμα του K-Means workflow συναντήσαμε και τα δύο παραπάνω προβλήματα που το IReS είναι φτιαγμένο να εντοπίζει. Αρχικά απενεργοποιήσαμε το Spark και δοκιμάσαμε να τρέξουμε το K-Means και με τις δύο παραμέτρους εκτέλεσης που αφορούν τις πλατφόρμες που υποστηρίζει (java, spark).

Στην αντίστοιχη περίπτωση το IReS θα δρομολογούσε την εφαρμογή να εκτελεστεί στη java και θα χρεωνόταν τον παραπάνω χρόνο εκτέλεσης. Αντίθετα το Rheem δεν φάνηκε να προβαίνει σε κάποιον έλεγχο και στο πλάνο που δημιούργησε προσπάθησε να διαβάσει το αρχείο , όπως και πριν μέσω του Spark. Προφανώς το αποτέλεσμα ήταν να τερματιστεί η εφαρμογή και να εμφανίσει τα πιθανά λάθη που συνέβησαν κατά την εκτέλεση.

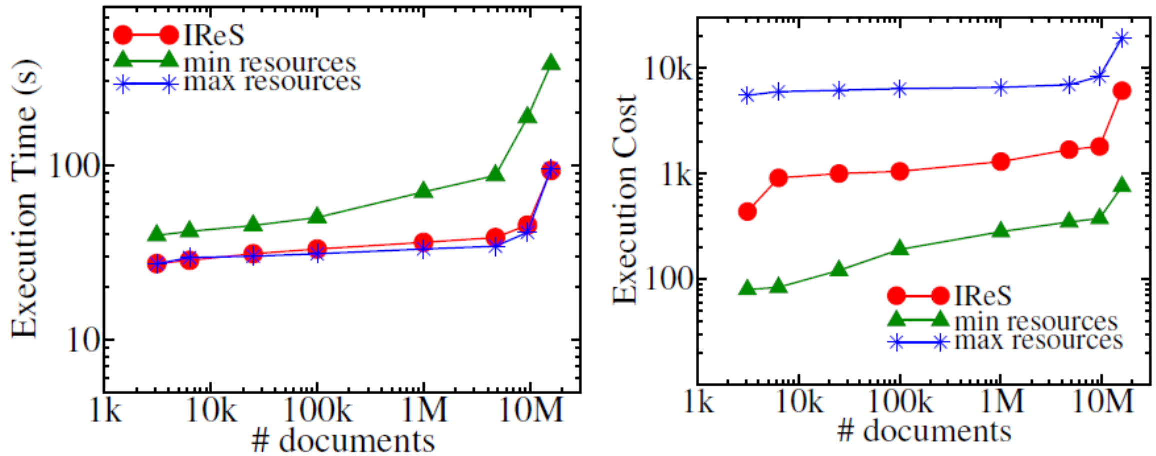
Η δεύτερη πιθανή αστοχία συνέβη σε μια απόπειρα μας να τρέξουμε την εφαρμογή του WordCount στο Spark δίνοντας για είσοδο ένα πολύ μεγάλο αρχείο κειμένου μεγέθους 8GB. To Rheem φαίνεται σε αυτήν την περίπτωση να ζήτησε πολύ παραπάνω μνήμη από τους Spark-Executors αδιαφορώντας για το άνω φράγμα επιτρεπόμενης μνήμης που επιτρέπεται να ζητηθεί. Σε αυτήν την περίπτωση το αποτέλεσμα ήταν να απενεργοποιηθούν κάποιοι κόμβοι του Spark και να χρειαστεί ολική επανεκκίνηση.

Όπως διαπιστώνουμε λοιπόν από τα παραπάνω το Rheem εν αντιθέσει με το IReS δεν παρέχει μηχανισμούς προστασίας σε περίπτωση αστοχιών.

4.3 Χρησιμοποίηση Υπολογιστικών Πόρων

Το εργαλείο δημιουργίας του πλάνου εκτέλεσης του IReS (planner) εκτός από το να επιλέγει την αποδοτικότερη μηχανή επεξεργασίας για τον κάθε τελεστή, έχει επίσης τη δυνατότητα να διαλέγει τους ελάχιστους υπολογιστικούς πόρους που θα χρειαστεί για να εκτελεστεί ο κάθε τελεστής. Οι προβλέψεις αυτές βασίζονται στη χρήση του γενετικού αλγορίθμου NSGA-II [7] πάνω στις παρεχόμενες μετρήσεις που αποθηκεύονται στη βάση του για την κάθε εκτέλεση των τελεστών. Η λειτουργία αυτή μπορεί να φανεί πολλή χρήσιμη ειδικά σε περιπτώσεις που μοιράζονται την ίδια υπολογιστική συστοιχία ομάδες ατόμων που χρονοδρομολογούν διαφορετικές εφαρμογές και ο διαμοιρασμός της μνήμης είναι πολύ σημαντικός.

Ένας ακόμη λόγος που πρέπει να τονιστεί είναι πως η παραπάνω λειτουργία αφορά και το πραγματικό κόστος χρήσης μιας υπολογιστικής συστοιχίας όπως αναφέρεται εδώ [19]. Υιοθετούμε την έννοια του κόστους από εδώ [9] $κόστος = \#VM \times cores / VM \times MM / VM \times t$, όπου VM = αριθμός των εικονικών μηχανών , $cores / VM$ = ο αριθμός των πυρήνων ανά εικονική μηχανή , MM / VM = η κύρια μνήμη ανά εικονική μηχανή σε GB και t ο χρόνος εκτέλεσης. Είναι φυσικό όταν εκτελούμε μια εργασία σε ένα κατανεμημένο περιβάλλον ο χρόνος εκτέλεσης να μειώνεται όσο προσφέρουμε παραπάνω υπολογιστικούς πόρους. Ταυτόχρονα όμως όπως διακρίνουμε εύκολα από την παραπάνω εξίσωση , περισσότεροι πόροι σημαίνει και μεγαλύτερο κόστος εκτέλεσης. Το IReS με τη λειτουργία αυτή λοιπόν καταφέρνει να πετύχει απόδοση που ισοδυναμεί με τη μέγιστη παραχώρηση πόρων και ταυτόχρονα κόστος που κυμαίνεται ανάμεσα στο ελάχιστο δυνατόν και το μέγιστο. Αυτό φαίνεται σε εκτελέσεις που έγιναν με τον tf-idf Spark (mllib) τελεστή σε υπολογιστική συστοιχία συνολικής μνήμης 54 GB και 32 πυρήνων όπως αναφέρεται σε αυτήν την πηγή [9] . Στην Εικόνα 16 παρακάτω μπορούμε να δούμε και γραφικά τα αποτελέσματα αυτής της εκτέλεσης.



Εικόνα 16 : Γραφική απεικόνιση εξοικονόμησης πόρων – επίδοσης του IReS

4.4 Τελεστές

Ενδιαφέρον παρουσιάζει επίσης η διαφορετική προσέγγιση των δύο συστημάτων όσον αφορά το τι σημαίνουν για αυτά οι τελεστές. Το Rheem αλλά και το IReS συμπεριφέρονται στους τελεστές σαν μαύρα-κουτιά. Αυτό ουσιαστικά σημαίνει πως ο χρήστης δεν θα πρέπει να ενδιαφέρεται για την υλοποίηση των τελεστών και από τη στιγμή που αυτή υφίσταται θα μπορεί να εκτελέσει τις διάφορες εργασίες που επιθυμεί. Το Rheem αρχικά θα μπορούσαμε να πούμε ότι έχει χαμηλού επιπέδου τελεστές σε σχέση με το IReS που μπορεί να χρησιμοποιήσει και τελεστές υψηλού επιπέδου. Συγκεκριμένα το Rheem φαίνεται να είναι δεμένο στο να μπορεί να μεταφράσει αρχέγονους τελεστές όπως το map , reduceBy κτλ στις διάφορες μηχανές προς εκτέλεση. Αυτό όπως αναφέραμε και παραπάνω μπορεί να προκαλέσει διάφορα προβλήματα που εστιάζονται κυρίως στην επεκτασιμότητα της πλατφόρμας αλλά και στην ευκολία της χρήσης.

Το IReS από την άλλη μεριά, θα μπορούσαμε να πούμε πως μπορεί να χειριστεί τελεστές και χαμηλού αλλά και υψηλού επιπέδου. Με τον όρο τελεστές υψηλού επιπέδου εννοούμε ολόκληρους αλγορίθμους που αντιμετωπίζονται από το IReS ως ένας τελεστής. Στα παραδείγματα που παρουσιάσαμε στο προηγούμενο κεφάλαιο είδαμε πως το IRes έχει τη δυνατότητα να εκλάβει ως υλοποιημένο τελεστή την υλοποίηση του WordCount σε Spark λ.χ. ενώ το Rheem για την ίδια υλοποίηση χρησιμοποιεί τους έξι αρχέγονους τελεστές που το υλοποιούν στην πραγματικότητα.

Αυτή φαίνεται να είναι και η κυρίαρχη διαφορά ανάμεσα στα δύο συστήματα που εξετάζουμε στην παρούσα διπλωματική εργασία. Ο λόγος που το υποστηρίζουμε αυτό είναι πως όλα τα υπόλοιπα κομμάτια που απαρτίζουν τις δύο αυτές εφαρμογές είναι άρρηκτα συνδεδεμένα σε αυτή τη διαφορά φιλοσοφίας.

Η προσκόλληση του Rheem σε αυτόν τον τρόπο μετάφρασης των εφαρμογών φαίνεται να παρουσιάζει μειονεκτήματα αλλά ίσως και κάποια πλεονεκτήματα σε συγκεκριμένες περιπτώσεις σε σχέση με το IReS.

Το μεγαλύτερο μειονέκτημα είναι σίγουρα ο περιορισμός της επεκτασιμότητας που υφίσταται από τη στιγμή που κάθε τελεστής για να μπορέσει να επιλεχθεί από το βέλτιστο πλάνο εκτέλεσης θα πρέπει να αντιστοιχίζεται σε έναν φυσικό τελεστή ή αλλιώς Rheem τελεστή. Αυτό πρακτικά σημαίνει πως για κάθε καινούρια μηχανή επεξεργασίας που μπορεί να χρειαστεί να ενσωματωθεί, ο προγραμματιστής είναι υποχρεωμένος να αντιστοιχίσει όλους τους διαθέσιμους φυσικούς τελεστές στους τελεστές εκτέλεσης της εκάστοτε μηχανής.

Επίσης ο χρήστης περιορίζεται αρκετά, αφού υπάρχουν δύο σενάρια που μπορεί να συμβούν. Στο πρώτο σενάριο της ολοκληρωτικής έλλειψης ενός τελεστή , ο χρήστης θα πρέπει να είναι σε θέση να δημιουργήσει εξαρχής την εφαρμογή του επεκτείνοντας τους αρχέγονους τελεστές του Rheem. Το σενάριο αυτό απαιτεί βαθιά κατανόηση τόσο του αλγορίθμου της εκάστοτε εφαρμογής όσο και της αρχιτεκτονικής του Rheem. Η διαδικασία που είναι αναγκασμένος να ακολουθήσει ξεφεύγει από τα όρια του χρήστη και έγγυται στο κομμάτι των προγραμματιστών. Το δεύτερο πιο ήπιο σενάριο είναι να υπάρχει ο τελεστής αλλά ο χρήστης να επιθυμεί την επέκτασή του για διαφορετική λειτουργία. Και σε αυτήν την περίπτωση ο χρήστης θα πρέπει να είναι πολύ εξοικειωμένος με τα τρία επίπεδα του Rheem καθώς και με τον αλγόριθμο εκτέλεσης.

Το πλεονέκτημα που παρουσιάζει αυτή η προσέγγιση είναι κυρίως σε πολύπλοκα workflows που η μεταφορά των δεδομένων για την αλλαγή μηχανής επεξεργασίας είναι η πιο αποδοτική. Το Rheem λόγω του ότι διαχωρίζει εις βάθος τους τελεστές μπορεί να παράγει το βέλτιστο πλάνο ψάχνοντας ανάμεσα σε όλους τους συνδυασμούς εκτέλεσης και κατ' επέκταση μεταφοράς των δεδομένων.

Το IReS μπορεί να αντιμετωπίσει τον κάθε τελεστή ανάλογα με το πως θα τον ορίσει ο χρήστης. Αυτό πρακτικά σημαίνει πως θα μπορούσε να χτισθεί μια εφαρμογή με την ίδια λογική του Rheem όσον αφορά τους τελεστές, αλλά ταυτόχρονα παρέχει και μια πιο υψηλού επιπέδου προσέγγιση η οποία είναι σίγουρα πιο κοντά στην έννοια του χρήστη και της εκτέλεσης αναλυτικών εργασιών.

4.5 Γραφικό Περιβάλλον

Τελευταίο αλλά εξίσου σημαντικό είναι να αναφέρουμε πως στην πορεία που ακολουθήθηκε για την κατανόηση και την εξοικείωση με το IReS έπαιξε καταλυτικό ρόλο το γραφικό του περιβάλλον. Σε αυτό μπορεί ο χρήστης να επαληθεύει την διαδικασία της δημιουργίας των εκάστοτε workflows. Ουσιαστικά κάθε κομμάτι που οδηγεί στην ολοκληρωτική δημιουργία τους αποτυπώνεται στην οθόνη τόσο με δεντρικές απεικονίσεις, όπου αφορά τις ιδιότητες των τελεστών που παρέχει ο χρήστης, όσο και με κατευθυνόμενους γράφους, που απεικονίζουν τις αποφάσεις που έχουν ληφθεί βάσει του βέλτιστου πλάνου αλλά και την ίδια την πορεία της εκτέλεσής του σε πραγματικό χρόνο. Συγκεκριμένα στις Εικόνα 17 που ακολουθεί μπορούμε να δούμε τα σύνολα δεδομένων που έχουν δημιουργηθεί, τους αφηρημένους τελεστές και κάποιους από τους υλοποιημένους τελεστές αντίστοιχα.

Datasets

- asapServerLog
- hdfs_file
- km_file_real
- mytfidf_file
- pagerank_file
- test_file
- textData
- tf_file
- tf_file2
- tf_file_real
- word_tf_file
- wordc2_file
- wordc_big1
- wordc_file
- wordc_med1
- wordc_small2

Abstract Operators

- HelloWorld
- HelloWorld1
- HelloWorld2
- HelloWorld3
- kmeans
- kmeans_mllib_real
- kmeans_scikit
- LineCount
- PageRank
- TestOp
- TestOp1
- TestOp2
- tf_idf_mllib
- tf_idf_mllib2
- tf_idf_mllib_real
- tf_idf_scikit
- tfidf
- tfidf_cilk
- WordCount

Operators

H

- HelloWorld
- HelloWorld1_1
- HelloWorld1_2
- HelloWorld2_1
- HelloWorld2_2
- HelloWorld2_3
- HelloWorld3

K

- kmeans_cilk
- KMEANS_mllib_real
- KMEANS_scikit

L

- LineCount

Εικόνα 17 : Κατάλογος συνόλου δεδομένων , αφηρημένων τελεστών και υλοποιημένων τελεστών

5 Μελλοντικά Σχέδια - Επεκτάσεις

Πρωταρχικός στόχος για το μέλλον με σκοπό τη βαθύτερη κατανόηση αλλά και εξοικείωση με το Rheem είναι η δημιουργία τελεστών που δεν υποστηρίζονται αυτή τη στιγμή από τους προγραμματιστές του. Είναι μια διαδικασία που από τη φύση της δυσκολίας της μπορεί να προσφέρει μεγαλύτερη άνεση τόσο ως προς το χειρισμό του συστήματος του ίδιου όσο και ως προς το ευρύτερο πεδίο της ανάλυσης δεδομένων μεγάλου όγκου. Επίσης όσον αφορά το IReS στόχος είναι η δημιουργία περισσότερων τελεστών και μετέπειτα workflows στον τομέα της επεξεργασίας των γράφων, αφού στην παρούσα διπλωματική ασχοληθήκαμε κυρίως με workflows ανάλυσης κειμένου.

6 Επίλογος

Αντιλαμβανόμενοι τη σημασία της πληροφορίας και τη δυσκολία της ανάκτησής της λόγω της ραγδαίας αύξησης των δεδομένων στις μέρες μας , στην εργασία αυτή παρουσιάσαμε δύο συστήματα που αγωνίζονται για το επόμενο βήμα στην ανάλυση των δεδομένων μεγάλου όγκου.

Σίγουρα η λύση για την αντιμετώπιση του προβλήματος των πολλών διαφορετικών συστημάτων επεξεργασίας που υπάρχουν στην αγορά, βρίσκεται στην ενοποίησή τους κάτω από ένα κοινό πρίσμα.

Κλείνοντας, αξίζει να σημειωθεί πως τα δύο αυτά συστήματα που κληθήκαμε να συγκρίνουμε, προφανώς και έχουν αρκετές παραπάνω δυνατότητες, που δεν ήταν δυνατόν να τις φανερώσουν κάτω από τις καλά ορισμένες συνθήκες τις οποίες τα προσεγγίσαμε. Αυτό συνεπάγεται πως για τα συμπεράσματα που οδηγηθήκαμε , βασιστήκαμε στα πειράματα που πραγματοποιήσαμε στα πλαίσια της εκπόνησης αυτής της διπλωματικής εργασίας και στις πληροφορίες που μας παρείχαν οι αντίστοιχοι δημιουργοί – προγραμματιστές τους.

7 Αναφορές

[1] https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes- of-data-created-every-day-how-does-cpg-retail-manage-it/
[2] http://hadoop.apache.org/
[3] https://spark.apache.org/
[4] https://wiki.apache.org/hadoop/HDFS
[5] K. Doka, N. Papailiou, D. Tsoumakos, C. Mantas, and N. Koziris. Ires: Intelligent, multi- engine resource scheduler for big data analytics workflows. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 1451–1456. ACM, 2015.
[6] Agrawal, D., Chawla, S., Elmagarmid, A.K., Ouzzani, Z.K.M., Papotti, P., Quiané-Ruiz, J., Tang, N., Zaki, M.J.: Road to freedom in big data analytics. In: EDBT, pp. 479–484 (2016)
[7] https://www.cl.cam.ac.uk/research/srg/netos/camsas/musketeer/
[8] https://en.wikipedia.org/wiki/Natural_language_processing

[9] K. Doka, N. Papailiou, V. Giannakouris, D. Tsoumakos, and N. Koziris Mix 'n' Match Multi Engine Analytics. Published in 2016 IEEE International Conference on Big Data (Big Data)

[10] I. Giannakopoulos et al., "PANIC: Modeling Application Performance over Virtualized Resources," in 2015 IEEE International Conference on Cloud Engineering, IC2E 2015, 2015, pp. 213–218.
[11] https://en.wikipedia.org/wiki/Support_vector_machine
[12] "FrameworkA Free and Open Source Java Framework for Multiobjective Optimization," http://moeaframework.org/
[13] K. Deb et al., "A fast and elitist multiobjective genetic algorithm: Nsga-ii," IEEE transactions on evolutionary computation, vol. 6, no. 2, pp. 182–197, 2002.
[14] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[15] "Kitten: For Developers Who Like Playing with YARN, https://github.com/cloudera/kitten.
[16] D. Agrawal et al., "Rheem: Enabling multi-platform task execution," Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 2016
[17] https://en.wikipedia.org/wiki/K-means_clustering
[18] http://ganglia.sourceforge.net/

[19] H.-L. Truong and S. Dustdar, “Composable cost estimation and monitoring for computational applications in cloud computing environments,” Procedia Computer Science, vol. 1, no. 1, pp. 2175–2184, 2010.