ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Apache Hadoop για επεξεργασία μεγάλων δεδομένων σε κατανεμημένο περιβάλλον

Του φοιτητή
Κέρτζαλη Θωμά-Δαυίδ
Αρ. Μητρώου: 154468

Επιβλέπων
Ονοματεπώνυμο: Σιδηρόπουλος Αντώνης
Βαθμίδα: Επίκουρος Καθηγητής

Ημερομηνία 5/2/2022

Τίτλος Δ.Ε.: Apache Hadoop για επεξεργασία μεγάλων δεδομένων σε κατανεμημένο περιβάλλον
Κωδικός Δ.Ε. 21182
Ονοματεπώνυμο φοιτητή: Κέρτζαλης Θωμάς Δαυίδ
Ονοματεπώνυμο εισηγητή: Σιδηρόπουλος Αντώνης
Ημερομηνία ανάληψης Δ.Ε.: 12/3/2021
Ημερομηνία περάτωσης Δ.Ε.: 5/2/2022

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Περίληψη

Η παρούσα πτυχιακή εργασία αποτελείται από δύο μέρη. Το πρώτο, αφορά την θεωρητική προσέγγιση, ανάλυση και κατανόηση του Apache Hadoop καθώς και όλων των συστημάτων και στρωμάτων που το αποτελούν. Στο δεύτερο και πρακτικό μέρος της εργασίας καταπιάνεται με την κατασκευή και ρύθμιση ενός λειτουργικού και κατανεμημένου cluster υπολογιστών. Η εγκατάσταση του Hadoop στους υπολογιστές των εργαστηριών του τμήματος Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙΠΑΕ θα πραγματοποιηθεί αυτόματα με την συγγραφή bash scripts. Αυτός ο cluster υπολογιστών θα αποτελέσει την υποδομή για πειραματισμό σε ενδιαφέροντα πεδία της επιστήμης πληροφορικής όπως η μηχανική μάθηση και η ανάλυση δεδομένων.

Apache Hadoop for processing big data in a distributed environment

Thomas David Kertzalis

Abstract

This thesis consists of two parts. The first one pertains to the theoretical approach, analysis and comprehension of Apache Hadoop as well as all the systems and layers that it is comprised. The second part is the practical one and refers to the construction of a fully functional and distributed cluster of computers. The installation of Hadoop will be conducted automatically with some bash scripts and it will be performed in the computers of the laboratories of the Department of Information and Electronic Engineering of the International Hellenic University. This cluster will be the infrastructure that will allow experimentation in some interesting fields of Computer Science, such as Machine Learning and Data Science.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Κώδικα

Κατάλογος Αποτελεσμάτων εντολών

Συντομογραφίες

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Π.Ε.	Πτυχιακή Εργασία

Κεφάλαιο 1ο: Σύντομη ματιά στο Hadoop

1.1 Εισαγωγή

Είναι γνωστό πως διανύουμε την εποχή των δεδομένων. Με την τεχνολογία να είναι πιο προσβάσιμη προσιτή από ποτέ, όλο και περισσότεροι άνθρωποι χρησιμοποιούν ηλεκτρονικές συσκευές και παράγουν ολοένα και περισσότερα δεδομένα. Ας μιλήσουμε όμως με αριθμούς. Η αλήθεια είναι πως το συνολικό μέγεθος των δεδομένων που είναι ηλεκτρονικά αποθηκευμένο είναι δύσκολο να υπολογιστεί. Η IDC (International Data Corporation) εκτιμά πως αυτό το νούμερο ανερχόταν στα 4.4 zettabytes το 2013 και σε 44 zettabytes το 2020. Για να κατανοήσουμε αυτό το αστρονομικό μέγεθος, 1 zettabyte ισούται με $1 * 10^{15}$ megabytes (1,000,000,000,000,000 megabytes). Ας δώσουμε όμως και μερικά πιο συγκεκριμένα παραδείγματα προς αυτή την κατεύθυνση.

Το Χρηματιστήριο της Νέας Υόρκης παράγει περίπου 4-5 petabyes δεδομένων κάθε μέρα.
Το Facebook φιλοξενεί περισσότερα από 240 δισεκατομμύρια φωτογραφίες, αυξάνοντας τον όγκο των δεδομένων κατά 7 petabyes κάθε μήνα.
Ο Large Hadron Collider στο CERN στην Γενεύη παράγει 30 petabyes κατά προσέγγιση ετησίως.

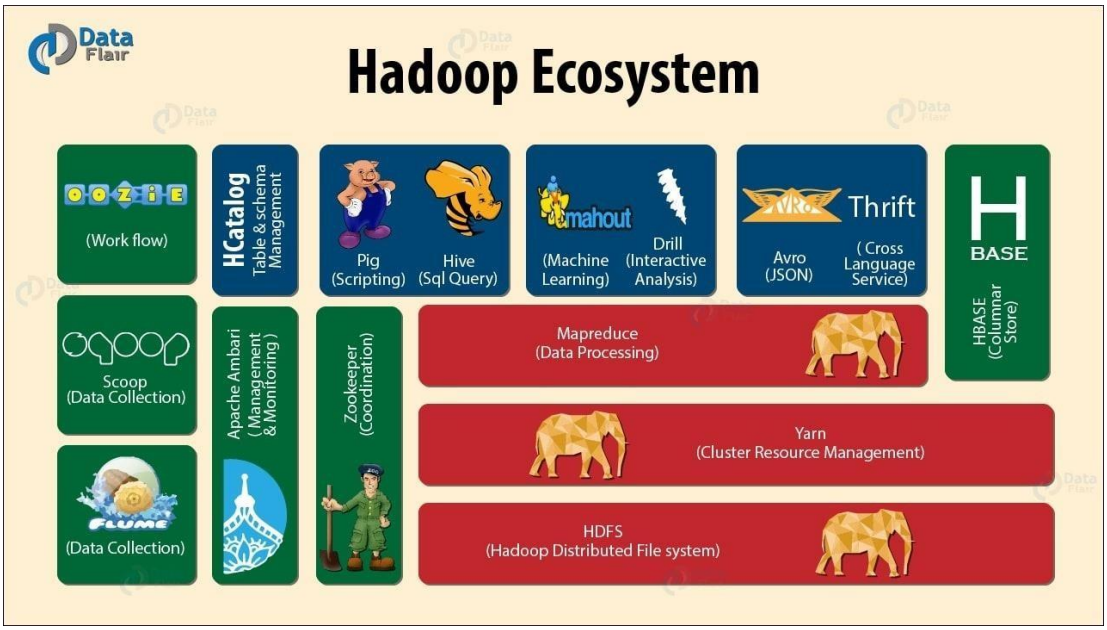
Όλα τα δεδομένα που αναφέρουμε παραπάνω παράγονται από ανθρώπους κατά κύριο λόγο. Αν σκεφτούμε πως η άνοδος του Internet Of Things είναι προ των πυλών, με RFID scanners, δίκτυα αισθητήρων ίχνη από GPS οχημάτων και πολλά άλλα να αρχίζουν να μπαίνουν σταδιακά στην καθημερινότητα μας στο κοντινό μέλλον, τα δεδομένα που παράγονται από τις μηχανές θα ξεπερνούν αυτά που προέρχονται από ανθρώπους. Ο πλέον σύγχρονος και δόκιμος όρος για τέτοιου είδους δεδομένα είναι γνωστός ως big data.

Ένα ακόμη πιο σημαντικό πρόβλημα από την αποθήκευση αυτών των τεράστιων σετ δεδομένων δεν είναι άλλο από την επεξεργασία τους. Η τελευταία είναι μια κρίσιμη διαδικασία που αν γίνει αποδοτικά μπορεί να μας δώσει χρήσιμες πληροφορίες που εσκεμμένα αναζητούμε, ακόμη όμως και να ανακαλύψει απρόσμενα μοτίβα μέσα στις ογκώδεις ποσότητες δεδομένων. Για παράδειγμα, η ανάλυση δεδομένων από δημοσιεύσεις στα κοινωνικά δίκτυα μπορεί να φανερώσει κάποια ενδιαφέροντα ζητήματα όπως την δημοφιλία ενός προσώπου ή ενός προϊόντος. Επιπρόσθετα αν φανταστούμε μια τεράστια φάρμα που εποπτεύεται από χιλιάδες αισθητήρες και drones τα οποία παράγουν μεγάλους όγκους δεδομένων, τα οποία πρέπει να επεξεργαστούν κατάλληλα ώστε να εξαχθούν ζωτικής σημασίας πληροφορίες που θα βοηθήσουν στην σωστή διαχείριση και ανάπτυξη της φάρμας.

Για να γίνουν όμως όλα αυτά πραγματικότητα χρειαζόμαστε και υπολογιστές, δηλαδή hardware. Δεδομένου όμως του όγκου των big data χρειαζόμαστε πολλούς υπολογιστές οι οποίοι θα συνεργάζονται μεταξύ τους με σκοπό την αποθήκευση και επεξεργασία δεδομένων. Ένα τέτοιο σύστημα είναι και το Hadoop.

Το Hadoop είναι ένα οικοσύστημα προγραμμάτων ανοιχτού κώδικα που είναι βασισμένο στην κατανεμημένη λογική και διευκολύνει τη χρήση ενός δικτύου πολλών υπολογιστών για την επίλυση προβλημάτων που περιλαμβάνουν τεράστιες ποσότητες δεδομένων και υπολογισμούς

Αρχικά ας περιγράψουμε σύντομα τα διαφορετικά επίπεδα της στοίβας του Hadoop και τον ρόλο που διαδραματίζει το καθένα από αυτά.



Εικόνα 1.1Επίπεδα της στοίβας του Hadoop

Όπως βλέπουμε στην παραπάνω εικόνα τα τρία κύρια επίπεδα στη στοίβα του Hadoop (χρωματισμένα με κόκκινο) είναι το HDFS, το YARN και το MapReduce. Αυτή η πτυχιακή θα εστιάσει κυρίως σε αυτά τα σημεία αναλύοντας τα και θα αναφέρει πιθανότατα λίγα λόγια για κάποια από τα υπόλοιπα χαρακτηριστικά του Hadoop (χρωματισμένα με μπλε και πράσινο).

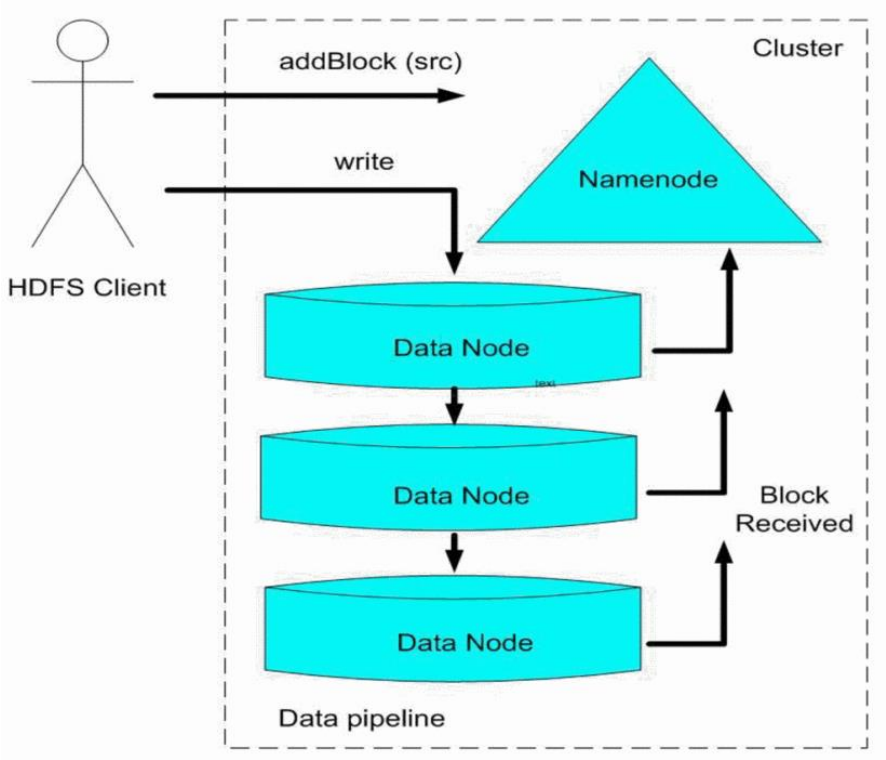
Το HDFS σημαίνει Hadoop Distributed File System. Είναι ένα σύστημα αρχείων βασισμένο στην κατανεμημένη προσέγγιση και αποτελεί τη βάση της πυραμίδας, παίζοντας τον ρόλο της αποθήκευσης. Βασίζεται στο UNIX σύστημα αρχείων και αποθηκεύει metadata και πραγματικά δεδομένα ξεχωριστά σε διαφορετικά μηχανήματα (κόμβους) , συνήθως σε μπλοκ των 128MB.

Το HDFS διαθέτει τρία κύρια συστατικά μέρη, τον NameNode, τον DataNode και τον HDFS client. Ο HDFS client είναι μια οντότητα που προσθέτει μπλοκ δεδομένων στον NameNode. Επίσης γράφει δεδομένα σε κάποιους DataNodes. Κάθε DataNode επικοινωνεί με τον NameNode με την μορφή του μπλοκ δεδομένων.

1.2 Σύντομη περιγραφή HDFS

Το HDFS σημαίνει Hadoop Distributed File System. Είναι ένα σύστημα αρχείων βασισμένο στην κατανεμημένη προσέγγιση και αποτελεί τη βάση της πυραμίδας, παίζοντας τον ρόλο της αποθήκευσης. Βασίζεται στο UNIX σύστημα αρχείων και αποθηκεύει metadata και πραγματικά δεδομένα ξεχωριστά σε διαφορετικά μηχανήματα (κόμβους) , συνήθως σε μπλοκ των 128MB.

Το HDFS διαθέτει τρία κύρια συστατικά μέρη, τον NameNode, τον DataNode και τον HDFS client. Ο HDFS client είναι μια οντότητα που προσθέτει μπλοκ δεδομένων στον NameNode. Επίσης γράφει δεδομένα σε κάποιους DataNodes. Κάθε DataNode επικοινωνεί με τον NameNode με την μορφή του μπλοκ δεδομένων.



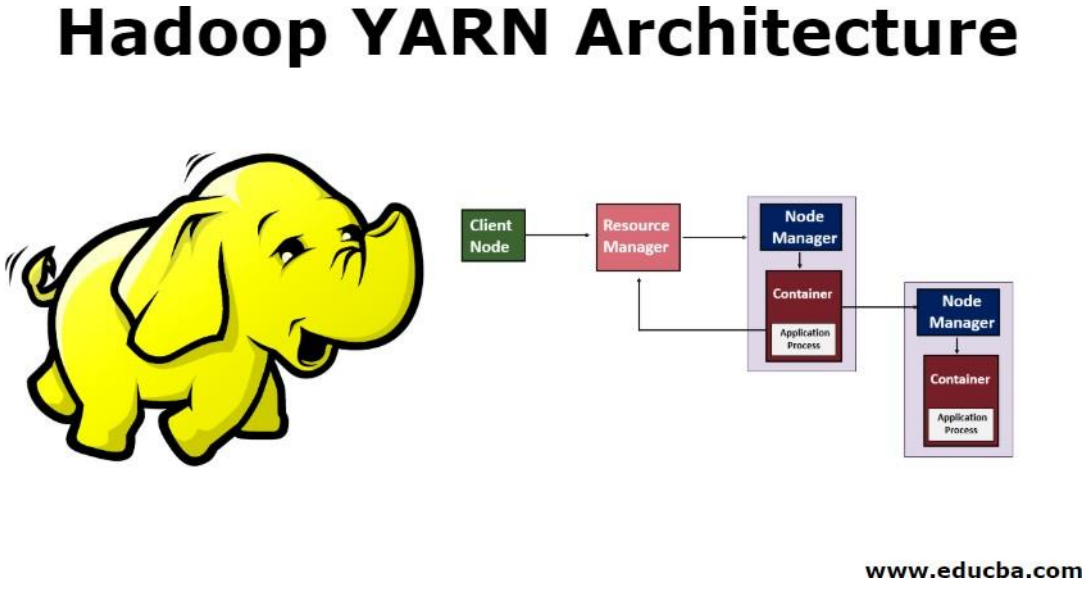
Εικόνα 1.2 Αρχιτεκτονική HDFS

Ο NameNode είναι ο υπολογιστής στο οποίο αποθηκεύονται τα μεταδεδομένα. Αυτά τα μεταδεδομένα εμπεριέχουν την αναπαράσταση του συστήματος αρχείων (αρχεία και κατάλογοι) ανά τους κόμβους καθώς και χαρακτηριστικά όπως άδειες αρχείων και χρόνους πρόσβασης για κάθε διαθέσιμο δίσκο. Επιπρόσθετα, διαθέτει ένα namespace δέντρο και χαρτογραφεί το ποια μπλοκ δεδομένων βρίσκονται σε ποιους DataNodes. Πιο συγκεκριμένα ο χρήστης παίρνει την τοποθεσία των αρχείων στους DataNodes που είναι προς ανάγνωση. Ο NameNode παρέχει τις απαραίτητες πληροφορίες για τους DataNodes, κατά την διάρκεια πράξεων εγγραφής.

Οι DataNodes είναι τα μηχανήματα στα οποία αποθηκεύονται τα πραγματικά δεδομένα της εφαρμογής σε μπλοκ των 128MB. Αυτά τα δεδομένα αντιγράφονται σε πολλαπλούς DataNodes, έτσι ώστε να αποτρέπεται η απώλεια κάποιου μπλοκ σε περίπτωση βλάβης κάποιου δίσκου. Ο χρήστης μπορεί να εκτελέσει πράξεις ανοίγματος, κλεισίματος, εγγραφής και ανάγνωσης στα αρχεία που υπόκεινται σε κάθε DataNode.

1.3 Σύντομη περιγραφή YARN

Το Apache YARN (Yet Another Resource Negotiator) κατοικεί στη μέση της στοίβας του Hadoop (βλέπε Εικόνα 1.3) και αποτελεί τον συνδετικό κρίκο ανάμεσα στο HDFS και το MapReduce. Εισήχθη στο Hadoop 2.X.X και είναι υπεύθυνο για την διαχείριση και τον διαμοιρασμό των πόρων (αποθηκευτικοί δίσκοι και πυρήνες επεξεργαστών) ανά τα μηχανήματα του cluster. Στην ουσία ξεχώρισε το διαχειριστικό στρώμα από το υπολογιστικό στρώμα, κάτι που δεν ίσχυε στο Hadoop 1.X.X. Αποτελείται από τρία κύρια χαρακτηριστικά, τον Resource Manager , τον Node Manager και ο Container.



Εικόνα 1.3 Αρχιτεκτονική YARN

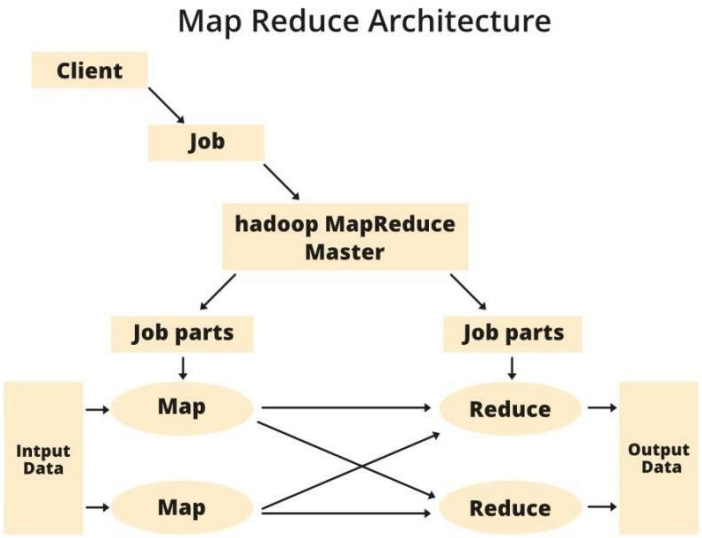
Ο Resource Manager ευθύνεται για την ανάθεση και την διαχείριση πόρων σε κάθε τύπου εφαρμογή. Όποτε δέχεται ένα αίτημα επεξεργασίας, το προωθεί στον κατάλληλο Node Manager και αναθέτει σε αυτόν πόρους για την ολοκλήρωση του αιτήματος. Διαθέτει δύο επιμέρους κομμάτια, τον Scheduler και τον Application Manager για τους οποίους θα γίνει λόγος σε επόμενο κεφάλαιο για λόγους απλότητας επί του παρόντος.

Ο Node Manager φροντίζει κάθε ένα ξεχωριστό κόμβο στον cluster και διαχειρίζεται την εφαρμογή και τη ροή των εργασιών για καθέναν από αυτούς. Παρακολουθεί τα επίπεδα χρήσης των πόρων και κάνει διαχείριση των logs, και την διαγραφή ή επαναδημιουργία ενός Container βάσει των οδηγιών που έχουν ληφθεί από τον Resource Manager.

Ο Container είναι απλά μια συλλογή φυσικών πόρων όπως RAM, πυρήνες επεξεργαστών και δίσκοι σε έναν κόμβο.

1.4 Σύντομη περιγραφή MapReduce

Το MapReduce είναι ένα προγραμματιστικό μοντέλο που δημιουργήθηκε με σκοπό την αποδοτική παράλληλη επεξεργασία μεγάλων όγκων δεδομένων με κατανεμημένη προσέγγιση. Τα δεδομένα ξεχωρίζονται σε πρώτη φάση και στη συνέχεια συνδυάζονται για την παραγωγή του τελικού αποτελέσματος. Οι βιβλιοθήκες του MapReduce είναι γραμμένες σε πολλές προγραμματιστικές γλώσσες και καθεμία από τις οποίες προσφέρει διαφορετικές και ποικίλες δυνατότητες. Το MapReduce στοχεύει στην καταγραφή (Map) κάθε Job και ύστερα την μείωση (Reduce) τους σε αντίστοιχα Tasks έτσι ώστε να ελαχιστοποιηθεί η καθυστέρηση στο δίκτυο του cluster, αλλά και για την μείωση της επεξεργαστικής ισχύς. Ένα MapReduce Task αποτελείται από δύο κύριες φάσεις, την φάση Map και την φάση Reduce.



Εικόνα 1.4Αρχιτεκτονική MapReduce

Χαρακτηριστικά της αρχιτεκτονικής MapReduce :

Client: Ο MapReduce client είναι η οντότητα που φέρνει ένα Job για επεξεργασία στο MapReduce. Υπάρχει δυνατότητα ύπαρξης πολλαπλών Clients που στέλνουν Jobs προς επεξεργασία στον MapReduce Master
Job: Είναι η πραγματική δουλεία που ο client ζητά να πραγματοποιηθεί και αποτελείται από επιμέρους μικρότερα κομμάτια που χρίζουν επεξεργασίας και εκτέλεσης.
MapReduce Master: Διαχωρίζει κάθε Job σε μικρότερες εργασίες (Tasks)
Tasks: Τα μικρότερα κομμάτια που προκύπτουν από τον διαχωρισμό ενός Job. Το αποτέλεσμα των Tasks συνδυάζεται για να προκύψει το τελικό αποτέλεσμα
Δεδομένα Input: Τα δεδομένα που εισάγονται στο MapReduce για επεξεργασία
Δεδομένα Output: Το τελικό αποτέλεσμα που απορρέει μετά την επεξεργασία

Αφού περιγράψαμε τα τρία βασικά στρώματα του Hadoop και τα χαρακτηριστικά από τα οποία αυτά αποτελούνται, είναι χρήσιμο να αναφέρουμε τα πλεονεκτήματα του Hadoop που είναι και οι λόγοι για τους οποίους είναι επιθυμητή η χρήση του.

1.5 Πλεονεκτήματα Hadoop

Δυνατότητα παράλληλης ανάγνωσης και εγγραφής δεδομένων από και προς πολλαπλούς δίσκους
Δυνατότητα αποθήκευσης πολλαπλών αντιγράφων σε διαφορετικούς υπολογιστές
Δυνατότητα αποδοτικού συνδυασμού πληροφορίας που βρίσκονται σε διαφορετικούς δίσκους

Αν και οι δυνατότητες αποθήκευσης των σκληρών δίσκων έχουν βελτιωθεί κατακόρυφα με το πέρασμα των χρόνων, οι ταχύτητες ανάκτησης, δηλαδή ο ρυθμός με τον οποίο δεδομένα μπορούν να διαβαστούν από τον δίσκο, δεν έχει συμβαδίσει. Για παράδειγμα, ένας τυπικός σκληρός δίσκος από το 1990 μπορούσε να αποθηκεύσει 1,370 MB δεδομένων με ταχύτητα ανάκτησης στα 4.4 MB/s, επομένως το σύνολο των δεδομένων μπορούσε να διαβαστεί σε περίπου πέντε λεπτά. Περισσότερο από δύο δεκαετίες μετά, το πιο σύνηθες μέγεθος για έναν σκληρό δίσκο ανέρχεται στο 1 terabyte, με

ταχύτητα ανάκτησης στα 100 MB/s. Αυτό σημαίνει ότι η ο απαιτούμενος χρόνος για την ανάγνωση όλων των δεδομένων του δίσκου είναι κάτι περισσότερο από δυόμιση ώρες με τις ταχύτητες εγγραφής να είναι ακόμη μικρότερες. Είναι προφανές λοιπόν, πως τέτοιου είδους χρόνοι είναι απαγορευτικοί για σύγχρονες εφαρμογές. Ο πιο απλός τρόπος για να αντιμετωπιστεί αυτό το πρόβλημα είναι η ταυτόχρονη ανάγνωση δεδομένων από πολλαπλούς δίσκους.

Αν είχαμε λόγου χάρη 100 δίσκους στους οποίους αποθηκεύαμε 10 gigabytes δεδομένων στο καθένα, δουλεύοντας παράλληλα θα μπορούσαν να αναγνωστούν όλα τα δεδομένα σε περίπου δύο λεπτά. Μπορεί να χρησιμοποιούμε στην ουσία το ένα εκατοστό της χωρητικότητας κάθε δίσκου, όμως παίρνουμε σε αντάλλαγμα ταχύτερους χρόνους επεξεργασίας και επομένως ευχαριστημένους χρήστες της εφαρμογής

Υπάρχουν όμως περισσότεροι λόγοι που κάνουν το Hadoop πιο ελκυστικό από την παράλληλη εγγραφή και ανάγνωση δεδομένων από πολλαπλούς δίσκους.

Εφόσον χρησιμοποιούνται πολλά κομμάτια hardware, η πιθανότητα βλάβης ενός από αυτών είναι σχετικά υψηλή. Αυτό μπορεί να λυθεί με την διατήρηση πολλαπλών αντιγράφων αποθηκευμένα σε διαφορετικά μηχανήματα για λόγους πλεονασμού (redundancy). Οπότε αν κάτι πάει στραβά και υπάρξει απώλεια ενός δίσκου, δεν θα χαθούν τα δεδομένα που ήταν αποθηκευμένα εκεί αφού μπορούν να ανακτηθούν κάποιο από τα αντίγραφα του. Με αυτόν τον τρόπο λειτουργεί και η τεχνολογία RAID, όμως το HDFS ακολουθεί μια λίγο διαφορετική προσέγγιση στο συγκεκριμένο ζήτημα.

Μία άλλη κρίσιμη πλευρά ενός τέτοιου συστήματος είναι δυνατότητα συνδυασμού των δεδομένων με κάποιο τρόπο, έτσι ώστε να προκύψει ένα κράμα των δεδομένων από των έναν δίσκο με αυτά από τους υπόλοιπους δίσκους του cluster. Το Hadoop δεν είναι ούτε το πρώτο, ούτε και το μοναδικό σύστημα που επιχείρησε ένα τέτοιου είδους εγχείρημα. Πολλά άλλα κατανεμημένα συστήματα ακολουθούν ένα παρόμοιο μονοπάτι. Όμως για να γίνει κάτι τέτοιο σωστά και πολλώ δε μάλλον αποδοτικά, είναι εξαιρετικά δύσκολο. Το MapReduce προσφέρει ένα προγραμματιστικό μοντέλο που προσεγγίζει το θέμα από ένα υψηλότερο επίπεδο, διαχωρίζοντας το από εγγραφές και αναγνώσεις σε δίσκους (χαμηλό επίπεδο), μεταμορφώνοντας το σε έναν υπολογισμό με αποτέλεσμα ζευγάρια ονομάτων-τιμής. Αυτό γίνεται σε δύο φάσεις Map και Reduce όπως προαναφέραμε, καθώς και στην διασύνδεση αυτών, όπου γίνεται και αυτή η πολυπόθητη μίξη δεδομένων.

Συμπερασματικά, το Hadoop προσφέρει μία αξιόπιστη και επεκτάσιμη λύση για αποθήκευση και επεξεργασία. Επιπρόσθετα, επειδή μπορεί να τρέξει σε κοινότυπα μηχανήματα καθώς και ότι είναι λογισμικό ανοιχτού κώδικα, καθιστά το Hadoop προσιτό.

Πριν ξεκινήσουμε να αναλύουμε τα βασικά κομμάτια του Hadoop, κρίνεται σκόπιμη η σύγκριση του σε σχέση με τις παραδοσιακές σχεσιακές βάσεις δεδομένων (RDBMS), που τα βασικά χαρακτηριστικά και η λειτουργία τους είναι σχετικά γνωστές. Με αυτόν τον τρόπο, θα έχουμε κατανοήσει τις διαφορές αλλά τους λόγους και τις περιπτώσεις που συστήνεται η χρήση του Hadoop.

1.6 Σύγκριση με Σχεσιακές Βάσεις Δεδομένων

Κατά μία έννοια, το MapReduce μπορεί να θεωρηθεί ως μια βελτίωση ενός RBDMS. Το MapReduce είναι προτιμότερο για προβλήματα που είναι απαραίτητη η ανάλυση ολόκληρου του σετ δεδομένων. Ένα RBDMS ταιριάζει καλύτερα σε ερωτήματα που αφορούν μία συγκεκριμένη τιμή ενός από τα πεδία ενός πίνακα (point queries) ή για ενημερώσεις (updates), το σετ δεδομένων έχει ευρετηριαστεί

για να παραδίδει και να ενημερώνει σχετικά μικρούς όγκους δεδομένων με μικρές καθυστερήσεις. Το MapReduce αρμόζει σε δεδομένα που εγγράφονται μία φορά και διαβάζονται πολλές φορές, ενώ μία σχεσιακή βάση δεδομένων αποδίδει καλύτερα σε δεδομένα που ενημερώνονται συνεχώς.

	Σχεσιακές βάσεις	MapReduce
Μέγεθος δεδομένων	Gigabytes	Petabytes
Πρόσβαση	Interactive and batch	Batch
Ενημερώσεις	Read and write many times	Write once, read many times
Συναλλαγές	ACID	None
Δομή	Schema-on-write	Schema-on-read
Ακεραιότητα	High	Low
Ανάπτυξη	Nonlinear	Linear

Πίνακας 1.1 Σύγκριση RDBMS και MapReduce

Με εξαίρεση τα παραπάνω, οι διαφορές ανάμεσα στις σχεσιακές βάσεις δεδομένων και το Hadoop δεν είναι και τόσο ξεκάθαρες. Οι σχεσιακές βάσεις έχουν αρχίσει να δανείζονται κάποια χαρακτηριστικά από το Hadoop και από την άλλη υποσυστήματα του Hadoop όπως το Hive προσθέτουν διαδραστικά στοιχεία όπως για παράδειγμα ευρετήρια και συναλλαγές.

Μια ακόμη διαφορά μεταξύ των δύο συστημάτων είναι και η δομή των σετ δεδομένων τα οποία επεξεργάζονται. Τα δομημένα δεδομένα οργανώνονται σε οντότητες που έχουν μια προκαθορισμένη μορφή όπως ένα XML έγγραφο. Σε αυτού του τύπου τα δεδομένα λειτουργούν τα RBDMS. Τα ημιδομημένα δεδομένα έχουν πιο χαλαρή δομή. Αν και μπορεί να υπάρχει σχήμα, συχνά αγνοείται ή χρησιμοποιείται ενδεικτικά ανάλογα την περίπτωση. Τα αδόμητα δεδομένα δεν έχουν καμία συγκεκριμένη μορφή. Ένα παράδειγμα αδόμητων δεδομένων είναι ένα απλό κείμενο (plain text) ή μια εικόνα. Το Hadoop αποδίδει καλύτερα σε ημιδομημένα και αδόμητα δεδομένα διότι έχει σχεδιαστεί να μεταφράζει τα δεδομένα κατά την επεξεργασία (schema-on-write). Αυτό προσδίδει ευελιξία και γλιτώνει την κοστοβόρα φάση της φόρτωσης των δεδομένων, αφού για το Hadoop είναι απλά ένα αντίγραφο αρχείου.

Το MapReduce όπως και άλλα επεξεργαστικά μοντέλα στο οικοσύστημα του Hadoop (π.χ. το Spark) αναπτύσσονται γραμμικά σε σχέση με το μέγεθος των δεδομένων. Αυτό πρακτικά σημαίνει ότι αν διπλασιαστούν τα δεδομένα εισαγωγής (input data), ένα Job θα τρέξει δύο φορές πιο αργά. Αν όπως διπλασιαστεί το μέγεθος του cluster, το Job θα εκτελεστεί όσο γρήγορα εκτελέστηκε και το Job στην πρώτη περίπτωση. Αυτό δεν ισχύει σε γενικές γραμμές για ερωτήματα SQL.

1.7 Ιστορική Αναδρομή του Hadoop

Οι απαρχές του Hadoop ξεκινούν το 2002 όταν ο Dough Cutting και ο Mike Cafarella ξεκίνησαν να δουλεύουν στο Apache Nutch.Το Apache Nutch ήταν ένα εγχείρημα κατασκευής μιας μηχανής αναζήτησης που θα μπορούσε να ευρετηριάσει 1 δισεκατομμύριο σελίδες. Μετά από εξονυχιστική έρευνα, συμπέραναν ότι ένα τέτοιο σύστημα θα κόστιζε περίπου μισό εκατομμύριο δολάρια σε hardware και τα μηνιαία κόστη λειτουργίας θα ανέρχονταν σε τριάντα χιλιάδες δολάρια, κάτι που προφανώς είναι υπερβολικά ακριβό. Έτσι συνειδητοποίησαν ότι θα πρέπει να βρουν μία λύση έτσι ώστε να μειωθεί το κόστος υλοποίησης όσο αλλά και να λυθεί το πρόβλημα της αποθήκευσης και επεξεργασίας μεγάλων σετ δεδομένων.

Το 2003, βρήκαν ένα άρθρο που περιέγραφε την αρχιτεκτονική του κατανεμημένου συστήματος της Google για αποθήκευση μεγάλων σετ δεδομένων, το οποίο ονομαζόταν GFS (Google File System) , δημοσιευμένο από την ίδια την Google. Αυτό όμως έλυσε το πρώτο μισό του προβλήματος, δηλαδή την αποθήκευση.

Το 2004, και πάλι η Google δημοσίευσε άλλο ένα άρθρο σχετικά με την τεχνική MapReduce, η οποία έλυνε το πρόβλημα της επεξεργασίας μεγάλων σετ δεδομένων. Παρ' όλα αυτά, οι δύο αυτές τεχνικές ήταν ακόμη σε θεωρητικό επίπεδο και δεν είχαν εφαρμοστεί στην πράξη από την Google. Έτσι, οι Dough Cutting και Mike Cafarella αποφάσισαν να εφαρμόσουν αυτές τις τεχνικές σε ένα εγχείρημα ανοιχτού κώδικα.

Το 2006, ο Cutting ξεκίνησε να εργάζεται για την Yahoo και πήρε μαζί του το Nutch. Ήθελε να προσφέρει στον κόσμο μια αξιόπιστη, επεκτάσιμη πλατφόρμα ανοιχτού κώδικα, με την βοήθεια της Yahoo. Έτσι ως μέλος της Yahoo, απέσπασε το κατανεμημένο υπολογιστικό κομμάτι από το Nutch και ίδρυσε ένα καινούργιο πρότζεκτ, το Hadoop.

Το 2007, η Yahoo τέσταρε επιτυχώς το Hadoop σε έναν cluster υπολογιστών με 1000 κόμβους και ξεκίνησε να το χρησιμοποιεί.

Τον Ιανουάριο το 2008, η Yahoo κυκλοφόρησε το Hadoop σαν λογισμικό ανοιχτού κώδικα υπό την αιγίδα της Apache Software Foundation. Επίσης το 2008 η Apache κατάφερε να τρέξει το Hadoop σε έναν cluster 4000 κόμβων.

Το 2009, το Hadoop κατάφερε να ταξινομήσει ένα petabyte δεδομένων σε λιγότερο από 17 ώρες, αφού ευρετηρίασε εκατομμύρια ιστοσελίδες.

Το 2011, η Apache κυκλοφόρησε την πρώτη έκδοση του και τον Αύγουστο την δεύτερη.

Τον Δεκέμβριο του 2017 κυκλοφόρησε και η τρίτη έκδοση.

Σε αυτό το σημείο ολοκληρώνεται το πρώτο κεφάλαιο έχοντας περιγράψει τα κύρια συστατικά του Hadoop (HDFS, YARN, MapReduce), όπως επίσης και τα πλεονεκτήματα που αποφέρει η χρήση του. Επίσης έγινε λόγος για την σύγκριση του Hadoop με τις γνωστές μας σχεσιακές βάσεις δεδομένων. Τέλος, αναφέρθηκαν μερικά ιστορικά στοιχεία για την ανάπτυξη και την εξέλιξη της πλατφόρμας. Στα κεφάλαια που ακολουθούν, θα περιγραφούν αναλυτικά τα τρία κύρια προαναφερθέντα στοιχεία, το

καθένα σε ξεχωριστό κεφάλαιο. Επίσης, θα δοθούν οδηγίες για την εγκατάσταση, παραμετροποίηση και λειτουργία του Hadoop, περιγράφοντας την διαδικασία βήμα-βήμα.

Κεφάλαιο 2ο: Αναλυτική περιγραφή του HDFS

2.1 Εισαγωγή

Το HDFS σύστημα αρχείων αποθηκεύει πολύ μεγάλα αρχεία με τεχνικές ροής δεδομένων που τρέχει σε κοινότυπο hardware. Τα αρχεία σε μέγεθος κυμαίνονται από εκατοντάδες megabytes , gigabytes ή και terabytes. Στις μέρες μας υπάρχουν Hadoop clusters που αποθηκεύουν Petabytes δεδομένων.

Το HDFS σχεδιάστηκε με γνώμονα ότι η πιο αποδοτική τεχνική για επεξεργασία δεδομένων, είναι αυτή κατά την οποία τα δεδομένα αποθηκεύονται μια και διαβάζονται πολλές φορές. Ένα σετ δεδομένων δημιουργείται ή αντιγράφεται από μία πηγή και εν συνεχεία γίνονται πολλές αναλύσεις πάνω σε αυτό με το πέρασμα του χρόνου. Κάθε ανάλυση χρησιμοποιεί ένα μεγάλο κομμάτι των δεδομένων (αν όχι ολόκληρο το σετ δεδομένων), οπότε ο χρόνος που απαιτείται για το διάβασμα όλου του σετ δεδομένων είναι σημαντικός.

Το Hadoop δεν χρειάζεται ακριβό και πολύ αξιόπιστο hardware. Έχει φτιαχτεί έτσι ώστε να τρέχει σε clusters που αποτελούνται από κοινό hardware (άμεσα διαθέσιμο που μπορεί να αγοραστεί από πολλαπλούς προμηθευτές σε λογικές τιμές), για τους οποίους η πιθανότητα ενός κόμβου να αποτύχει είναι σχετικά μεγάλη, ειδικά σε μεγάλους clusters. Το HDFS μπορεί να διαχειριστεί τέτοιες καταστάσεις χωρίς ο χρήστης να παρατηρήσει κάποια αλλαγή στην υπηρεσία που χρησιμοποιεί.

Θα ήταν επίσης αρκετά ωφέλιμο να αναφέρουμε και τις εφαρμογές για τις οποίες το HDFS δεν ανταποκρίνεται και τόσο καλά. Αν και αυτό μπορεί στο μέλλον να αλλάξει, αυτές είναι οι περιοχές στις οποίες το HDFS υστερεί.

Πρόσβαση δεδομένων με μικρή καθυστέρηση : Εφαρμογές που απαιτούν πρόσβαση σε δεδομένα με πολύ μικρή καθυστέρηση, της τάξης των δεκάτων του εκατομμυριοστού του δευτερολέπτου δεν θα δουλέψουν σωστά με το HDFS. Το τελευταίο έχει φτιαχτεί για να παραδίδει δεδομένα με υψηλή διεκπεραιωτική ικανότητα , κάτι που μπορεί να θυσιάσει την καθυστέρηση για να επιτύχει το σκοπό του.

Πολλά μικρά αρχεία : Επειδή ο namenode εμπεριέχει τα μεταδεδομένα του συστήματος αρχείων στην μνήμη, το όριο του αριθμού των αρχείων σε ένα τέτοιου είδους κατανεμημένο συστήματος αρχείων τίθεται από την μνήμη του namenode. Κάθε αρχείο, κατάλογος και μπλοκ δεδομένων πιάνει περίπου 150 bytes. Έτσι, για παράδειγμα αν είχαμε ένα εκατομμύριο αρχεία, που το καθένα τους έπιανε ένα μπλοκ, τότε θα χρειαζόμασταν 300 MB μνήμης. Αν και η αποθήκευση εκατομμυρίων αρχείων είναι υλοποιήσιμη, τα δισεκατομμύρια αρχεία είναι πέραν από τις δυνατότητες του υπάρχοντος hardware.

Πολλαπλοί εγγραφείς : Τα αρχεία στο HDFS μπορούν να γράφονται από μόνο έναν. Οι εγγραφές γίνονται πάντα στο τέλος του αρχείου, δηλαδή με νέες προσθήκες στο ήδη υπάρχον περιεχόμενο. Επίσης δεν υποστηρίζονται οι πολλαπλοί εγγραφείς.

2.2 Μπλοκ αρχείων

Ένας δίσκος διαθέτει ένα μέγεθος μπλοκ, το οποίο είναι η μικρότερη μονάδα μέτρησης δεδομένων που μπορεί να αναγνωστεί ή να εγγραφεί. Τα συστήματα αρχείων μονού δίσκου βασίζονται πάνω σε αυτό

και χειρίζονται τα δεδομένα ως μπλοκ που είναι όμως πολλαπλάσιο από το μπλοκ του δίσκου. Τα μπλοκ των συστημάτων αρχείων είναι συνήθως μερικά KB, ενώ αυτά των δίσκων πιάνουν 512 bytes στις περισσότερες περιπτώσεις.

Το HDFS υιοθετεί και αυτό την ιδέα του μπλοκ, αλλά ως μία πολύ μεγαλύτερη μονάδα (128 MB). Όπως και σε ένα σύστημα αρχείων με έναν δίσκο, τα αρχεία στο HDFS χωρίζονται σε μπλοκ που αποθηκεύονται σαν ανεξάρτητα κομμάτια. Σε αντίθεση με ένα σύστημα αρχείων μονού δίσκου, ένα αρχείο στο HDFS που είναι μικρότερο από ένα μπλοκ δεν καταλαμβάνει ολόκληρο το μέγεθος του μπλοκ στη μνήμη. Για παράδειγμα, αν έχουμε ένα αρχείο 1 MB σε ένα μπλοκ 128 MB, αυτό το μπλοκ θα χρησιμοποιήσει 1 μόνο MB μνήμης

Γιατί όμως ένα μπλοκ στο HDFS είναι τόσο μεγαλύτερο ; Σαφώς τα μπλοκ στο HDFS είναι κατά πολύ μεγαλύτερα από τα μπλοκ των δίσκων. Αυτό συμβαίνει διότι έτσι ελαχιστοποιείται το κόστος των αναζητήσεων. Αν ένα μπλοκ είναι αρκετά μεγάλο, ο χρόνος που χρειάζεται για να μεταφερθούν τα δεδομένα από τον δίσκο μπορεί να είναι αρκετά μεγαλύτερος από τον χρόνο αναζήτησης της αρχής του μπλοκ. Συνεπώς, η μεταφορά ενός μεγάλου αρχείου που αποτελείται από πολλά μπλοκ λειτουργεί με τον ρυθμό μεταφοράς του δίσκου.

Με έναν γρήγορο υπολογισμό φαίνεται ότι αν ο χρόνος αναζήτησης είναι γύρω στα 10 ms και ο ρυθμός μεταφοράς είναι 100 MB/s, για να κάνουμε τον χρόνο αναζήτησης το 1% του χρόνου μεταφοράς, πρέπει να έχουμε ένα μπλοκ μεγέθους 100 MB, που είναι πολύ κοντά στα 128 MB που είναι το προεπιλεγμένο μεγεθος των μπλοκ του HDFS.

Η αφαιρετικότητα των μπλοκ για ένα κατανεμημένο σύστημα αρχείων φέρνει μαζί της αρκετά πλεονεκτήματα. Πρώτον, ένα αρχείο μπορεί να είναι μεγαλύτερο από όλους τους δίσκους τους cluster. Δεν υπάρχει κάποιος περιορισμός που να απαιτεί τα μπλοκ ενός αρχείου να είναι αποθηκευμένα στον ίδιο δίσκο, ώστε να μπορούν να εκμεταλλευτούν όλοι οι δίσκοι που υπάρχουν διαθέσιμοι.

Δεύτερον, κάνοντας την μονάδα ένα μπλοκ και όχι ένα αρχείο, απλοποιείται το υποσύστημα της αποθήκευσης. Η απλότητα είναι εξαιρετικά σημαντική για ένα κατανεμημένο σύστημα στο οποίο οι βλάβες και οι αποτυχίες των κόμβων ποικίλουν. Επειδή τα μπλοκ έχουν ένα συγκεκριμένο μέγεθος είναι εύκολο να υπολογιστεί πόσα μπορούν να αποθηκευτούν σε έναν δίσκο. Επίσης τα μπλοκ είναι απλά κομμάτια δεδομένων. Μεταδεδομένα όπως άδειες αρχείων δεν χρειάζεται να αποθηκευτούν μέσα στο μπλοκ, οπότε ένα άλλο σύστημα μπορεί εύκολα να χειριστεί αυτή την πληροφορία.

Επιπρόσθετα, τα μπλοκ ταιριάζουν και με την ύπαρξη πολλαπλών αντιγράφων έτσι ώστε να προσφέρεται ανοχή σε λάθη και υψηλή διαθεσιμότητα των δεδομένων. Για να εξασφαλιστεί η ακεραιότητα των δεδομένων από διάφορα σφάλματα που μπορεί να προκύψουν, κάθε μπλοκ αντιγράφεται σε έναν μικρό αριθμό φυσικών μηχανημάτων (τυπικά τρία). Αν ένα μπλοκ δεν είναι διαθέσιμο, ένα αντίγραφό του μπορεί να βρεθεί σε μια άλλη τοποθεσία. Αν είναι μπλοκ έχει χαθεί, μπορεί να αντιγραφεί από ένα άλλο υπολογιστή στον οποίο υπάρχει ήδη, έτσι ώστε ο αριθμός των αντιγράφων να επανέλθει στον καθορισμένο αριθμό.

2.3 Namenodes & Datanodes

Ένα HDFS cluster διαθέτει δύο τύπους κόμβων που λειτουργούν με μια master – worker αρχιτεκτονική. Ο namenode είναι ο master και οι datanodes είναι οι workers. Ο namenode διαχειρίζεται το namespace του συστήματος αρχείων. Διατηρεί το δέντρο του συστήματος αρχείων για όλα τα αρχεία και τους καταλόγους στο δέντρο. Αυτή η πληροφορία αποθηκεύεται στην μορφή δύο αρχείων : του namespace image και του edit log. Ο namenode γνωρίζει επίσης τους datanodes στους οποίους βρίσκονται τα μπλοκ για κάποιο αρχείο.

Ένας client παίρνει πρόσβαση στο σύστημα αρχείων για λογαριασμό ενός χρήστη επικοινωνώντας με τον namenode και τους datanodes. Ο client παρουσιάζει μια διεπαφή για το σύστημα αρχείων, έτσι ώστε ο χρήστης να μην χρειάζεται να ξέρει για τον namenode και τους datanodes για να χρησιμοποιήσει την εφαρμογή.

Οι datanodes είναι οι εργάτες του συστήματος αρχείων. Αποθηκεύουν και ανακτούν μπλοκ όταν τους αιτηθεί κάτι τέτοιο (είτε από τον namenode, είτε από κάποιον client), και δίνουν αναφορά στον namenode περιοδικά με λίστες από μπλοκ που αποθηκεύουν.

Χωρίς τον namenode, το σύστημα αρχείων δεν μπορεί να χρησιμοποιηθεί. Στην ουσία αν ο namenode χαλάσει, τότε όλα τα αρχεία του συστήματος θα χαθούν, αφού δεν υπάρχει άλλος τρόπος να ξέρουμε πώς να ανακατασκευάσουμε τα αρχεία από τα μπλοκ των datanodes. Για αυτό τον λόγο είναι υψίστης σημασίας να γίνει ο namenode ανθεκτικός σε βλάβες και το Hadoop προσφέρει δύο μηχανισμούς για αυτό τον σκοπό.

Ο πρώτος τρόπος είναι να παρθεί ένα back-up των μεταδεδομένων των αρχείων που αποτελούν το σύστημα αρχείων. Το Hadoop μπορεί να ρυθμιστεί έτσι ώστε να γράφει την κατάσταση του σε πολλαπλά συστήματα αρχείων. Αυτές οι εγγραφές είναι ταυτόχρονες και ατομικές. Συνήθως γράφεται ένας τοπικός καθώς και ένας απομακρυσμένος δίσκος που παίζει τον ρόλο του back-up.

Είναι επίσης δυνατό να τρέξουμε έναν secondary namenode, που παρά την ονομασία του δεν λειτουργεί ως namenode. Ο κύριος ρόλος του είναι να ενώνει το namespace image με το edit log, έτσι ώστε το τελευταίο να μην γίνει υπερβολικά μεγάλο. Ο secondary namenode συνήθως τρέχει σε ένα ξεχωριστό φυσικό υπολογιστή, επειδή απαιτεί καλή επεξεργαστική ισχύ αλλά και μνήμη για να κάνει την δουλειά του. Διατηρεί ένα αντίγραφο του namespace image, το οποίο μπορεί να χρησιμοποιηθεί σε περίπτωση που αποτύχει ο namenode. Παρ' όλα αυτά, σε περίπτωση που ο namenode αποτύχει ολοκληρωτικά (χωρίς να μπορέσει να επανέλθει), η απώλεια δεδομένων είναι σχεδόν βέβαιη. Η δράση που συστήνεται σε αυτή την περίπτωση είναι να αντιγραφούν τα δεδομένα που υπάρχουν στον απομακρυσμένο δίσκο στον secondary namenode και αυτός να οριστεί ως ο κύριος namenode.

Συνήθως ένας datanode διαβάζει μπλοκ από τον δίσκο, αλλά για αρχεία που ζητούνται συχνά τα μπλοκ μπορεί να αποθηκευτούν και σε μια cache τύπου στοίβας στην μνήμη του datanode. Από προεπιλογή, ένα μπλοκ αποθηκεύεται στην cache ενός μόνο datanode. Αυτό όμως μπορεί να αλλαχτεί στις ρυθμίσεις.

Υπάρχει επίσης η δυνατότητα να οριστούν πολλαπλοί namenodes, που όμως είναι υπεύθυνοι για ένα ξεχωριστό υποσύνολο από datanodes. Οι πολλαπλοί datanode είναι ανεξάρτητοι μεταξύ τους και δεν επικοινωνούν, οπότε σε περίπτωση που κάποιος χαλάσει, οι υπόλοιποι δεν επηρεάζονται καθόλου.

2.4 Υψηλή Διαθεσιμότητα

Το HDFS συνδυάζει την αντιγραφή των metadata του namenode σε πολλαπλά συστήματα αρχείων με την χρήση του secondary namenode , που κρατάει στιγμιότυπα του συστήματος αρχείων. Με αυτόν τον τρόπο υπάρχει μια δικλείδα ασφαλείας που αποτρέπει την απώλεια δεδομένων, όμως αυτό δεν προϋποθέτει και την υψηλή διαθεσιμότητα του συστήματος αρχείων. Ο namenode συνεχίζει να αποτελεί ένα μοναδικό σημείο αποτυχίας (SPOF). Αν ο τελευταίος αποτύχει, όλοι οι clients δεν θα είναι σε θέση να διαβάσουν, να γράψουν ή να εμφανίσουν αρχεία, αφού ο namenode είναι το μόνο αποθετήριο των metadata όπως επίσης και της αντιστοίχησης αρχείων σε μπλοκ. Αν συμβεί κάτι τέτοιο, όλος ο Hadoop cluster θα περιέλθει σε μια κατάσταση στην οποία δεν μπορεί να λειτουργήσει εξ' ολοκλήρου μέχρι ένας νέος namenode να είναι διαθέσιμος.

Για την επαναφορά μετά από ένα περιστατικό όπως αυτό που περιγράφεται παραπάνω, ο διαχειριστής του Hadoop cluster δημιουργεί έναν νέο primary namenode με ένα στιγμιότυπο από αυτά που κρατάει ο secondary namenode και ρυθμίζει τους datanodes και τους clients να χρησιμοποιούν αυτόν τον primary namenode. Αυτός ο καινούργιος primary namenode δεν έχει την δυνατότητα να εξυπηρετήσει αιτήσεις από τους clients μέχρι να φορτώσει στην μνήμη το namespace image, να κάνει ένα πέρασμα του edit log από την αρχή και τέλος να έχει λάβει αρκετά block reports από τους datanodes για να εξέλθει από την ασφαλή λειτουργία. Σε μεγάλους clusters για παράδειγμα, με πολλά αρχεία και μπλοκ, η ώρα που χρειάζεται ο namenode μέχρι να είναι πλήρως λειτουργικός μπορεί να φτάσει και τα 30 λεπτά ή και περισσότερο.

Ο μεγάλος χρόνος επαναφοράς αποτελεί αγκάθι και για την προγραμματισμένη συντήρηση. Στην πραγματικότητα, επειδή η απρόσμενη αποτυχία του namenode είναι τόσο σπάνια, η περίπτωση της σχεδιασμένης συντήρησης, κατά την διάρκεια της οποίας ο cluster είναι μη διαθέσιμος, είναι πιο σημαντική στην πράξη.

Το Hadoop 2 διόρθωσε αυτήν την κατάσταση προσθέτοντας υποστήριξη για την υψηλή διαθεσιμότητα του HDFS. Σε αυτήν την έκδοση, υπάρχει ένα ζευγάρι από namenodes σε μία ρύθμιση active-standby, όπου ό ένας από αυτούς είναι ενεργός και ο άλλος περιμένει να πάρει την θέση του ενεργού namenode σε περίπτωση βλάβης χωρίς να υπάρξει αξιοσημείωτη διακοπή υπηρεσιών. Για να μπορεί να συμβεί κάτι τέτοιο, χρειάζεται να γίνουν κάποιες τροποποιήσεις σε επίπεδο αρχιτεκτονικής που περιγράφονται παρακάτω :

Οι namenodes πρέπει να χρησιμοποιούν κοινόχρηστη μνήμη υψηλής διαθεσιμότητας για τον διαμοιρασμό του edit log. Όταν ο standby namenode αναλάβει καθήκοντα, διαβάζει μέχρι το τέλος του edit log για να συγχρονίσει την κατάσταση του με τον active namenode, και έπειτα διαβάζει καινούργιες εγγραφές όπως είναι γραμμένες από τον active namenode.
Οι datanodes πρέπει να στέλνουν αναφορές μπλοκ και στους δύο namenodes, επειδή οι αντιστοιχήσεις μεταξύ μπλοκ και αρχείων αποθηκεύονται στην μνήμη του namenode και όχι στον δίσκο.
Οι clients πρέπει να ρυθμιστούν έτσι ώστε να είναι σε θέση να διαχειριστούν κάποια πιθανή βλάβη του namenode, με έναν τρόπο έτσι ώστε να υπάρχει ενημέρωση και στους χρήστες.
Τα καθήκοντα και οι ευθύνες του secondary namenode αναλαμβάνει ο standby namenode, οποίος παίρνει ανά τακτά χρονικά διαστήματα (σύμφωνα πάντα με τις ρυθμίσεις) στιγμιότυπα του image του συστήματος αρχείων και του edit log στον primary namenode.

Υπάρχουν δύο επιλογές για την κοινόχρηστη μνήμη υψηλής διαθεσιμότητας: ένας NFS filer ή ένας quorum journal manager (QJM). Ο QJM είναι μία υλοποίηση που έχει φτιαχτεί καθαρά για το HDFS, σχεδιασμένη με μοναδικό σκοπό να παρέχει ένα υψηλά διαθέσιμο edit log, και είναι η επιλογή που

συνίσταται για τις περισσότερες εγκαταστάσεις HDFS. Ο QJM τρέχει σαν μια ομάδα από journal nodes,και κάθε τροποποίηση πρέπει να γραφτεί σε μια πλειοψηφία από αυτούς τους κόμβους. Συνήθως, υπάρχουν τρεις journal nodes, έτσι ώστε το σύστημα να μπορεί να διαχειριστεί την απώλεια του ενός από αυτών. Με παρόμοιο τρόπο δουλεύει και το ZooKeeper, αν και είναι σημαντικό να γίνει κατανοητό ότι η υλοποίηση QJM δεν χρησιμοποιεί το ZooKeeper.

Αν ο active namenode αποτύχει, ο standby μπορεί να αναλάβει τον ρόλο του πρώτου πολύ γρήγορα (μερικά δέκατα του δευτερολέπτου) ,επειδή έχει την πιο πρόσφατη κατάσταση διαθέσιμη στην μνήμη : και την τελευταία έκδοση του edit log, αλλά και μία ενημερωμένη αντιστοίχηση των μπλοκ. Ο πραγματικός χρόνος θα είναι μεγαλύτερος (γύρω στο ένα λεπτό), διότι το σύστημα πρέπει να είναι πιο συντηρητικό με την απόφαση για τον αν ο primary namenode, έχει όντως αποτύχει.

Στο απίθανο γεγονός που ο standby namenode είναι μη διαθέσιμος όταν ο active αποτύχει, ο διαχειριστής του cluster μπορεί να εκκινήσει τον standby χειροκίνητα. Ένα τέτοιο σενάριο δεν είναι χειρότερο από την περίπτωση που περιγράφεται παραπάνω, όπου δεν υπάρχει υψηλή διαθεσιμότητα.

Την μετάβαση από τον ενεργό namenode στον standby διαχειρίζεται μια οντότητα του συστήματος που ονομάζεται failover controller. Υπάρχουν πολλοί failover controllers, όμως η προεπιλεγμένη υλοποίηση χρησιμοποιεί το ZooKeeper για να είναι βέβαιο ότι μόνο ένας namenode είναι ενεργός. Κάθε namenode διαθέτει έναν ελαφρύ failover controller, που η δουλειά του είναι να επιτηρεί τον namenode του για βλάβες, χρησιμοποιώντας έναν μηχανισμό «καρδιακού παλμού» κατά τον οποίο ο namenode στέλνει πολύ απλά μηνύματα ανά τακτά διαστήματα για να δηλώσει ότι είναι ακόμη λειτουργικός. Εάν αυτό αργήσει να συμβεί, τότε είναι πολύ πιθανό ο namenode να έχει αποτύχει και έτσι σηματοδοτείται αυτό το γεγονός από τον failover controller.

Αυτή η διαδικασία, η αλλαγή των namenodes από primary σε standby και αντίστροφα μπορεί να γίνει και χειροκίνητα από τον διαχειριστή του συστήματος. Σε μία τέτοια περίπτωση ο failover controller προγραμματίζει την μετάβαση με μία σειρά έτσι ώστε αυτή η αλλαγή να είναι ομαλή και για τους δύο namenodes.

Αν όμως η μετάβαση δεν γίνεται από τον διαχειριστή, αλλά συμβεί αυτόματα, είναι αδύνατο να αποφανθούμε με σιγουριά για το αν ο primary namenode έχει όντως σταματήσει να λειτουργεί. Λόγου χάρη, ένα επιφορτισμένο δίκτυο μπορεί να σηματοδοτήσει την μετάβαση, ακόμη κι αν ο namenode συνεχίζει να τρέχει. Δηλαδή, λόγω του αργού δικτύου μπορεί ο παλμός του namenode να ξεπεράσει το χρονικό όριο που έχει τεθεί να σημαίνει πως ο namenode έχει αποτύχει. Η υλοποίηση της υψηλής διαθεσιμότητας καταβάλλει μεγάλη προσπάθεια για να αποτρέψει τον μέχρι πρότινος ενεργός namenode από το να προκαλέσει ζημιά ή κάποια αλλοίωση με μία μέθοδο που ονομάζεται fencing.

Ο QJM επιτρέπει την εγγραφή στο edit log μόνο από έναν namenode την φορά. Παρ' όλα αυτά ο μέχρι πρότινος ενεργός namenode θα μπορούσε να απαντά ακόμη σε αιτήσεις ανάγνωσης από τους clients (ενώ πλέον δεν είναι ο active namenode). Γι' αυτό ακριβώς τον λόγο, η δημιουργία μίας SSH fencing εντολής, η οποία θα σκοτώνει την διεργασία του namenode είναι μία καλή ιδέα. Αν χρησιμοποιείται NFS filer για το κοινόχρηστο edit log, είναι αναγκαίες ισχυρότερες fencing μέθοδοι. Η χρήση του NFS filer έχει ως αποτέλεσμα να μπορούν και οι δύο namenodes να γράψουν στο edit log ταυτόχρονα (αυτός είναι και ο λόγος που προτείνεται η χρήση του QJM). Το βελησνεκές των fencing μηχανισμών περιλαμβάνει την ανάκληση της πρόσβασης του namenode στο κοινόχρηστο μέσο αποθήκευσης και το κλείσιμο του port μέσω μιας απομακρυσμένης εντολής. Σαν τελευταίο μέτρο, ο μέχρι πρότινος ενεργός namenode μπορεί να αποκλειστεί σε μία τεχνική που αποκαλείται

STONITH (“shoot the other node in the head”), που χρησιμοποιεί μία εξειδικευμένη μονάδα διανομής ισχύος, η οποία απενεργοποιεί «με τη βία» τον υπολογιστή.

2.5 Άδειες Αρχείων στο Hadoop

Στο HDFS υποστηρίζει όλα όσα κανείς θα περίμενε, δηλαδή πράξεις όπως διάβασμα αρχείων, δημιουργία καταλόγων, μεταφορά αρχείων, διαγραφή δεδομένων και προβολή καταλόγων. Ας υποθέσουμε ότι έχουμε στήσει έναν cluster και μέσω της γραμμής εντολών, επιθυμούμε να φτιάξουμε έναν καινούργιο κατάλογο. Αυτό μπορεί να γίνει με την εντολή **hadoop fs –mkdir books**. Στην συνέχεια, για να επιβεβαιώσουμε ότι η εντολή δούλεψε σωστά δίνουμε την εντολή **hadoop fs –ls**. Το αποτέλεσμα της εντολής φαίνεται στην παρακάτω εικόνα

```
Found 2 items
drwxr-xr-x   - tom supergroup          0 2014-10-04 13:22 books
-rw-r--r--   1 tom supergroup        119 2014-10-04 13:21 quangle.txt
```

Εικόνα 2.1 Αποτέλεσμα δημιουργίας καταλόγου και εμφάνισης του

Οι πληροφορίες που εμφανίζονται μοιάζουν πολύ με το αποτέλεσμα που θα βλέπαμε αν δίναμε την εντολή **ls –l**, με κάποιες μικρές τροποποιήσεις. Στην πρώτη στήλη φαίνεται ο τύπος του αρχείου (κατάλογος ή αρχείο). Στην δεύτερη στήλη αναγράφεται ο αριθμός των αντιγράφων που υπάρχουν για το συγκεκριμένο αρχείο (κάτι που τα Unix συστήματα αρχείων δεν διαθέτουν). Βλέπουμε ότι για τους φακέλους αυτό το σημείο είναι κενό, και αυτό συμβαίνει διότι, η ιδέα της αντιγραφής δεν εφαρμόζεται γι’ αυτούς, αφού οι κατάλογοι είναι στην ουσία μεταδεδομένα που αποθηκεύονται στον namenode και όχι στους datanodes. Η τρίτη και η τέταρτη στήλη απεικονίζει τον ιδιοκτήτη και την ομάδα. Η πέμπτη είναι το μέγεθος του αρχείου σε bytes (για τους καταλόγους είναι 0). Η έκτη και η έβδομη στήλη αντιπροσωπεύουν την ώρα και ημερομηνία που τροποποιήθηκε τελευταία φορά το αρχείο. Τέλος, στην όγδοη στήλη είναι το όνομα του αρχείου ή του καταλόγου.

Το HDFS διαθέτει ένα μοντέλο αδειών πρόσβασης για αρχεία και καταλόγους που είναι παρόμοιο με το μοντέλο POSIX. Υπάρχουν τρία είδη αδειών: η άδεια ανάγνωσης (r), η άδεια εγγραφής (w) και η άδεια εκτέλεσης (x). Η άδεια ανάγνωσης απαιτείται για το διάβασμα αρχείων και για την προβολή των περιεχομένων ενός καταλόγου, η άδεια εγγραφής χρειάζεται για το γράψιμο αρχείων και όσον αφορά τους καταλόγους, για τη δημιουργία ή διαγραφή αρχείων μέσα σε αυτούς. Τέλος, η άδεια εκτέλεσης δεν έχει νόημα για τα αρχεία στο HDFS, επειδή δεν είναι δυνατή η εκτέλεση ενός αρχείου (ανόμοια με το μοντέλο POSIX) και για τους καταλόγους αυτή η άδεια είναι αναγκαία για την πρόσβαση στους υποκαταλόγους τους.

Κάθε αρχείο διαθέτει ιδιοκτήτη, ομάδα και κοινόχρηστη πρόσβαση. Οι άδειες του ιδιοκτήτη αφορούν την πρόσβαση που έχει ο ίδιος πάνω στο αρχείο. Οι άδειες της ομάδας αφορούν την πρόσβαση που έχει η ομάδα στην οποία ανήκει ο χρήστης και οι άδειες της κοινόχρηστης πρόσβασης αφορούν τις άδειες όλων των υπόλοιπων χρηστών.

Σαν προεπιλογή, το Hadoop τρέχει με την ασφάλεια απενεργοποιημένη, κάτι που σημαίνει ότι η ταυτότητα του client δεν ελέγχεται. Επειδή οι clients είναι απομακρυσμένοι, είναι δυνατό για έναν client να γίνει ένας αυθαίρετος χρήστης, απλά δημιουργώντας έναν λογαριασμό με το όνομα του στο απομακρυσμένο σύστημα. Αυτό δεν είναι δυνατό αν η ασφάλεια είναι ενεργοποιημένη. Παρ’ όλα αυτά, αξίζει να υπάρχουν οι άδειες πρόσβασης για να αποφευχθούν οι αθέμιτες τροποποιήσεις ή

διαγραφές σημαντικών κομματιών του συστήματος αρχείων από τους χρήστες είτε από αυτοματοποιημένα εργαλεία και προγράμματα.

Όταν ο έλεγχος αδειών είναι ενεργοποιημένος, οι άδειες του ιδιοκτήτη ελέγχονται , αν δηλαδή το όνομα χρήστη του client ταιριάζει με τον ιδιοκτήτη και οι άδειες της ομάδας ελέγχουν αν ο client είναι μέλος της συγκεκριμένης ομάδας. Σε διαφορετική περίπτωση, ελέγχονται οι κοινόχρηστες άδειες.

Επιπρόσθετα, υπάρχει και η έννοια του superuser, που είναι στην ουσία η ταυτότητα της διεργασίας του namenode. Οι έλεγχοι αδειών δεν πραγματοποιούνται για τον superuser.

2.6 Συστήματα Αρχείων στο Hadoop

Το Hadoop έχει μια αφαιρετική έννοια των συστημάτων αρχείων, εκ των οποίων το HDFS είναι μόνο μία υλοποίηση. Υπάρχουν αρκετές διαφορετικές υλοποιήσεις συστημάτων αρχείων, καθένα από τα οποία χρησιμοποιείται για κάποιον ειδικό σκοπό, ανάλογα με τις υποδομές και τις ανάγκες του κάθε cluster.

Η abstract Java κλάση org.apache.hadoop.fs.FileSystem αντιπροσωπεύει τη διεπαφή του client σε ένα σύστημα αρχείων στο Hadoop και υπάρχουν αρκετές ακόμη υλοποιήσεις. Οι κυριότερες από αυτές περιγράφονται παρακάτω στον Πίνακα 2.1

Σύστημα αρχείων	URI σχήμα	Υλοποίηση Java	Περιγραφή
Local	file	fs.LocalFileSystem	Ένα σύστημα αρχείων για έναν τοπικό δίσκο με checksums στην πλευρά του πελάτη. Χρησιμοποιεί το RawLocalFileSystem για τοπικά συστήματα αρχείων χωρίς checksums
HDFS	hdfs	hdfs.DistributedFileSystem	Το κατανεμημένο σύστημα αρχείων του Hadoop.Το HDFS είναι σχεδιασμένο για να δουλεύει σε συνεργασία με το MapReduce
WebHDFS	webhdfs	hdfs.web.WebHdfsFileSystem	Ένα σύστημα αρχείων που παρέχει πρόσβαση για ανάγνωση/εγγραφή στο HDFS μέσω HTTP
Secure WebHDFS	swebhdfs	hdfs.web.SWebHdfsFileSystem	Η HTTPS έκδοση του WebHDFS
HAR	har	fs.HarFileSystem	Ένα σύστημα αρχείων τοποθετημένο σε ένα άλλο σύστημα αρχείων. Τα Hadoop Archives χρησιμοποιούνται για

			να πακετάρουν πολλά αρχεία HDFS σε ένα Archive για εξοικονόμηση μνήμης
FTP	ftp	fs.ftp.FTPFileSystem	Ένα σύστημα αρχείων που υποστηρίζεται από έναν FTP server
S3	s3a	Fs.s3a.S3AFileSystem	Ένα σύστημα αρχείων που υποστηρίζεται από την Amazon S3
Azure	wasb	fs.azure.NativeAzureFileSystem	Ένα σύστημα αρχείων που υποστηρίζεται από την Microsoft Azure
Swift	swift	fs.swift.snative.SwiftNativeFileSystem	Ένα σύστημα αρχείων που υποστηρίζεται από την OpenStack Swift

Πίνακας 2.1 Υποστηριζόμενα συστήματα αρχείων από το Hadoop

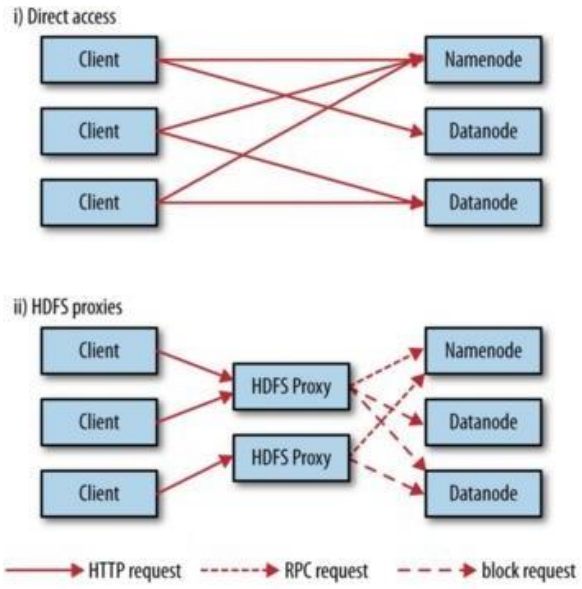
Το Hadoop παρέχει πολλές διεπαφές στα συστήματα αρχείων του και γενικά χρησιμοποιεί το URI σχήμα για να διαλέξει το σωστό στιγμιότυπο συστήματος αρχείων για να επικοινωνήσει. Αν και είναι δυνατό (και κάποιες φορές πολύ βολικό) να τρέξουν MapReduce προγράμματα που έχουν πρόσβαση σε οποιοδήποτε από αυτά τα αρχεία, όταν επεξεργάζονται μεγάλοι όγκοι δεδομένων πρέπει να επιλέγεται ένα κατανεμημένο σύστημα αρχείων, όπως το HDFS.

Το Hadoop είναι γραμμένο σε Java, οπότε οι περισσότερες αλληλεπιδράσεις του συστήματος αρχείων του Hadoop πραγματοποιούνται μέσω του Java API. Το κέλυφος του συστήματος αρχείων είναι μια Java εφαρμογή που χρησιμοποιεί την κλάση FileSystem για να παρέχει τις πράξεις του συστήματος αρχείων (εγγραφή, ανάγνωση, κτλ).

Κάνοντας διαθέσιμο στον έξω κόσμο το σύστημα αρχείων του μέσω του Java API, το Hadoop δυσκολεύει την πρόσβαση σε εφαρμογές που δεν είναι γραμμένες σε Java. Το HTTP REST API που εκτίθεται από το πρωτόκολλο WebHDFS , διευκολύνει την διάδραση με το HDFS για άλλες γλώσσες. Αξίζει να σημειωθεί πως η HTTP διεπαφή είναι πιο αργή από αυτήν της Java, συνεπώς θα πρέπει να αποφεύγεται η χρήση της για πολύ μεγάλες μεταφορές δεδομένων, αν αυτό είναι δυνατό.

Υπάρχουν δύο τρόποι για την πρόσβαση στο HDFS μέσω HTTP: απευθείας, όπου τα HDFS deamons εξυπηρετούν HTTP αιτήσεις από clients και μέσω proxies, που παίρνουν πρόσβαση στο HDFS για λογαριασμό των clients χρησιμοποιώντας το DistributedFileSystem API. Οι δύο αυτοί τρόποι φαίνονται στην εικόνα 2.2 παρακάτω. Και οι δύο χρησιμοποιούν το πρωτόκολλο WebHDFS.

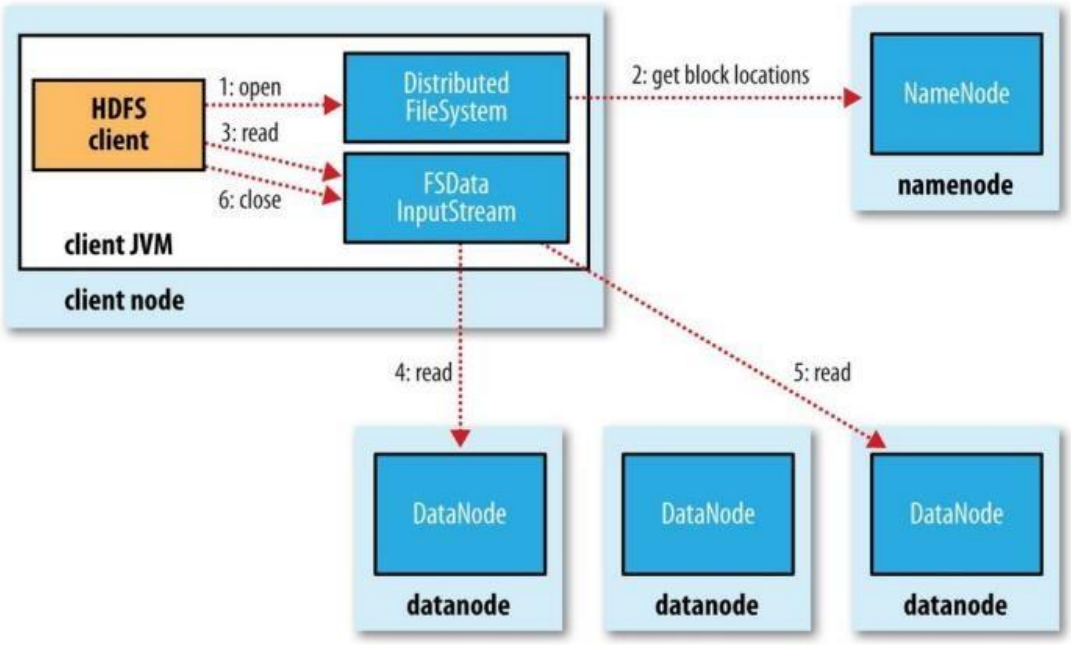
Στην πρώτη περίπτωση (άμεση πρόσβαση) οι servers στον namenode και στους datanodes δρουν ως WebHDFS τερματικά σημεία. Ο χειρισμός των μεταδεδομένων γίνεται από τον namenode, ενώ η ανάγνωση και η εγγραφή σε αρχεία στέλνονται πρώτα στον namenode, ο οποίος ανακατευθύνει τον client, δείχνοντας τον datanode από τον οποίο θα ζητήσει η θα στείλει αρχεία.



Εικόνα 2.2 Πρόσβαση στο HDFS μέσω HTTP απευθείας και μέσω proxy

2.7 Ανάγνωση Αρχείων

Για την καλύτερη κατανόηση της ροής των δεδομένων κατά την διάρκεια ανάγνωσης αρχείων, αυτή η διαδικασία παρουσιάζεται στην παρακάτω εικόνα με όλα τα επιμέρους βήματα που την αποτελούν.



Εικόνα 2.3 Ανάγνωση δεδομένων από το HDFS

Ο client ανοίγει το αρχείο που επιθυμεί να διαβάσει καλώντας την μέθοδο open() στο αντικείμενο FileSystemObject, που για το HDFS είναι του τύπου DistributedFileSystem (βήμα 1). Το DistributedFileSystem καλεί τον namenode, για να αποφασίσει τις τοποθεσίες των πρώτων μπλοκ του αρχείου (βήμα 2). Για κάθε μπλοκ, ο namenode επιστρέφει τις διευθύνσεις των datanodes που έχουν ένα αντίγραφο του συγκεκριμένου μπλοκ. Επιπλέον, οι datanodes ταξινομούνται σύμφωνα με την εγγύτητα τους σε σχέση με τον client (σύμφωνα με την τοπολογία του δικτύου του cluster). Αν ο

client είναι στην ουσία ένας datanode (στην περίπτωση ενός MapReduce task), ο client θα διαβάσει από τον τοπικό datanode αν αυτός έχει ένα αντίγραφο του μπλοκ.

Το DistributedFileSystem επιστρέφει ένα FSDataInputStream (μία ροή δεδομένων που υποστηρίζει αναζητήσεις αρχείων) στον client για να διαβαστούν από αυτό τα δεδομένα. Το FSDataInputStream με τη σειρά του ενθυλακώνει ένα DFSInputStream, το οποίο διαχειρίζεται την E/E (I/O) του namenode και του datanode.

Ο client καλεί την μέθοδο read() (βήμα 3). Το DFSInputStream, που έχει αποθηκεύσει τις διευθύνσεις των datanodes για τα πρώτα μπλοκ του αρχείου. Τα δεδομένα ρέουν από τον datanode πίσω στον πελάτη, που καλεί την read() επανειλημμένως (βήμα 4). Όταν φτάσει το τέλος του αρχείου, το DFSInputStream θα τερματίσει την σύνδεση με τον datanode και θα ψάξει τον καλύτερο datanode για το επόμενο μπλοκ στο αρχείο. (βήμα 5) Ο client από την μεριά του διαβάζει μία συνεχής ροή δεδομένων.

Τα μπλοκ διαβάζονται με τη σειρά, με το DFSInputStream να ανοίγει καινούργιες συνδέσεις με datanodes ενώ ο client διαβάζει την ροή δεδομένων. Θα καλέσει επίσης τον namenode να λάβει τις τοποθεσίες των datanodes για την επόμενη παρτίδα από μπλοκ που χρειάζονται. Όταν ο client τελειώσει, καλεί την μέθοδο close() στο FSDataInputStream (βήμα 6).

Κατά την διάρκεια της ανάγνωσης δεδομένων, αν το DFSInputStream συναντήσει ένα σφάλμα ενώ επικοινωνούσε με έναν datanode, θα δοκιμάσει τον επόμενο πιο κοντινό για αυτό το μπλοκ. Θα θυμάται επίσης τους datanodes που απέτυχαν έτσι ώστε να μην προσπαθήσει ξανά να συνδεθεί σε αυτούς, χάνοντας χρόνο , για επόμενα μπλοκ. Το DFSInputStream επιβεβαιώνει και checksums για τα δεδομένα που μεταφέρονται σε αυτό από τον datanode. Εάν βρεθεί ένα αλλοιωμένο μπλοκ, το DFSInputStream προσπαθεί να διαβάσει ένα αντίγραφο του μπλοκ από άλλο datanode. Επίσης, αναφέρει το αλλοιωμένο μπλοκ στον namenode.

Η αρχιτεκτονική αυτή είναι ζωτικής σημασίας αφού ο client επικοινωνεί με τους datanodes άμεσα για να λάβει δεδομένα και κατευθύνεται από τον namenode στον «καλύτερο» datanode. Αυτή η αρχιτεκτονική επιτρέπει το HDFS να ανταποκριθεί σε πολλούς clients ταυτόχρονα επειδή η κίνηση των δεδομένων κατά μήκος όλων των datanodes του cluster. Εντωμεταξύ, ο namenode θα πρέπει απλά να σερβίρει τις τοποθεσίες των μπλοκ και όχι να σερβίρει δεδομένα, κάτι που θα τον καθιστούσε πολύ σύντομα τον αδύναμο κρίκο του συστήματος καθώς οι clients αυξανόταν.

Αξίζει να κάνουμε μια μικρή αναφορά για το κριτήριο με το οποίο αποφασίζει το Hadoop ποιος είναι ο καλύτερος datanode για να εξυπηρετήσει το αίτημα του client. Σε ότι αφορά την επεξεργασία δεδομένων με μεγάλο όγκο, ο παράγοντας που μας περιορίζει είναι ο ρυθμός με τον οποίο μεταφέρονται δεδομένα μεταξύ κόμβων. Η ιδέα είναι να χρησιμοποιηθεί το εύρος ζώνης ως μέτρο της απόστασης μεταξύ δύο κόμβων.

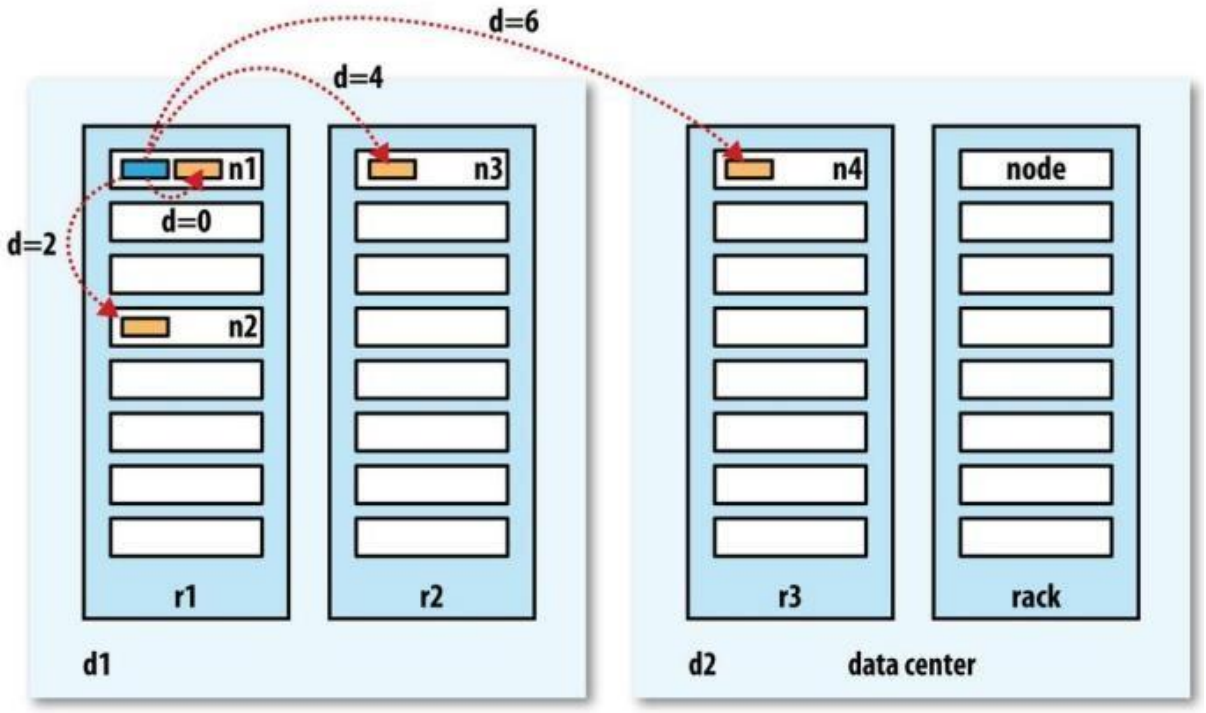
Αντί της μέτρησης του εύρους ζώνης, που μπορεί να είναι δύσκολο στην πράξη, το Hadoop χρησιμοποιεί μια απλούστερη προσέγγιση κατά την οποία το δίκτυο αναπαρίσταται ως ένα δέντρο και η απόσταση μεταξύ δύο κόμβων είναι το άθροισμα των αποστάσεων μέχρι τον πιο κοντινό κοινό τους πρόγονο. Τα επίπεδα του δέντρου δεν είναι προκαθορισμένα, αλλά είναι κοινό να υπάρχουν επίπεδα που συμπίπτουν με το data center, το rack (ράφι με πολλούς υπολογιστές, hardware, κτλ) και τον κόμβο στον οποίο τρέχει μια διεργασία. Το εύρος ζώνης που είναι διαθέσιμο για κάθε ένα από τα παρακάτω σενάρια γίνεται όλο και λιγότερο.

Διεργασίες στον ίδιο κόμβο
Διαφορετικοί κόμβοι στο ίδιο rack

Κόμβοι σε διαφορετικά racks, στο ίδιο data center
Κόμβοι σε διαφορετικά datacenters

Για παράδειγμα, ένας κόμβος n1 στο rack r1 σε ένα data center d1. Αυτό μπορεί να αναπαρασταθεί ως /d1/r1/n1. Χρησιμοποιώντας αυτή την σήμανση, αυτές είναι οι αποστάσεις για τα τέσσερα παραπάνω σενάρια. (βλέπε Εικόνα 2.4)

απόσταση(/d1/r1/n1, /d1/r1/n1) = 0 (1)
απόσταση (/d1/r1/n1, /d1/r1/n2) = 2 (2)
απόσταση (/d1/r1/n1, /d1/r2/n3) = 4 (3)
απόσταση (/d1/r1/n1, /d2/r3/n4) = 6 (4)



Εικόνα 2.4 Απόσταση δικτύου στο Hadoop

Τέλος, θα πρέπει να συνειδητοποιήσουμε ότι το Hadoop δεν μπορεί να ανακαλύψει την τοπολογία του δικτύου του cluster μας ως δια μαγείας. Ως προεπιλογή, όμως , υποθέτει ότι το δίκτυο είναι μονοδιάστατο, δηλαδή όλοι οι κόμβοι είναι σε ένα rack σε ένα data center. Για μικρούς clusters, δεν χρειάζονται περεταίρω ρυθμίσεις.

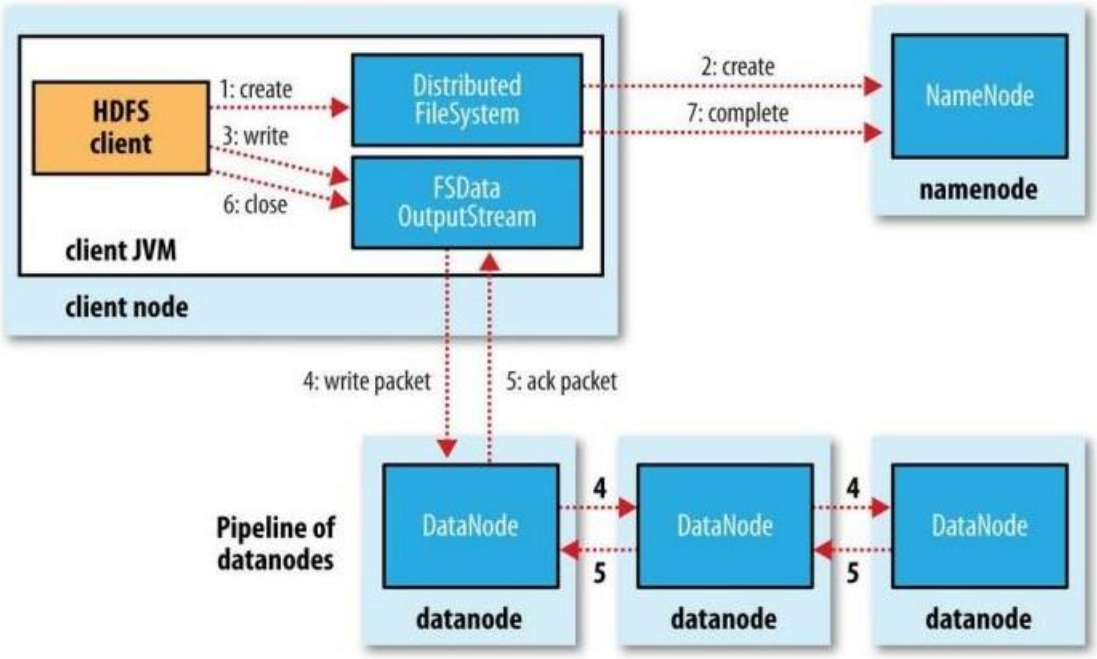
2.8 Εγγραφή Αρχείων

Μετά την ανάγνωση αρχείων είναι σημαντικό θα κατανοήσουμε και τον τρόπο που γίνεται η εγγραφή αρχείων. Σε αυτή την ενότητα θα περιγραφεί η εξής διαδικασία : δημιουργία αρχείου, εγγραφή δεδομένων σε αυτό και τέλος το κλείσιμο του αρχείου. Αυτό φαίνεται και στο Εικόνα 2.5 παρακάτω.

Ο client δημιουργεί ένα αρχείο καλώντας την μέθοδο create() στο DistributedFileSystem (βήμα 1 στο Εικόνα 2.5). Το DistributedFileSystem καλεί με την σειρά του τον namenode για να δημιουργήσει ένα καινούργιο αρχείο στο σύστημα αρχείων, χωρίς να υπάρχει κάποιο μπλοκ που σχετίζεται με αυτό (βήμα 2). Ο namenode πραγματοποιεί αρκετούς ελέγχους για να είναι βέβαιο ότι το αρχείο δεν

υπάρχει ήδη και ότι ο client έχει τις κατάλληλες άδειες πρόσβασης για να δημιουργήσει το αρχείο. Αν αυτοί οι έλεγχοι γίνουν επιτυχώς, ο namenode καταγράφει αυτό το νέο αρχείο. Σε διαφορετική περίπτωση, η δημιουργία αρχείου αποτυγχάνει και στον client εμφανίζεται μήνυμα σφάλματος (IOException). Το DistributedFileSystem επιστρέφει ένα FSDataOutputStream για να ξεκινήσει ο client να γράφει δεδομένα. Όπως και στην περίπτωση της ανάγνωσης, το FSDataOutputStream ενθυλακώνει ένα DFSOutputStream, το οποίο χειρίζεται την επικοινωνία μεταξύ namenode και datanodes. Ενώ ο client γράφει δεδομένα (βήμα 3) το DFSOutputStream χωρίζεται σε πακέτα, τα οποία γράφουν σε μία εσωτερική ουρά που ονομάζεται data queue. Η data queue καταναλώνεται από το DataStreamer, που είναι υπεύθυνο να ζητήσει από τον namenode να εκχωρήσει καινούργια μπλοκ επιλέγοντας μια λίστα από datanodes για να αποθηκευτούν τα αντίγραφα. Η λίστα των datanodes σχηματίζει μια δομή διασωλήνωσης (pipeline) και ας υποθέσουμε ότι υπάρχουν τρεις κόμβοι μέσα σε αυτήν. Το DataStreamer στέλνει πακέτα στον πρώτο datanode, ο οποίος αποθηκεύει κάθε πακέτο και τα προωθεί στον δεύτερο κόμβο. Παρομοίως, ο δεύτερος κόμβος κάνει την ίδια δουλειά για τον τρίτο (βήμα 4).

Το FSDataOutputStream διατηρεί επίσης μία εσωτερική ουρά από πακέτα που περιμένουν να επιβεβαιωθούν, η οποία ονομάζεται ack queue. Ένα πακέτο αφαιρείται από την ack queue μόνο όταν έχει επιβεβαιωθεί από όλους τους datanodes στην διασωλήνωση (βήμα 5).



Εικόνα 2.5 Εγγραφή δεδομένων στο HDFS

Αν κάποιος datanode αποτύχει ενώ εγγράφονται σε αυτόν δεδομένα, τότε λαμβάνονται τα εξής μέτρα. Πρώτον, η διασωλήνωση κλείνει και κάθε πακέτο στην ack queue προστίθεται μπροστά στην data queue έτσι ώστε οι datanodes που έπονται του κόμβου που έχει αποτύχει να μην χάσουν κάποιο πακέτο. Το μπλοκ στους datanodes που είναι σε λειτουργία παίρνει μια νέα ταυτότητα, η οποία στέλνεται στον namenode, έτσι ώστε το μισοτελειωμένο μπλοκ στον datanode που απέτυχε να διαγραφεί αν αυτός ο κόμβος επανέλθει αργότερα. Μία καινούργια διασωλήνωση συγκροτείται από τους δύο εναπομείναντες datanodes. Το υπόλοιπο των δεδομένων του μπλοκ γράφεται στους κόμβους

της νέας διασωλήνωσης. Ο namenode αντιλαμβάνεται ότι το συγκεκριμένο μπλοκ πρέπει να αντιγραφεί περεταίρω κι έτσι κανονίζει για την δημιουργία ενός ακόμη αντιγράφου σε κάποιον άλλο κόμβο.

Όταν ο client τελειώσει με την εγγραφή δεδομένων, καλεί την μέθοδο close() (βήμα 6). Αυτή η ενέργεια απορρίπτει όλα τα εναπομείναντα πακέτα στην διασωλήνωση των datanodes και περιμένει για επιβεβαίωση πριν επικοινωνήσει με τον namenode και να σημάνει ότι το αρχείο έχει ολοκληρωθεί (βήμα 7).

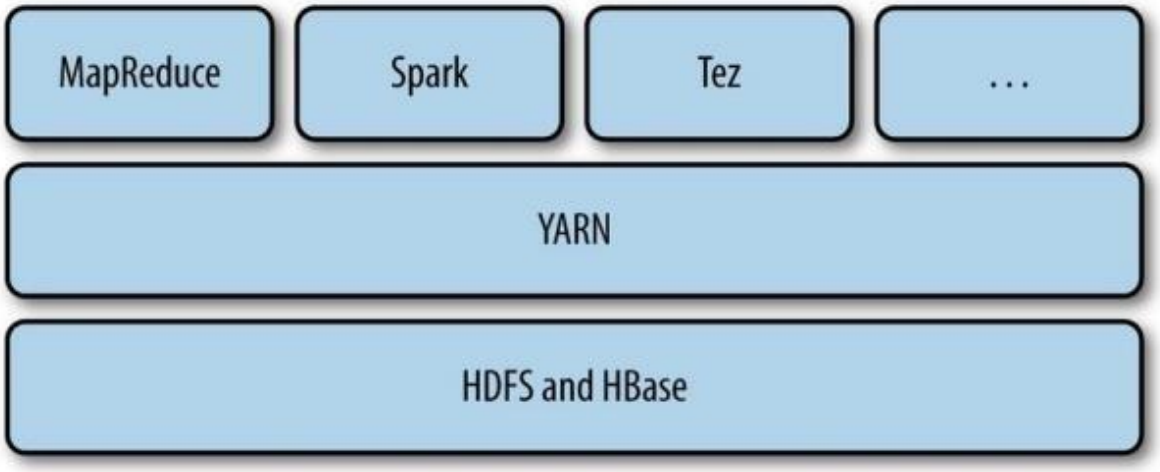
Από όσα αναφέρονται σε αυτή την ενότητα, προκύπτει ένα εύλογο ερώτημα. Πως διαλέγει ο namenode τους κόμβους στους οποίους εγγράφονται τα δεδομένα; Υπάρχει ένα αντάλλαγμα μεταξύ της αξιοπιστίας και του εύρους ζώνης για εγγραφή ή ανάγνωση. Για παράδειγμα, η τοποθέτηση όλων των αντιγράφων σε έναν κόμβο επιφέρει την μικρότερη δυνατή ποινή σε εύρος ζώνης εγγραφής, αλλά δεν προσφέρει κάτι όσον αφορά την εφεδρεία (redundancy). Στο άλλο άκρο, η τοποθέτηση αντιγράφων σε διαφορετικά data centers μπορεί να μεγιστοποιεί την εφεδρεία (redundancy), όμως αυτό γίνεται με το κόστος του εύρους ζώνης. Ακόμη και στο ίδιο data center, υπάρχουν ποικίλες πιθανές στρατηγικές για την τοποθέτηση των αντιγράφων.

Η στρατηγική του Hadoop είναι να τοποθετεί το πρώτο αντίγραφο στον ίδιο κόμβο με τον client(για clients εκτός του cluster, επιλέγεται ένας τυχαίος datanode). Το δεύτερο αντίγραφο τοποθετείται σε διαφορετικό rack από το πρώτο, με τυχαία επιλογή.Το τρίτο αντίγραφο τοποθετείται στο ίδιο rack με το δεύτερο, αλλά σε διαφορετικό κόμβο με τυχαία επιλογή. Τυχόν παραπάνω αντίγραφα τοποθετούνται σε τυχαίους κόμβους στον cluster, αν και το σύστημα προσπαθεί να αποφύγει να τοποθετεί πολλά αντίγραφα στο ίδιο rack.

Κεφάλαιο 3ο: Αναλυτική περιγραφή του YARN

3.1 Εισαγωγή

Το Apache YARN (Yet Another Resource Negotiator) είναι το σύστημα διαχείρισης πόρων του Hadoop. Έκανε την εμφάνιση του στην δεύτερη έκδοση του Hadoop για να βελτιώσει τη διασύνδεση με το MapReduce, αλλά γενικά μπορεί να υποστηρίξει και άλλες υλοποιήσεις κατανεμημένων βιβλιοθηκών για επεξεργασία μεγάλου όγκου δεδομένων, όπως το Spark ή το Tez.



Εικόνα 3.1Επίπεδα ενός κατανεμημένου cluster

Μέχρι το Hadoop 1 η διαχείριση των πόρων καθώς και η χρονοδρομολόγηση γινόταν στο επίπεδο του MapReduce , κάτι που δεν ήταν πρακτικό αφού μια τέτοια υλοποίηση αυξάνει την πολυπλοκότητα ενός ήδη περίπλοκου συστήματος όπως το MapReduce. Με την έλευση του YARN, το MapReduce δρα καθαρά ως το επεξεργαστικό επίπεδο του cluster. Πέρα από αυτό, υπάρχει η δυνατότητα να χρησιμοποιηθούν εναλλακτικές επιλογές αντί του MapReduce, το οποίο είναι γνωστό για την δυσκολία συγγραφής προγραμμάτων.

Υπάρχει άλλο ένα επίπεδο εφαρμογών που έχει σχεδιαστεί να τρέχει πάνω από το MapReduce και το Spark. Αυτό το επίπεδο περιλαμβάνει το Pig, το Hive και το Crunch τα οποία δεν χρειάζεται να αλληλεπιδρούν με το YARN.

Σε αυτό το κεφάλαιο θα αναλυθούν τα χαρακτηριστικά και οι δυνατότητες του YARN, κάτι που θα αποτελέσει τη βάση για την καλύτερη κατανόηση του παρακάτω κεφαλαίου όπου θα περιγραφούν κάποιες από τις βιβλιοθήκες κατανεμημένης επεξεργασίας δεδομένων.

3.2 Η ανατομία του YARN

Το YARN διαθέτει τις κύριες υπηρεσίες του μέσω δύο τύπων δαιμονίων: τον resource manager (ένας ανά cluster) για τη διαχείριση των πόρων κατά μήκος του cluster και τον node manager που τρέχει σε όλους τους κόμβους και είναι υπεύθυνος για την δημιουργία και την επίβλεψη των containers. Ένας container εκτελεί μια διεργασία εξειδικευμένη για κάθε εφαρμογή με περιορισμένους πόρους (μνήμη, πυρήνες επεξεργαστή, κτλ.). Η εικόνα 3.2 απεικονίζει πως το YARN τρέχει μια εφαρμογή.

Για το τρέξιμο μιας εφαρμογής στο YARN, ένας client επικοινωνεί με τον resource manager και του ζητάει να τρέξει μια διεργασία που ονομάζεται application master (βήμα 1). Ο resource manager βρίσκει έναν node manager που μπορεί να δημιουργήσει μια application master διεργασία μέσα σε έναν container (βήματα 2a και 2b). Οι δράσεις της application master εξαρτώνται καθαρά από την εφαρμογή. Θα μπορούσε απλά να κάνει έναν υπολογισμό και να επιστρέψει το αποτέλεσμα στον client. Επίσης θα μπορούσε να ζητήσει περισσότερους containers από τον resource manager (βήμα 3) και να τους χρησιμοποιήσει για την εκτέλεση κατανεμημένων υπολογισμών (βήματα 4a και 4b).

Από την εικόνα 3.2 μπορούμε να εξάγουμε το συμπέρασμα ότι το YARN δεν προσφέρει κάποιον τρόπο για την επικοινωνία των επιμέρους κομματιών που το αποτελούν. Οι περισσότερες αξιοσημείωτες εφαρμογές σε YARN χρησιμοποιούν κάποιας μορφής απομακρυσμένης επικοινωνίας για την ανταλλαγή μηνυμάτων που αφορούν αλλαγές στην κατάσταση των κόμβων, αλλά αυτά είναι συγκεκριμένα ανάλογα με την εφαρμογή.

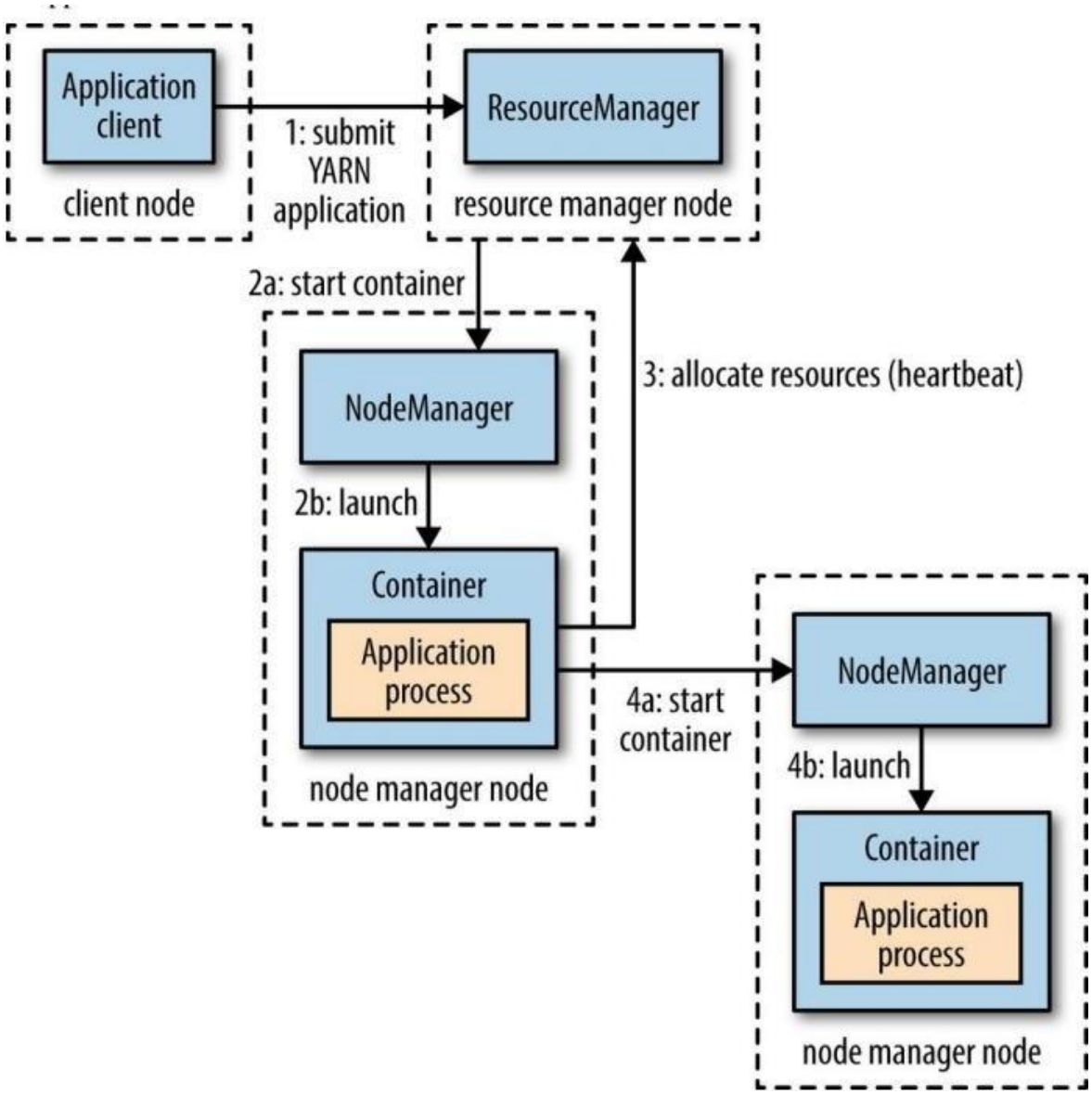
Το YARN έχει ένα ευέλικτο μοντέλο για να κάνει αιτήσεις για πόρους. Μία αίτηση για ένα σετ από containers μπορεί να εκφράζει την ποσότητα υπολογιστικών πόρων που χρειάζονται για κάθε container, καθώς επίσης και περιορισμούς τοπικότητας για τους containers σε αυτή την αίτηση.

Η τοπικότητα είναι κρίσιμη για να είναι σίγουρο ότι οι αλγόριθμοι επεξεργασίας κατανεμημένων δεδομένων χρησιμοποιούν το εύρος ζώνης του cluster αποδοτικά, γι' αυτό και το YARN επιτρέπει σε μία εφαρμογή να ορίσει περιορισμούς τοπικότητας για τους containersπου ζητάει. Αυτοί οι περιορισμοί μπορούν να χρησιμοποιηθούν για την αίτηση ενός container σε ένα συγκεκριμένο κόμβο ή rack, είτε και οπουδήποτε στον cluster.

Στην κοινή περίπτωση που η δημιουργία ενός container για την επεξεργασία ενός μπλοκ του HDFS, η εφαρμογή θα ζητήσει έναν container σε έναν από τους κόμβους που φιλοξενούν τα αντίγραφα του μπλοκ ή σε έναν κόμβο που βρίσκεται σε κάποιο από τα racks όπου υπάρχουν τα αντίγραφα. Σε περίπτωση που οι δυο παραπάνω περιπτώσεις δεν είναι εφικτές το container που θα ζητηθεί μπορεί να βρίσκεται σε οποιονδήποτε κόμβο του cluster.

Μια YARN εφαρμογή μπορεί να κάνει αιτήσεις ανά πάσα στιγμή ενώ τρέχει. Για παράδειγμα, μια εφαρμογή μπορεί να κάνει τις αιτήσεις για containers εκ των προτέρων, ή μπορεί να ζητάει περισσότερους πόρους δυναμικά ανάλογα με τις μεταβαλλόμενες ανάγκες της εφαρμογής.

Το Spark ακολουθεί την πρώτη επιλογή, αρχίζοντας με έναν σταθερό αριθμό από containers στον cluster. Το MapReduce από την άλλη πλευρά, έχει δύο φάσεις. Οι map containers ζητούνται εκ των προτέρων, αλλά οι reduce containers αρχίζουν αργότερα. Επίσης, αν κάποιες εργασίες αποτύχουν, επιπλέον containers θα ζητηθούν έτσι ώστε να εκτελεστούν ξανά αυτές που απέτυχαν.



Εικόνα 3.2 Εκτέλεση μιας εφαρμογής από το YARN

Η διάρκεια ζωής μίας YARN εφαρμογής μπορεί να διαφέρει σε μεγάλο βαθμό ανάλογα με την περίσταση. Από την μία πλευρά, μπορεί μία βραχύχρονη εφαρμογή να εκτελεστεί μέσα σε μερικά δευτερόλεπτα και από την άλλη ενδέχεται μία εφαρμογή να τρέχει για ολόκληρες μέρες ή και σε ακραίες περιπτώσεις για μήνες. Μία πιο ωφέλιμη κατηγοριοποίηση των εφαρμογών YARN θα ήταν όχι βάσει του χρόνου εκτέλεσης, αλλά βάσει του τρόπου με τον οποίο αντιστοιχίζονται με τις διεργασίες που τρέχουν οι χρήστες. Η απλούστερη προσέγγιση είναι μία εφαρμογή ανά διεργασία, που είναι και η προσέγγιση που ακολουθεί και το MapReduce.

Η δεύτερη προσέγγιση περιλαμβάνει την εκτέλεση μίας εφαρμογής ανά συνεδρία χρήστη που πιθανώς αφορά μη σχετικές διεργασίες. Αυτή η προσέγγιση μπορεί να είναι πιο αποδοτική από την πρώτη, αφού οι containers μπορούν να επαναχρησιμοποιηθούν από τις διεργασίες. Επίσης υπάρχει η δυνατότητα να αποθηκευτούν στην cache ενδιάμεσα δεδομένα ανάμεσα στις διεργασίες. Το Spark είναι ένα παράδειγμα αυτού του μοντέλου.

Για να είμαστε ακριβείς, υπάρχει και μία τρίτη προσέγγιση για μακροσκελείς διεργασίες που διαμοιράζονται σε πολλούς χρήστες ταυτόχρονα. Παρ' όλα αυτά δεν θα αναφερθούμε περεταίρω σχετικά με αυτήν.

Μέχρι τώρα έχουμε αναλύσει τα επί μέρους τμήματα μιας YARN εφαρμογής. Δεν έχει αναφερθεί κάτι για τον τρόπο κατασκευής της. Γι' αυτό το λόγο, αξίζει να αναφέρουμε μερικά πράγματα.

Η συγγραφή μίας YARN εφαρμογής είναι μια αρκετά περίπλοκη διαδικασία που πολλές φορές δεν είναι καν απαραίτητη, μιας και συχνά είναι δυνατό να χρησιμοποιηθεί μία ήδη υπάρχουσα εφαρμογή. Για παράδειγμα, για την εκτέλεση ενός κατευθυνόμενου άκυκλου γράφου από διεργασίες (DAG), τότε το Spark ή το Tez είναι κατάλληλα. Όσον αφορά την επεξεργασία ροής δεδομένων συστήνεται το Spark ή το Storm.

Υπάρχουν κάποια projectsπου διευκολύνουν αρκετά την διαδικασία συγγραφής YARN εφαρμογών. Ένα από αυτά είναι το Apache Slider, το οποίο δίνει την δυνατότητα εκτέλεσης ήδη υπαρχόντων κατανεμημένων εφαρμογών στο YARN. Οι χρήστες μπορούν να τρέξουν τα δικά τους στιγμιότυπα μίας εφαρμογής σε έναν cluster, ανεξάρτητα από τις ενέργειες των άλλων χρηστών, οι οποίοι μπορούν να τρέχουν διαφορετικές εκδόσεις της ίδιας εφαρμογής. Το Slider παρέχει αρκετές δυνατότητες σε σχέση με τον έλεγχο αυτής της διαδικασίας. Πιο συγκεκριμένα, μπορεί να τροποποιηθεί ο αριθμός των κόμβων που τρέχουν την εφαρμογή, καθώς επίσης να ανασταλεί και να συνεχιστεί η εκτέλεση μίας εφαρμογής.

Το Apache Twill είναι παρόμοιο με το Slider που επιπρόσθετα προσφέρει ένα απλό προγραμματιστικό μοντέλο για την ανάπτυξη κατανεμημένων εφαρμογών στο YARN. Επίσης επιτρέπει τον καθορισμό διεργασιών ως μια επέκταση του Java Runnable, που στην συνέχεια εκτελούνται σε YARN containers του cluster.

Στην περίπτωση κατά την οποία αυτές οι επιλογές δεν είναι αρκετές, δηλαδή αν η εφαρμογή έχει περίπλοκες απαιτήσεις χρονοδρομολόγησης, τότε η κατανεμημένη εφαρμογή κελύφους που είναι μέρος του YARN, αποτελεί παράδειγμα του τρόπου συγγραφής μίας YARN εφαρμογής. Καταδεικνύει το πώς χρησιμοποιείται το API του YARN client για την διαχείριση της επικοινωνίας ανάμεσα στον client ή στον application master και στις YARN διεργασίες υπηρεσιών

3.3 Χρονοδρομολόγηση στο YARN

Σε ένα ιδανικό σενάριο, οι αιτήσεις που κάνει μια YARN εφαρμογή θα εκπληρώνονταν αμέσως. Παρ' όλα αυτά, σε ρεαλιστικές συνθήκες οι πόροι είναι περιορισμένοι και σε έναν φορτωμένο cluster, μια εφαρμογή θα χρειαστεί να περιμένει μέχρι να της ανατεθούν οι απαραίτητοι πόροι για να εκτελεστεί. Μέσα στα καθήκοντα της χρονοδρομολόγησης του YARN, είναι η ανάθεση των υπολογιστικών πόρων σύμφωνα με μία προκαθορισμένη πολιτική. Η χρονοδρομολόγηση είναι ένα σύνθετο ζήτημα στο οποίο δεν υπάρχει μία και μοναδική λύση που χρησιμοποιείται σαν πανάκεια. Γι' αυτό το λόγο το YARN προσφέρει διάφορες επιλογές και ρυθμίσεις. Παρακάτω θα δούμε κάποιες από αυτές.

Υπάρχουν τρία είδη χρονοδρομολόγησης που είναι διαθέσιμα στο YARN, η FIFO, η Capacity και οι Fair χρονοδρομολόγηση.

Η FIFO χρονοδρομολόγηση έχει το πλεονέκτημα ότι είναι απλή στην κατανόηση της και στο ότι δεν χρειάζεται περεταίρω ρυθμίσεις, όμως δεν είναι κατάλληλη για κοινόχρηστους clusters. Οι μεγάλες

εφαρμογές θα χρησιμοποιήσουν όλους τους πόρους, κι έτσι κάθε εφαρμογή θα πρέπει να περιμένει τη σειρά της. Σε έναν κοινόχρηστο cluster είναι καλύτερο να χρησιμοποιηθεί η Fair ή η Capacity χρονοδρομολόγηση. Και οι δύο προαναφερθείσες χρονοδρομολογήσεις επιτρέπουν την ολοκλήρωση μακροπρόθεσμων διεργασιών σε λογικά χρονικά πλαίσια, ενώ ταυτόχρονα εκτελούνται και σχετικά μικρότερες και πιο απλές εργασίες που έχουν αιτηθεί οι χρήστες.

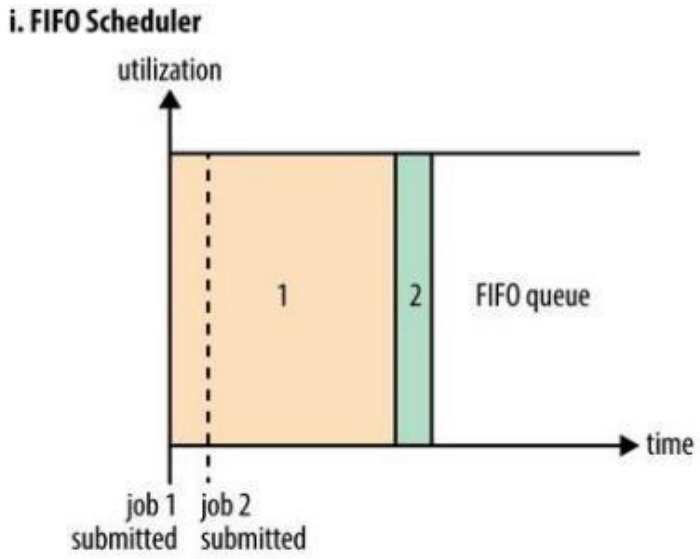
Η διαφορά μεταξύ των χρονοδρομολογήσεων φαίνονται στις εικόνες 3.3, 3.34, 3.5 παρακάτω, όπου παρατηρείται ότι στην FIFO η μικρή διεργασία μπλοκάρεται μέχρι να ολοκληρωθεί η μεγάλη διεργασία.

Με την Capacity, μια ξεχωριστή ουρά δίνει την ευκαιρία στην μικρή διεργασία να ξεκινήσει αμέσως μετά την κατάθεση της. Αυτό όμως έχει κάποιο κόστος όσον αφορά την χρησιμοποίηση του cluster, αφού η χωρητικότητα της ουράς κρατείται για διεργασίες μόνο της συγκεκριμένης ουράς. Δηλαδή, οι μεγάλη διεργασία θα τελειώσει αργότερα σε σύγκριση με την FIFO χρονοδρομολόγηση.

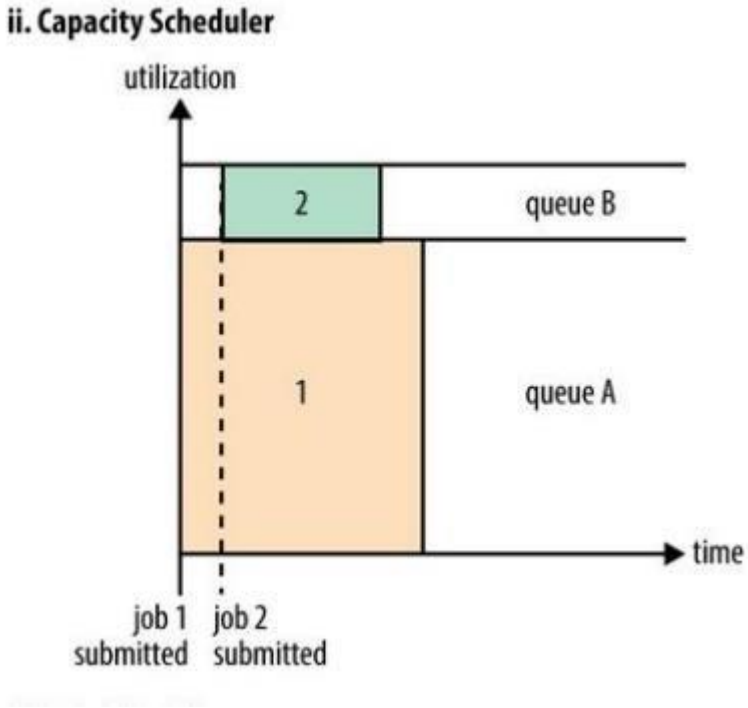
Με την Fair, δεν χρειάζεται η κράτηση μίας προκαθορισμένης χωρητικότητας, επειδή οι πόροι θα καταμεριστούν δυναμικά σε όλες τις εκτελούμενες διεργασίες. Λίγο μετά την εκκίνηση της μεγάλης διεργασίας, αυτή είναι η μόνη που τρέχει κι έτσι παίρνει όλους τους πόρους του cluster. Όταν ξεκινήσει η δεύτερη διεργασία, της ανατίθενται οι μισοί διαθέσιμοι πόροι έτσι ώστε κάθε διεργασία να χρησιμοποιεί το δίκαιο μέρος των πόρων που της αναλογεί.

Αξίζει να σημειωθεί ότι υπάρχει μια καθυστέρηση μεταξύ της εκκίνησης της δεύτερης διεργασίας και της ανάθεσης του μέρους των πόρων που της αναλογούν. Αυτό συμβαίνει διότι, πρέπει να ελευθερωθούν οι containers με τους πόρους που χρησιμοποιούνται από την άλλη διεργασία. Μετά την ολοκλήρωση της μικρής διεργασίας, η μεγάλη διεργασία επιστρέφει στη χρησιμοποίηση όλων των πόρων του cluster. Η χρήση αυτού του είδους της χρονοδρομολόγησης επιφέρει τα εξής αποτελέσματα: υψηλή αξιοποίηση του cluster και εκτέλεση των διεργασιών σε σχετικά σύντομο χρονικό διάστημα.

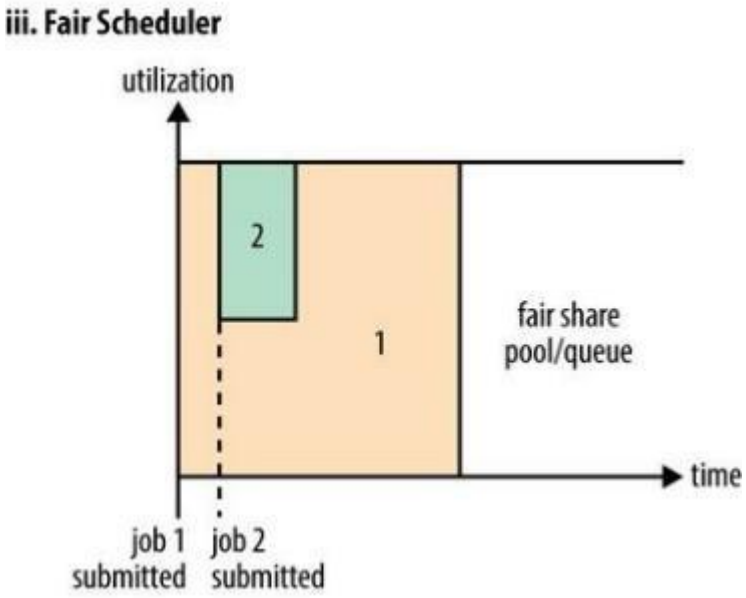
Οι παρακάτω εικόνες τονίζουν τις διαφορές που υπάρχουν στα τρία είδη των χρονοδρομολογήσεων. Η Capacity και η Fair απαιτούν κάποιες πιο σύνθετες ρυθμίσεις που θα εξετάσουμε στην συνέχεια.



Εικόνα 3.3 Χρονοδρομολόγηση FIFO



Εικόνα 3.4 Χρονοδρομολόγηση Capacity



Εικόνα 3.5 Χρονοδρομολόγηση Fair

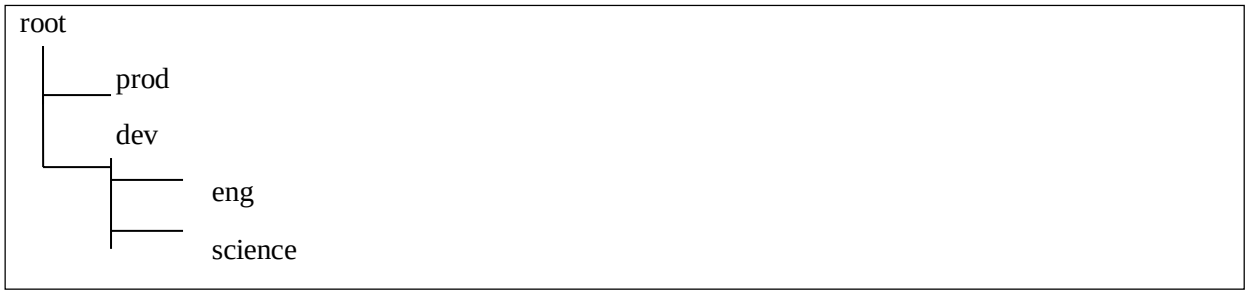
3.4 Capacity Χρονοδρομολόγηση

Η Capacity χρονοδρομολόγηση επιτρέπει τον διαμοιρασμό ενός Hadoop cluster σε διάφορα τμήματα ανάλογα με την ιεραρχία που θα καθοριστεί, όπου κάθε τμήμα παίρνει ένα μέρος του συνολικού cluster. Επίσης θα τμήμα παίρνει και μια ειδικά αφιερωμένη ουρά σε αυτό. Οι ουρές μπορεί να διαχωριστούν περεταίρω με ιεραρχικό τρόπο, επιτρέποντας κάθε τμήμα να μπορεί να μοιραστεί τους πόρους που του αναλογούν μεταξύ διαφορετικών ομάδων χρηστών. Μέσα σε μία ουρά, οι εφαρμογές χρονοδρομολογούνται χρησιμοποιώντας την FIFO.

Όπως είδαμε και στην εικόνα 3.4, μία διεργασία δεν χρησιμοποιεί περισσότερους πόρους από των χωρητικότητα της ουράς της. Παρ' όλα αυτά, αν υπάρχουν παραπάνω από μία διεργασίες σε μία ουρά και υπάρχουν αδρανείς πόροι διαθέσιμοι στην ουρά, τότε η Capacity χρονοδρομολόγηση μπορεί να αναθέσει τους πόρους που περισσεύουν σε διεργασίες στην ουρά, ακόμη κι αν αυτό έχει ως αποτέλεσμα την υπέρβαση της χωρητικότητας της ουράς. Αυτή η συμπεριφορά είναι γνωστή ως ελαστικότητα ουράς.

Υπό κανονικές συνθήκες, η Capacity χρονοδρομολόγηση δεν αδειάζει containers με τη βία (ενώ ακόμη τρέχουν). Έτσι, αν μία ουρά είναι σχετικά άδεια λόγω χαμηλής ζήτησης και στην συνέχεια η ζήτηση αυξηθεί, η ουρά θα επιστρέψει σε πλήρη χωρητικότητα , καθώς οι πόροι απελευθερώνονται από άλλες ουρές. Είναι δυνατό κάτι τέτοιο να μετριαστεί ρυθμίζοντας τις ουρές με μία μέγιστη χωρητικότητα έτσι ώστε να μην κλέβουν τους πόρους από άλλες ουρές τόσο πολύ. Αυτό γίνεται με κόστος όσον αφορά την ελαστικότητα της ουράς, οπότε μια λογική ανταλλαγή πρέπει να βρεθεί με δοκιμές και διορθώσεις.

Ας φανταστούμε την εξής ιεραρχία :



Εικόνα 3.6 Ιεραρχία μίας ουράς σε χρονοδρομολόγηση Capacity

Παρακάτω παρατίθεται ένα δείγμα ενός αρχείου ρύθμισης της Capacity χρονοδρομολόγησης. Ορίζει δύο ουρές κάτω από την root ουρά, την prod και την dev, οι οποίες έχουν το 40% και το 60% της χωρητικότητας αντίστοιχα

```
<configuration>
      <property>
            <name>yarn.scheduler.capacity.root.queues</name>
            <value>prod,dev</value>
      </property>
      <property>
            <name>yarn.scheduler.capacity.root.dev.queues</name>
            <value>eng,science</value>
      </property>
      <property>
            <name>yarn.scheduler.capacity.root.prod.capacity</name>
            <value>40</value>
      </property>
      <property>
            <name>yarn.scheduler.capacity.root.dev.capacity</name>
            <value>60</value>
      </property>
      <property>
            <name>yarn.scheduler.capacity.root.dev.maximumcapacity</name>
            <value>75</value>
```

```
      </property>
      <property>
            <name>yarn.scheduler.capacity.root.dev.eng.capacity</name>
            <value>50</value>
      </property>
      <property>
            <name>yarn.scheduler.capacity.root.dev.science.capacity</name>
            <value>50</value>
      </property>
</configuration>
```

Κώδικας 3.1 Αρχείο ρύθμισης της χρονοδρομολόγησης Capacity

Όπως βλέπουμε παραπάνω, η dev ουρά χωρίζεται στην eng και την science ουρά με ίδια χωρητικότητα. Για να μην χρησιμοποιήσει η dev ουρά όλους τους πόρους όταν η prod ουρά είναι αδρανής, έχει ρυθμιστεί η μέγιστη χωρητικότητα στο 75%. Με άλλα λόγια, η prod ουρά έχει πάντα το 25% των πόρων άμεσα διαθέσιμο για χρήση. Αφού δεν έχουν ρυθμιστεί μέγιστες χωρητικότητες για άλλες ουρές, είναι δυνατό για άλλες διεργασίες στις eng και science ουρές να χρησιμοποιήσουν όλη την χωρητικότητα της dev ουράς (μέχρι και 75% του cluster), ή για την prod ουρά να χρησιμοποιήσει τον cluster εξ ολοκλήρου

3.5 Fair Χρονοδρομολόγηση

Η Fair χρονοδρομολόγηση προσπαθεί να εκχωρήσει τους πόρους έτσι ώστε όλες οι εκτελούμενες εφαρμογές να πάρουν το ίδιο μερίδιο πόρων. Η εικόνα 3.5 δείχνει πως η Fair χρονοδρομολόγηση δουλεύει για εφαρμογές στην ίδια ουρά. Ωστόσο, η Fair δουλεύει και μεταξύ ουρών, όπως θα δούμε στη συνέχεια.

Για να κατανοήσουμε καλύτερα πως οι πόροι μοιράζονται μεταξύ ουρών, ας φανταστούμε δύο χρήστες, τον Α και τον Β, ο καθένας από τους οποίους έχει την δική του ουρά. Ο Α ξεκινά μια διεργασία και του δεσμεύει όλους τους πόρους που είναι διαθέσιμοι, αφού δεν υπάρχει ζήτηση από τον Β. Στη συνέχεια ο Β ξεκινά μία διεργασία ενώ η διεργασία του Α ακόμη τρέχει, και μετά από λίγο κάθε διεργασία χρησιμοποιεί τους μισούς πόρους, όπως είδαμε προηγουμένως. Τώρα αν ο Β ξεκινήσει μία δεύτερη διεργασία ενώ οι άλλες διεργασίες τρέχουν, θα μοιραστεί τους πόρους του με την άλλη διεργασία που τρέχει ο Β, κι έτσι κάθε διεργασία του Β θα έχει το ένα τέταρτο των πόρων, ενώ η διεργασία του Α θα έχει το υπόλοιπο μισό. Ως αποτέλεσμα οι πόροι μοιράζονται δίκαια στους χρήστες.

Η Fair χρονοδρομολόγηση ρυθμίζεται με ένα αρχείο εκχώρησης (allocation file). Σε περίπτωση απουσίας του αρχείου εκχώρησης, η Fair χρονοδρομολόγηση λειτουργεί όπως περιγράφηκε νωρίτερα: κάθε εφαρμογή τοποθετείται σε μία ουρά που ονομάζεται από τον χρήστη και οι ουρές δημιουργούνται δυναμικά όταν οι χρήστες καταθέσουν την πρώτη τους εφαρμογή.

Η ρύθμιση της κάθε ουράς διαφαίνεται στον αρχείο εκχώρησης. Αυτό επιτρέπει την ρύθμιση ιεραρχικών ουρών όπως αυτές που υποστηρίζονται από την Capacity χρονοδρομολόγηση. Για παράδειγμα, μπορούμε να ορίσουμε prod και dev ουρές όπως έγινε με την Capacity χρονοδρομολόγηση, χρησιμοποιώντας αρχεία εκχώρησης όπως το παρακάτω στο Παράδειγμα 4.2.

```
<allocations>
   <defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>
   <queue name=”prod”>
```

```
      <weight>40<weight>
      <schedulingPolicy>fifo</schedulingPolicy>
   </queue>
       <queue name="dev">
      <weight>60</weight>
      <queue name="eng"/>
      <queue name="science"/>
  </queue>
  <queuePlacementPolicy>
      <rule name="specified" create="false">
      rule name="primaryGroup" create="false">
      rule name="default" queue="dev.eng">
  </queuePlacementPolicy>
</allocation>
```

Κώδικας 3.2 Δημιουργία ιεραρχικών ουρών στη Fair χρονοδρομολόγηση

Η ιεραρχία ουράς ορίζεται με εμφωλευμένα queue στοιχεία. Όλες οι ουρές είναι παιδιά της root ουράς. Εδώ, διαχωρίζουμε την dev ουρά σε μία ουρά που ονομάζεται eng και σε άλλη μία που λέγεται science.

Οι ουρές έχουν βάρη, τα οποία χρησιμοποιούνται για τον υπολογισμό του δίκαιου διαμοιρασμού. Σε αυτό το παράδειγμα, η ανάθεση των πόρων θεωρείται δίκαια αν και διαχωρίζεται σε μερίδια 40:60 μεταξύ του prod και του dev.Οι eng και science ουρές δεν έχουν καθορισμένα βάρη, οπότε διαχωρίζονται στη μέση. Τα βάρη δεν είναι ακριβώς ίδια με τα ποσοστά, ακόμη κι αν οι αριθμοί που χρησιμοποιούνται αθροίζουν το 100. Αυτό γίνεται για λόγους απλότητας. Θα μπορούσαμε να είχαμε ορίσει τα βάρη ως 2 και 3 στις αντίστοιχες ουρές και το αποτέλεσμα θα ήταν ακριβώς το ίδιο.

Οι ουρές μπορούν να έχουν διαφορετικές πολιτικές χρονοδρομολόγησης. Η προκαθορισμένη πολιτική για ουρές μπορεί να οριστεί με το στοιχείο `defaultQueueSchedulingPolicy` και αν αυτό παραλειφθεί, τότε χρησιμοποιείται η Fair χρονοδρομολόγηση. Ανεξαρτήτως ονόματος η Fair χρονοδρομολόγηση χρησιμοποιεί επίσης την πολιτική FIFO σε ουρές, αλλά και την Dominant Resource Fairness, η οποία δεν θα περιγραφεί στα πλαίσια αυτής της εργασίας.

Η πολιτική για μία συγκεκριμένη ουρά μπορεί να υπερκαλυφθεί χρησιμοποιώντας το στοιχείο schedulingPolicy. Στο παραπάνω παράδειγμα, η prod ουρά χρησιμοποιεί FIFO αφού θέλουμε κάθε διεργασία να τρέχει σειριακά και να ολοκληρώνεται στον ελάχιστο δυνατό χρόνο. Να αναφερθεί ότι ο δίκαιος διαμοιρασμός συνεχίζει να χρησιμοποιείται για τον διαχωρισμό τον πόρων μεταξύ των ουρών dev και prod, όπως επίσης και μεταξύ των ουρών eng και science.

Αν και δεν φαίνεται σε αυτό το αρχείο εκχώρησης, οι ουρές μπορούν να ρυθμιστούν με ελάχιστους και μέγιστους πόρους και με μέγιστο αριθμό εκτελούμενων εφαρμογών. Οι ελάχιστοι πόροι δεν είναι ένα αυστηρό όριο, αλλά χρησιμοποιείται από την χρονοδρομολόγηση για να αποφασιστεί η προτεραιότητα των αναθέσεων των πόρων. Αν δύο ουρές είναι κάτω από το μερίδιο τους, τότε αυτή που είναι πιο κάτω σε σχέση με την ελάχιστη εκχώρηση πόρων, παίρνει πρώτη πόρους.

Έχοντας περιγράψει το HDFS και το YARN, σειρά έχει το επεξεργαστικό επίπεδο, στο οποίο μπορούμε να βρούμε ένα από τα MapReduce και Spark τα όποια θα περιγραφούν στα πλαίσια αυτής της εργασίας, είτε κάποια άλλη βιβλιοθήκη όπως το Tez.

Κεφάλαιο 4ο: Αναλυτική περιγραφή MapReduce και μια μάτια στο Spark

4.1 Εισαγωγή

Το MapReduce είναι ένα προγραμματιστικό μοντέλο για επεξεργασία δεδομένων. Αυτό το μοντέλο υποστηρίζει πολλές γλώσσες προγραμματισμού στις οποίες μπορούν να γραφούν προγράμματα. Μερικές από αυτές είναι η Java, η Ruby και η Python. Τα προγράμματα αυτά είναι εγγενώς παράλληλα, κι έτσι κάνουν την ανάλυση δεδομένων μεγάλου όγκου προσβάσιμη σε οποιονδήποτε διαθέτει αρκετούς υπολογιστές για αυτό τον σκοπό.

4.2 Ροή δεδομένων στο MapReduce

Ας ξεκινήσουμε με την ορολογία. Μια MapReduce job είναι μία μονάδα εργασίας που επιθυμεί να πραγματοποιήσει ο client. Αποτελείται από τα δεδομένα εισόδου, το πρόγραμμα MapReduce και τις πληροφορίες ρύθμισης (configuration).Το Hadoop εκτελεί αυτή την διεργασία χωρίζοντας την σε μικρότερα κομμάτια που λέγονται tasks. Υπάρχουν δύο είδη από tasks, τα map tasks και τα reduce tasks. Αυτά χρονοδρομολογούνται με τη χρήση του YARN και εκτελούνται σε κόμβους του cluster. Αν ένα task αποτύχει, θα δρομολογηθεί ξανά αυτόματα σε κάποιον άλλο κόμβο.

Το Hadoop διαχωρίζει τα δεδομένα εισόδου του MapReduce σε κομμάτια συγκεκριμένου μεγέθους, που ονομάζονται splits. Το Hadoop δημιουργεί ένα Map task για κάθε split, το οποίο τρέχει την συνάρτηση Map του προγράμματος για κάθε εγγραφή του split.

Αν έχουμε πολλά splits, αυτό σημαίνει πως ο χρόνος που χρειάζεται για την επεξεργασία του καθενός από αυτά είναι σαφώς μικρότερος από τον χρόνο θα χρειαζόταν για να επεξεργαστεί ολόκληρο το σετ των δεδομένων εισόδου. Επομένως, αν η επεξεργασία των splits γίνεται παράλληλα, ο φόρτος της επεξεργασίας μοιράζεται καλύτερα σε splits που είναι μικρά, αφού ένας γρηγορότερος υπολογιστής θα είναι σε θέση να υπολογίσει αναλογικά περισσότερα splits κατά την διάρκεια ενός job σε σύγκριση με έναν πιο αργό υπολογιστή. Ακόμη και αν οι υπολογιστές είναι παρόμοιοι, οι αποτυχημένες διεργασίες τρέχουν ταυτόχρονα και κάνουν το μοίρασμα του φόρτου επιθυμητό, αλλά και η ποιότητα του τελευταίου αυξάνεται καθώς τα job γίνονται όλο και πιο λεπτομερή.

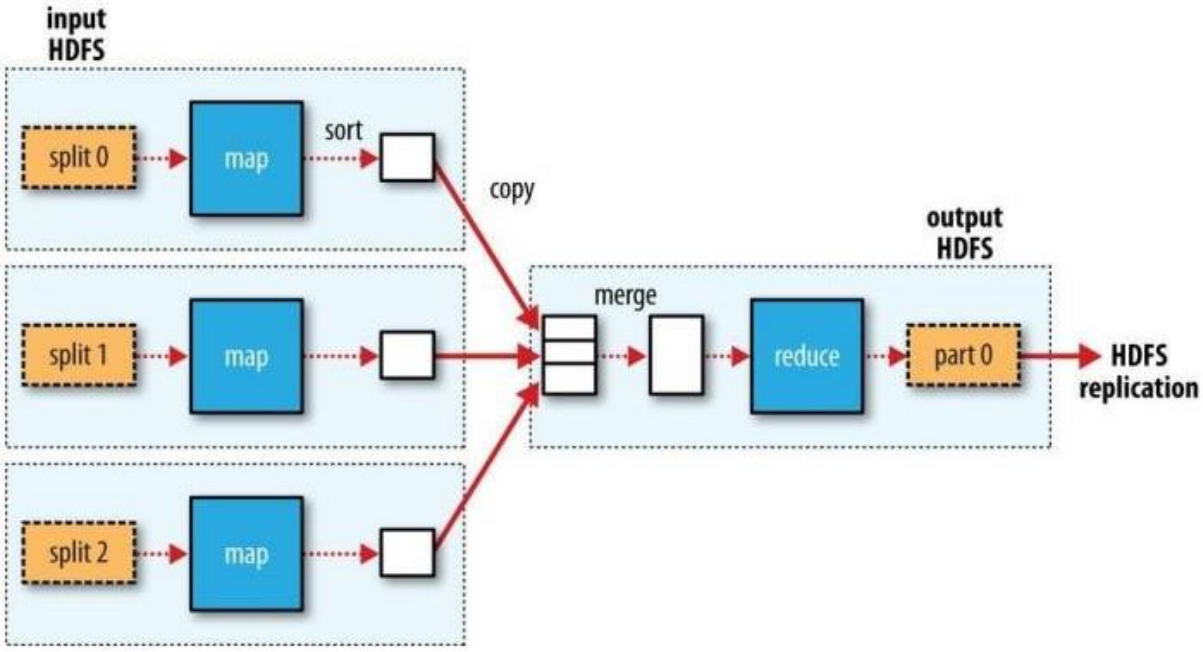
Από την άλλη πλευρά, αν τα splits είναι υπερβολικά μικρά, η καθυστέρηση της διαχείρισης των splits και η δημιουργία του map task αρχίζουν να κυριαρχούν τον συνολικό χρόνο εκτέλεσης. Για τα περισσότερα jobs, ένα καλό μέγεθος split τείνει να είναι 128 MB, όσο δηλαδή και το μέγεθος ενός HDFS block. Παρ' όλα αυτά το μέγεθος του split μπορεί να αλλάξει ή να προσδιορίζεται όταν δημιουργείται κάθε αρχείο.

Το Hadoop προσπαθεί να τρέχει κάθε Map taskστον κόμβο όπου βρίσκονται τα δεδομένα εισόδου στο HDFS, διότι δεν χρησιμοποιεί το πολύτιμο εύρος ζώνης. Αυτό ονομάζεται **βελτιστοποίηση τοπικότητας δεδομένων**. Κάποιες φορές όμως, όλοι οι κόμβοι που φιλοξενούν τα HDFS αντίγραφα του split κάποιου Map task, εκτελούν άλλα tasks. Έτσι, ο χρονοδρομολογητής θα ψάξει μια ελεύθερη θέση σε έναν κόμβο στο ίδιο rack με ένα από τα μπλοκ.

Τα Map tasks γράφουν το αποτέλεσμά τους στον τοπικό δίσκο και όχι στο HDFS. Αυτό συμβαίνει γιατί αυτό είναι ένα ενδιάμεσο αποτέλεσμα το ποίο πρέπει να επεξεργαστεί από κάποιο Reduce task για την εξαγωγή του τελικού αποτελέσματος και μόλις αυτό συμβεί, το αποτέλεσμα της Map μπορεί να απορριφθεί. Αν ο κόμβος που τρέχει το Map task αποτύχει πριν το αποτέλεσμα περάσει στο

Reduce task, το Hadoop θα ξανατρέξει το Map task σε άλλο κόμβο για την αναπαραγωγή του αποτελέσματος.

Τα Reduce tasks δεν έχουν το πλεονέκτημα της τοπικότητας των δεδομένων, επειδή η είσοδος κάθε ενός από αυτά είναι τυπικά η έξοδος από όλα τα Map tasks. Στο παράδειγμα που φαίνεται παρακάτω στην Εικόνα 4.1, βλέπουμε ένα Reduce task που τροφοδοτείται από όλα τα Map tasks.



Εικόνα 4.1Η διαδικασία των δύο φάσεων Map και Reduce

Επομένως, τα ταξινομημένα Map αποτελέσματα πρέπει να μεταφερθούν κατά μήκος του δικτύου στον κόμβο που τρέχει το Reduce task, όπου και συγχωνεύονται και περνάνε στην συνάρτηση reduce του προγράμματος. Το αποτέλεσμα αποθηκεύεται συνήθως στο HDFS.

4.3 Ανατομία εκτέλεσης μιας MapReduce job

Μια MapReduce job εκτελείται καλώντας την μέθοδο submit() σε ένα αντικείμενο Job. Η κλήση αυτής της μεθόδου κρατά μεγάλο μέρος της επεξεργασίας στο παρασκήνιο. Αυτό το κεφάλαιο εξετάζει τα βήματα που ακολουθεί το Hadoop για την εκτέλεση ενός job.

Όλη η διαδικασία εκτέλεσης μιας MapReduce job φαίνεται αναλυτικά στην παραπάνω εικόνα. Στο υψηλότερο επίπεδο, υπάρχουν πέντε ανεξάρτητες οντότητες:

Ο client, που καταθέτει το MapReduce job
Ο YARN Resource Manager, ο οποίος συντονίζει την ανάθεση των υπολογιστικών πόρων στους υπολογιστές του cluster
Οι YARN node managers, που αρχικοποιούν και παρακολουθούν τους containers στους υπολογιστές
Ο MapReduce application master, που συγχρονίζει τα tasks που συνθέτουν το job
Το κατανεμημένο σύστημα αρχείων (συνήθως HDFS), που χρησιμοποιείται για το μοίρασμα των job αρχείων με άλλες οντότητες

Η μέθοδος submit() στο αντικείμενο Job δημιουργεί ένα εσωτερικό στιγμιότυπο το οποίο καλεί την submitJobInternal() μέθοδο (βήμα 1). Έχοντας καταθέσει το job, η μέθοδος waitForCompletion() ελέγχει την πρόοδο του job μία φορά το δευτερόλεπτο και την αναφέρει στην κονσόλα αν έχει αλλάξει σε σχέση με την προηγούμενη αναφορά. Όταν το job ολοκληρωθεί επιτυχώς, οι μετρητές του job εμφανίζονται. Σε διαφορετική περίπτωση, το σφάλμα που προκάλεσε την αποτυχία του job καταγράφεται στην κονσόλα.

Η διαδικασία κατάθεσης του job που υλοποιείται από το JobSubmitter αντικείμενο κάνει τα ακόλουθα:

Ζητάει από τον resource manager για ένα νέο ID εφαρμογής, που χρησιμοποιείται για το ID του MapReduce job (βήμα 2)
Ελέγχει το αποτέλεσμα του job. Για παράδειγμα, αν ο κατάλογος του αποτελέσματος δεν έχει προσδιοριστεί ή υπάρχει ήδη, το job δεν καταθέτεται και το MapReduce πρόγραμμα αποτυγχάνει
Υπολογίζει τα splits των δεδομένων εισόδου. Αν τα splits δεν μπορούν να υπολογιστούν, , το job δεν καταθέτεται και το MapReduce πρόγραμμα αποτυγχάνει
Αντιγράφει τους πόρους που χρειάζονται για την εκτέλεση του job, συμπεριλαμβανομένου του JAR αρχείου, του αρχείου ρυθμίσεων και των υπολογισθέντων splits στο κοινόχρηστο σύστημα αρχείων σε έναν κατάλογο που ονομάζεται από το ID του job (βήμα 3)
Καταθέτει το job καλώντας την submitApplication() στον Resource Manager

Όταν ο resource manager λάβει την κλήση της submitApplication() μεθόδου, παραδίδει την αίτηση στον YARN χρονοδρομολογητή. Ο χρονοδρομολογητής αναθέτει έναν container και στην συνέχεια ο resource manager εκκινεί την διεργασία του application master, υπό την διαχείριση του node manager. (βήματα 5α και 5β)

Ο application master για MapReduce jobs είναι μία Java εφαρμογή. Αρχικοποιεί το job δημιουργώντας κάποια αντικείμενα για την ιχνηλάτηση της προόδου του job, καθώς λαμβάνει αναφορές προόδου και ολοκλήρωσης από τα tasks (βήμα 6). Στη συνέχεια, ανακτά τα splits των δεδομένων εισόδου που υπολογίστηκαν στον client από το κοινόχρηστο σύστημα αρχείων (βήμα 7). Μετά δημιουργεί ένα map task αντικείμενο για κάθε split, αλλά και έναν αριθμό από αντικείμενα reduce task. Σε αυτό το σημείο τα tasks παίρνουν IDs.

Ο application master ζητάει containers για όλα τα map και reduce tasks του job από τον resource manager (βήμα 8). Οι αιτήσεις για map tasks γίνονται πρώτες και με υψηλότερη προτεραιότητα από αυτήν των reduce tasks, αφού όλα τα map tasks πρέπει να ολοκληρωθούν πριν την φάση της ταξινόμησης των reduce tasks. Οι αιτήσεις για τα reduce tasks δεν γίνονται μέχρι να έχει ολοκληρωθεί το 5% των map tasks.

Οι αιτήσεις προσδιορίζουν επίσης και τις ανάγκες για μνήμη και πυρήνες επεξεργαστή για τα tasks. Εξ ορισμού, κάθε map και reduce task παίρνει 1.024 MB μνήμης και έναν ψηφιακό πυρήνα. Οι τιμές αυτές είναι ρυθμίσιμες για κάθε job ξεχωριστά μέσω κάποιων παραμέτρων.

Όταν σε ένα taskέχουν ανατεθεί πόροι για έναν container σε έναν συγκεκριμένο κόμβο από τον χρονοδρομολογητή του resource manager, ο application master αρχικοποιεί τον container επικοινωνώντας με τον node manager (βήματα 9α και 9β). Το task εκτελείται από μια Java εφαρμογή της οποίας η main κλάση είναι η YarnChild. Πριν μπορεί να τρέξει το task, εντοπίζει τους πόρους που χρειάζονται (βήμα 10). Τελικά, τρέχει το map ή το reduce task (βήμα 11).

Τα MapReduce jobs είναι διεργασίες μακράς διάρκειας που παίρνουν από δέκατα του δευτερολέπτου μέχρι και κάποιες ώρες για να ολοκληρωθούν. Επειδή αυτό μπορεί να διαρκέσει πολύ ώρα, είναι σημαντικό για τους χρήστες να παίρνουν κάποια ανάδραση για το πώς προχωράει το job. Ένα jobκαι κάθε ένα από τα task του έχουν μια κατάσταση (status), που περιλαμβάνει θέματα όπως την κατάσταση του εκάστοτε task, την πρόοδο των map και reduce, τις τιμές των μετρητών των jobs και ένα μήνυμα κατάστασης ή μια περιγραφή. Αυτές οι καταστάσεις μεταβάλλονται κατά την διάρκεια ενός job.

Όταν ένα task εκτελείται, καταγράφει την πρόοδό του (το μέρος του task που έχει ολοκληρωθεί). Για τα map tasks αυτό αφορά το μέρος των δεδομένων εισόδου που έχει επεξεργαστεί. Για τα reduce tasks, είναι λίγο πιο σύνθετο, αλλά το σύστημα μπορεί και πάλι να εκτιμήσει το μέρος των reduce δεδομένων εισόδου που έχει επεξεργαστεί. Αυτό το κάνει διαχωρίζοντας την συνολική πρόοδο σε τρία κομμάτια, που αντιστοιχούν στις τρεις φάσεις του ανακατέματος (shuffle).

Η πρόοδος δεν είναι πάντα μετρήσιμη, παρ' όλα αυτά, σημαίνει για το Hadoop ένα task κάνει κάτι. Για παράδειγμα, ένα task γράφοντας εγγραφές κάνει πρόοδο ακόμη κι αν δεν μπορεί να εκφραστεί ως ποσοστό ενός συνολικού αριθμού που θα γραφτεί.

Η αναφορά της προόδου είναι σημαντική, διότι το Hadoop δεν θα διακόψει ένα task που κάνει πρόοδο. Όλες οι παρακάτω πράξεις συνιστούν πρόοδο:

Διάβασμα μιας εγγραφής εισόδου(σε mapper ή reducer)
Γράψιμο μιας εγγραφής εξόδου (σε mapper ή reducer)
Ορισμός της περιγραφής κατάστασης
Αύξηση ενός μετρητή

Τα tasks έχουν και κάποια σετ από μετρητές που μετρούν διάφορα γεγονότα ενώ τρέχει το task, τα οποία είναι είτε μέρη του framework, όπως ο αριθμός των map εγγραφών εξόδου, είτε ορίζονται από τους χρήστες.

Όσο το map ή το reduce task τρέχει, η διεργασία παιδί επικοινωνεί με τον γονικό application master. Το task αναφέρει την πρόοδο του και την κατάστασή του πίσω στον application master, που έχει μία συνολική άποψη περί του job, κάθε τρία δευτερόλεπτα.

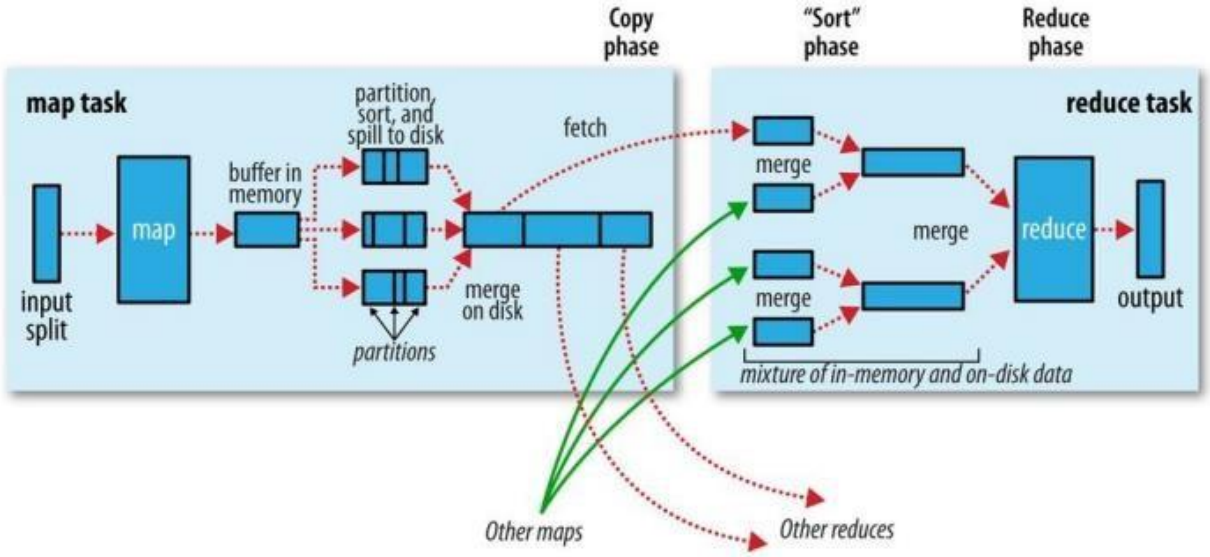
Η web διεπαφή χρήστη του resource manager εμφανίζει όλες τις εκτελούμενες εφαρμογές με συνδέσμους στις web διεπαφές χρήστη τους, η κάθε μία από τις οποίες εμφανίζει επιπλέον πληροφορίες για την MapReduce job, συμπεριλαμβανομένου και της προόδου του.

4.4 Shuffle και Sort στο MapReduce

Το MapReduce εγγυάται ότι η είσοδος σε κάθε reducer ταξινομείται με βάση ένα κλειδί. Η διαδικασία με την οποία το σύστημα ταξινομεί και μεταφέρει τα δεδομένα εξόδου των mappers στις reducers σαν δεδομένα εισόδου είναι γνωστή ως **shuffle**. Σε αυτό το κεφάλαιο θα εξετάσουμε το πώς δουλεύει το shuffle, που είναι μία απαραίτητη διαδικασία σε ένα MapReduce πρόγραμμα. Στην ουσία, το shuffle είναι η περιοχή του προγράμματος στην οποία γίνονται συνεχώς τελειοποιήσεις και βελτιώσεις, έτσι η παρακάτω περιγραφή αναγκαστικά κρύβει πολλές λεπτομέρειες. Δηλαδή, το shuffle αποτελεί την καρδιά του MapReduce.

Όταν η συνάρτηση map αρχίσει να παράγει δεδομένα εξόδου, αυτά δεν εγγράφονται απλά στον δίσκο. Η διαδικασία είναι πιο περίπλοκη και εκμεταλλεύεται το γεγονός της προσωρινής αποθήκευσης

δεδομένων στην μνήμη αλλά και μίας ταξινόμησης εκ των προτέρων. Στην παρακάτω εικόνα φαίνεται η όλη διαδικασία.



Εικόνα 4.2 Οι φάσεις ενός MapReduce προγράμματος

Κάθε map task έχει μία κυκλική προσωρινή μνήμη που γράφει τα δεδομένα εξόδου. Το μέγεθος αυτής της μνήμης είναι εξ ορισμού 100 MB (το οποίο μπορεί να ρυθμιστεί σε διαφορετικό μέγεθος). Όταν τα περιεχόμενα αυτής της προσωρινής μνήμης φτάσουν σε ένα συγκεκριμένο όριο, ένα νήμα (thread) που δουλεύει στο παρασκήνιο, θα ξεκινήσει να διαρρέει τα δεδομένα στον δίσκο. Τα δεδομένα εξόδου των mappers θα συνεχίσουν να καταφθάνουν στην προσωρινή μνήμη καθώς η προαναφερθείσα διαδικασία λαμβάνει χώρα. Αν όμως η προσωρινή μνήμη γεμίσει τελείως, η map συνάρτηση θα μπλοκαριστεί μέχρι η διαρροή των δεδομένων στο δίσκο να ολοκληρωθεί.

Προτού γράψει στον δίσκο, το νήμα διαχωρίζει τα δεδομένα σε κομμάτια που αντιστοιχούν σε reducers στις οποίες εν τέλει θα σταλθούν. Σε κάθε κομμάτι, το νήμα πραγματοποιεί μία ταξινόμηση μέσα στην μνήμη βάσει ενός κλειδιού, και αν υπάρχει μία συνάρτηση combiner, αυτή εκτελείται στα δεδομένα εξόδου της ταξινόμησης. Αυτή η εκτέλεση της combiner συνάρτησης βοηθάει στη δημιουργία πιο συμπυκνωμένων map δεδομένων εξόδου, έτσι ώστε να υπάρχουν λιγότερα δεδομένα προς μεταφορά στον δίσκο αλλά και προς τις reducers.

Κάθε φορά που η προσωρινή μνήμη φτάνει στο όριο που έχει οριστεί, ένα νέο αρχείο διαρροής δημιουργείται, κι έτσι μετά την τελευταία εγγραφή του map task, μπορεί να υπάρχουν πολλαπλά αρχεία διαρροής. Πριν την ολοκλήρωση του task, τα αρχεία διαρροής συμπτύσσονται σε ένα ταξινομημένο αρχείο δεδομένων εξόδου.

Αν υπάρχουν τουλάχιστον τρία αρχεία διαρροής, η combiner εκτελείται ξανά πριν γραφτεί το αρχείο δεδομένων εξόδου. Αν υπάρχουν μόνο ένα ή δύο αρχεία διαρροής, τότε η πιθανή μείωση των δεδομένων εξόδου της map, δεν αξίζει αναλογικά με την καθυστέρηση που θα προκληθεί από την κλήση της combiner, άρα δεν εκτελείται ξανά.

Συχνά είναι καλή ιδέα η συμπίεση των δεδομένων εξόδου της map, επειδή έτσι γίνεται γρηγορότερα η εγγραφή στον δίσκο, σώζεται πολύτιμος χώρος στον δίσκο και μειώνονται τα δεδομένα προς μεταφορά στην reducer. Σύμφωνα με τις προεπιλεγμένες ρυθμίσεις, τα δεδομένα εξόδου δεν

συμπιέζονται, αλλά η ενεργοποίηση αυτού του χαρακτηριστικού είναι εύκολη με την ρύθμιση της κατάλληλης παραμέτρου. Τα κομμάτια των δεδομένων εξόδου στέλνονται στις reducers μέσω HTTP.

Τώρα σειρά έχει η ανάλυση της διαδικασίας του reduce. Το map αρχείο εξόδου βρίσκεται στον δίσκο του υπολογιστή που εκτέλεσε το map task, αλλά τώρα χρειάζεται στον υπολογιστή που πρόκειται να εκτελέσει το reduce task. Επιπρόσθετα, τα reduce tasks χρειάζονται τις map εξόδους που βρίσκονται σε πολλαπλούς υπολογιστές του cluster. Τα map tasks μπορεί να τελειώσουν σε διαφορετικούς χρόνους, έτσι η reduce task αρχίζει να αντιγράφει τα δεδομένα εξόδου τους μόλις κάθε μία από αυτές ολοκληρώσει. Αυτή είναι γνωστή ως η φάση αντιγραφής του reduce task. Το reduce task διαθέτει ένα μικρό αριθμό από νήματα αντιγραφής για να μπορεί να φέρνει map δεδομένα εξόδου παράλληλα. Εξ ορισμού υπάρχουν πέντε νήματα, αριθμός που μπορεί να αλλάξει από τις ρυθμίσεις.

Τα map δεδομένα εξόδου αντιγράφονται στην JVM μνήμη του reduce task αν είναι αρκετά μικρά, αλλιώς αντιγράφονται στον δίσκο. Όταν η προσωρινή μνήμη φτάσει στο όριο, τα δεδομένα μεταφέρονται στον δίσκο. Αν υπάρχει μία combiner συνάρτηση, θα εκτελεστεί για την μείωση των δεδομένων που θα εγγραφούν στον δίσκο.

Όσο τα αντίγραφα συσσωρεύονται στον δίσκο ένα νήμα που τρέχει στο παρασκήνιο τα ενώνει σε μεγαλύτερα και ταξινομημένα αρχεία. Αυτό σώζει χρόνο στην πορεία. Αξίζει να σημειωθεί πως όσα δεδομένα εισόδου συμπιέστηκαν θα πρέπει να αποσυμπιεστούν στην μνήμη έτσι ώστε αυτά να ενωθούν.

Όταν όλα τα δεδομένα εξόδου αντιγραφούν, το reduce task προχωρά στην φάση της ταξινόμησης (sort), η οποία κανονικά θα έπρεπε να ονομάζεται φάση σύμπτυξης (merge), καθώς η ταξινόμηση έχει πραγματοποιηθεί στην πλευρά της map. Η ταξινόμηση ενώνει όλα τα δεδομένα εξόδου και τα διατηρεί με ταξινομημένη σειρά. Αυτό συμβαίνει σε γύρους. Για παράδειγμα, αν υπάρχουν 50 αρχεία map δεδομένων εξόδου και ο παράγοντας σύμπτυξης είναι το 10, θα υπάρξουν 5 γύροι. Κάθε γύρος θα ενώσει 10 αρχεία σε 1 και στο τέλος θα υπάρχουν 5 ενδιάμεσα αρχεία.

Αντί να υπάρχει ένας τελευταίος γύρος κατά τον οποίο ενώνονται αυτά τα 5 αρχεία σε ένα ταξινομημένο αρχείο, η διαδικασία της σύμπτυξης γλιτώνει μία διαδρομή στον δίσκο σερβίροντας απευθείας την reduce συνάρτηση στην τελική φάση, δηλαδή στην φάση αφαίρεσης (reduce).

Κατά την φάση της αφαίρεσης, η reduce συνάρτηση καλείται για κάθε κλειδί των ταξινομημένων δεδομένων εισόδου. Η έξοδος αυτής της φάσης γράφεται απευθείας στο σύστημα αρχείων που τυπικά είναι το HDFS. Στην περίπτωση του HDFS, επειδή ο node manager τρέχει σε έναν datanode, το πρώτο αντίγραφο θα γράφεται στον τοπικό δίσκο.

4.5 Είδη αποτυχιών στο Hadoop

Στον πραγματικό κόσμο ο κώδικας έχει σφάλματα οι διεργασίες αποτυγχάνουν και οι υπολογιστές παθαίνουν βλάβες. Ένα από τα κυριότερα πλεονεκτήματα του Hadoop είναι η ικανότητά του να διαχειρίζεται τέτοιες αποτυχίες και να επιτρέπει στα jobs να ολοκληρώνονται επιτυχώς. Είναι αναγκαίο να αναλογιστούμε τις αποτυχίες για τις παρακάτω οντότητες: το task, ο application master, ο node manager και ο resource manager.

Ας αναλογιστούμε την πρώτη περίπτωση, δηλαδή την αποτυχία ενός task. Η πιο κοινή περίπτωση αυτής της αποτυχίας είναι όταν ο κώδικας πετάξει ένα runtime exception. Αν αυτό συμβεί, το task JVM αναφέρει το σφάλμα στον γονικό application master,προτού τερματίσει. Το σφάλμα φτάνει τελικά στα logs. Ο application master σημειώνει την απόπειρα του task ως αποτυχημένη και ελευθερώνει το container έτσι ώστε οι πόροι του να γίνουν διαθέσιμοι για κάποιο άλλο task.

Μία άλλη αποτυχία είναι η ξαφνική έξοδος του JVM task.Σε αυτή την περίπτωση, ο node manager αντιλαμβάνεται ότι η διεργασία έχει τερματιστεί απότομα και ενημερώνει τον application master, για να σημειωθεί αυτή η προσπάθεια σαν αποτυχημένη.

Τα tasks που «κρεμάνε» χρίζουν διαφορετικής αντιμετώπισης. Ο application master καταλαβαίνει ότι δεν έχει λάβει κάποια αναφορά προόδου για αρκετή ώρα και προχωρά στην καταγραφή του συγκεκριμένου task σαν αποτυχημένου. Το task JVM θα τερματιστεί αυτόματα μετά από αυτή την περίοδο, η οποία κυμαίνεται γύρω στα 10 λεπτά και μπορεί να ρυθμιστεί ξεχωριστά για κάθε job.

Αν αυτή η περίοδος τεθεί στο μηδέν, τότε οι μακροχρόνιες διεργασίες δεν μαρκάρονται ποτέ ως αποτυχημένες. Υπό αυτές τις συνθήκες μία τέτοια διεργασία δεν θα ελευθερώσει ποτέ το container της και με το πέρασμα του χρόνου μπορεί να υπάρξει καθυστέρηση σε όλο τον cluster. Συνεπώς αυτή η προσέγγιση θα ήταν καλό να αποφεύγεται.

Όταν ο application master ειδοποιείται ότι ένα task αποτύχει, θα προγραμματίσει ξανά την εκτέλεσή του. Ο application master θα προσπαθήσει να αποφύγει να προγραμματίσει το αποτυχημένο task σε κάποιον node manager στον οποίο έχει ήδη αποτύχει. Επιπρόσθετα, αν ένα task αποτύχει 4 φορές δεν θα γίνει απόπειρα να εκτελεστεί ξανά. Εξ ορισμού, αν ένα task αποτύχει 4 φορές, τότε αποτυγχάνει ολόκληρο το job.

Όπως και τα MapReduce tasks,έτσι και οι εφαρμογές σε YARN έχουν αρκετές ευκαιρίες να επιτύχουν σε περίπτωση σφαλμάτων. Το YARN επιβάλλει ένα όριο στο μέγιστο αριθμό προσπαθειών για κάθε YARN application master που τρέχει στον cluster και οι εφαρμογές δεν μπορούν να ξεπεράσουν αυτό το όριο.

Ο τρόπος με τον οποίο δουλεύει η επαναφορά είναι ο ακόλουθος. Ένας application master στέλνει περιοδικά έναν παλμό στον resource manager και στο γεγονός της αποτυχίας του application master, ο resource manager θα εντοπίσει την αποτυχία και θα ξεκινήσει ένα νέο στιγμιότυπο του master σε ένα καινούργιο container. Στην περίπτωση του MapReduce application master, θα χρησιμοποιηθεί το ιστορικό των jobs για την επαναφορά των καταστάσεων των tasks που έχουν ήδη εκτελεστεί από την εφαρμογή που έχει αποτύχει, έτσι ώστε να μην ξανατρέξουν.

Ο MapReduce client ζητά από τον application master για αναφορές προόδου, αλλά αν ο application master του αποτύχει, ο client θα πρέπει να εντοπίσει το καινούργιο στιγμιότυπο. Κατά την διάρκεια της αρχικοποίησης του job, ο client ρωτά τον resource manager για την διεύθυνση του application master και την αποθηκεύει στην cache για να μην κατακλύζει τον resource manager με αιτήσεις κάθε φορά που ζητά αναφορές προόδου από τον application master. Αν όμως ο application master αποτύχει, ο client θα παρατηρήσει μία καθυστέρηση όταν εκδώσει μία ανανέωση κατάστασης. Στο σημείο αυτό, ο client θα πάει πίσω στον resource manager για να ζητήσει την καινούργια διεύθυνση του application master. Αυτή η διαδικασία γνωστοποιείται στον χρήστη.

Αν ένας node manager αποτύχει ολοκληρωτικά ή τρέχει πολύ αργά, θα σταματήσει να στέλνει παλμούς προς τον resource manager (ή θα τους στέλνει ακαθόριστα). Ένας resource manager θα συνειδητοποιήσει ότι ένας node manager έχει σταματήσει να δίνει σημεία ζωής αν δεν έχει λάβει κάτι για 10 λεπτά και θα τον αφαιρέσει από την λίστα με τους κόμβους που είναι προς ανάθεση πόρων.

Κάθε task ή application master που τρέχει στον αποτυχημένο node manager θα επανέλθει χρησιμοποιώντας τους μηχανισμούς που περιγράφηκαν παραπάνω. Επίσης, ο application master κανονίζει τα map tasks που εκτελέστηκαν και ολοκληρώθηκαν επιτυχώς στον κόμβο που έχει αποτύχει να εκτελεστούν ξανά αν ανήκουν σε μη ολοκληρωμένα jobs. Αυτό γίνεται διότι η ενδιάμεση έξοδος που βρίσκεται στο τοπικό σύστημα αρχείων του κόμβου που έχει αποτύχει, μπορεί να μην είναι προσβάσιμη στο reduce task.

Οι node managers μπορεί να μπουν σε μαύρη λίστα αν ο αριθμός των αποτυχιών για μία εφαρμογή είναι υψηλός, ακόμη κι αν ο node manager αυτός καθαυτός δεν έχει αποτύχει. Η τοποθέτηση στην μαύρη λίστα πραγματοποιείται από τον application master και για το MapReduce ο application master θα αποπειραθεί να επαναπρογραμματίσει τα tasks σε άλλους κόμβους αν περισσότερα από τρία tasks αποτύχουν σε έναν node manager.

Η αποτυχία του resource manager είναι σοβαρή, επειδή χωρίς αυτόν, ούτε τα jobs ούτε και τα containers δεν μπορούν να εκκινήσουν. Σύμφωνα με τις προεπιλεγμένες ρυθμίσεις, ο resource manager αποτελεί μοναδικό σημείο αποτυχίας, αφού στην σχετικά απίθανη περίπτωση που αυτός αποτύχει, όλες τα εκτελούμενα jobs αποτυγχάνουν και δεν μπορούν να επαναφερθούν.

Για την επίτευξη της υψηλής διαθεσιμότητας, είναι απαραίτητο να τρέχουν δύο resource managers με μια ρύθμιση active-standby. Αν ο ενεργός resource manager αποτύχει, ο standby θα αναλάβει καθήκοντα χωρίς κάποια αξιοσημείωτη καθυστέρηση.

Πληροφορίες σχετικά με τις εφαρμογές που εκτελούνται αποθηκεύονται σε μία αποθήκη υψηλής διαθεσιμότητας, έτσι ώστε ο standby να μπορέσει να επαναφέρει την κύρια κατάσταση του resource manager που έχει αποτύχει. Οι πληροφορίες που αφορούν τους node managers δεν αποθηκεύονται στην προαναφερθείσα αποθήκη καθώς μπορούν να ανακατασκευαστούν σχετικά γρήγορα μόλις ο καινούργιος resource manager στείλει τον πρώτο παλμό.

Όταν ο καινούργιος resource manager ξεκινήσει, διαβάζει τις πληροφορίες των εφαρμογών από την αποθήκη (στην ουσία αναφερόμαστε σε ένα είδος log file) και επανεκκινεί τους application masters για όλες τις εφαρμογές που τρέχουν στον cluster. Αυτό δεν μετράει σαν αποτυχημένη προσπάθεια εφαρμογής, μιας και η εφαρμογή δεν απέτυχε λόγω σφάλματος στον κώδικα, αλλά τερματίστηκε βίαια από το σύστημα. Στην πράξη, η επανεκκίνηση του application master δεν είναι θέμα για τις MapReduce εφαρμογές αφού επαναφέρουν την δουλειά που έχει πραγματοποιηθεί από ολοκληρωμένα tasks.

4.6 Σύντομη αναφορά στο Spark

4.6.1 Εισαγωγή

Έχοντας περιγράψει όλα τα βασικά κομμάτια που αποτελούν ένα Hadoop cluster, κρίνεται αναγκαίο να ρίξουμε μία ματιά και στο Spark, αφού θα χρησιμοποιηθεί στα πλαίσια της εργασίας ως εναλλακτική λύση του MapReduce.

4.6.2 Βασικά χαρακτηριστικά του Spark

Για αρχή ας δούμε τα βασικά χαρακτηριστικά που το κάνουν να διαφέρει αλλά και να προτιμάται σε σχέση με το MapReduce.

Ταχύτητα: Το Spark τρέχει έως και 100 φορές γρηγορότερα από το MapReduce για επεξεργασία μεγάλων όγκων δεδομένων

Ισχυρό caching: Απλό προγραμματιστικό επίπεδο που παρέχει ισχυρές caching ικανότητες

Επιλογές υποδομής: Μπορεί να βασιστεί στο Mesos, στο Hadoop μέσω YARN ή ακόμη και σε δικό του cluster manager

Πραγματικός χρόνος: Προσφέρει επεξεργασία σε πραγματικό χρόνο και ελάχιστες καθυστερήσεις λόγω της επεξεργασίας μέσα στη μνήμη
Πολύγλωσσο: Το Spark παρέχει APIs σε Java, Python, Scala και R. Ο κώδικας μπορεί να γραφεί σε οποιαδήποτε από αυτές τις γλώσσες και προσφέρει ένα διαδραστικό shell σε Scala και Python

Ως ένα αρνητικό του θα πρέπει να αναφέρουμε ότι απαιτεί υπολογιστές με τουλάχιστον 8 GB RAM, διότι όλα τα δεδομένα εισόδου θα πρέπει να χωρέσουν ολόκληρα μέσα στη μνήμη. Αυτός είναι και ο λόγος που εκμεταλλεύεται το caching και καταφέρνει να είναι τόσο γρήγορο

4.6.3 Το οικοσύστημα του Spark

Το επόμενο σημείο που αξίζει να σταθούμε είναι αυτό του οικοσυστήματος του Spark. Δηλαδή, τα επιμέρους εγχειρήματα που χρησιμοποιούνται το καθένα για κάποιον συγκεκριμένο σκοπό.

Spark Core: Είναι η βασική μηχανή για παράλληλη και κατανεμημένη επεξεργασία δεδομένων. Είναι υπεύθυνο για διαχείριση μνήμης, ανάκαμψη από σφάλματα και χρονοπρογραμματισμό.

Spark Streaming: Είναι το κομμάτι του Spark που χρησιμοποιείται για επεξεργασία ροής δεδομένων σε πραγματικό χρόνο.

Spark SQL: Η Spark SQL είναι μία καινούργια βιβλιοθήκη που ενσωματώνει σχεσιακή επεξεργασία με το λειτουργικό API του Spark. Υποστηρίζει δεδομένα με ερωτήσεις μέσω SQL ή μέσω του Hive

GraphX: Είναι το API του Spark για (παράλληλο) υπολογισμό γράφων. Σε υψηλό επίπεδο, το GraphX επεκτείνει την RDD αφαιρετικότητα του Spark.

MLib: Χρησιμοποιείται για μηχανική μάθηση σε έναν cluster με Spark

SparkR: Είναι ένα R πακέτο που παρέχει κατανεμημένη επεξεργασία για ανάλυση δεδομένων. Επίσης υποστηρίζει λειτουργίες όπως η επιλογή, το φιλτράρισμα και η συνάθροιση σε μεγάλα σετ δεδομένων

4.6.4 Η αρχιτεκτονική του Spark

Η αρχιτεκτονική του Spark είναι δομημένη σε στρώματα και όλα τα στοιχεία που την καταρτίζουν είναι ενσωματωμένα με ποικίλες επεκτάσεις και βιβλιοθήκες που κάνουν το Spark ένα πολυεργαλείο σε ότι αφορά την επεξεργασία big data. Βασίζεται σε δύο κύριες οντότητες:

Resilient Distributed Dataset (RDD)

Directed Acyclic Graph (DAG)

Τα RDDs είναι η ακρογωνιαία λίθος κάθε Spark εφαρμογής.

Resilient : Ανοχή σε σφάλματα και δυνατότητα ανακατασκευής των δεδομένων
Distributed : Κατανεμημένα δεδομένα σε πολλαπλούς κόμβους του cluster
Dataset : Συλλογή από δεδομένα χωρισμένα σε κομμάτια με τιμές

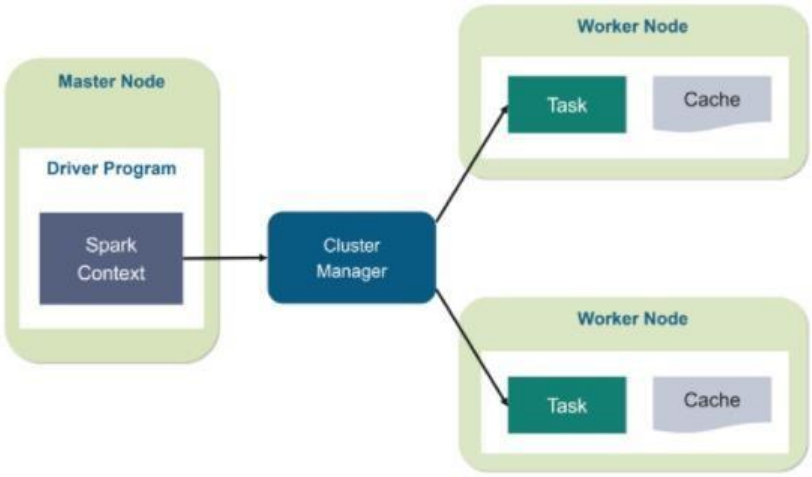
Ένα RDD χωρίζεται σε κομμάτια (partitions) βάσει ενός κλειδιού. Είναι μία δομή με υψηλή ανθεκτικότητα σε σφάλματα καθώς μπορεί να επανέλθει σε περίπτωση βλάβης καθώς εκμεταλλεύεται το γεγονός ότι υπάρχουν πολλαπλά αντίγραφα των δεδομένων. Έτσι, χρησιμοποιώντας την δύναμη

των πολλών κόμβων του cluster επεξεργάζεται δεδομένα ακόμη και στην περίπτωση που κάποιοι από αυτούς αποτύχουν.

Κάθε σετ δεδομένων σε RDD χωρίζεται σε λογικά τμήματα, που μπορεί να επεξεργαστούν σε διαφορετικούς κόμβους. Λόγω αυτού, μπορούν να πραγματοποιηθούν **transformations** και **actions** σε όλα τα δεδομένα παράλληλα. Το Spark αναλαμβάνει επίσης την κατανομή των δεδομένων.

Υπάρχουν δύο τρόποι για την δημιουργία των RDDs: η παραλληλοποίηση μίας ήδη υπάρχουσας συλλογής ή εναλλακτικά η αναφορά ενός σετ δεδομένων από ένα εξωτερικό αποθηκευτικό σύστημα, όπως για παράδειγμα το HDFS.

Με τα RDDs μπορούν να πραγματοποιηθούν δύο είδη λειτουργιών, τα transformations που είναι ενέργειες για τη δημιουργία ενός καινούργιου RDD και τα actions που αφορούν τις οδηγίες που πρέπει να εφαρμοστούν για την επεξεργασία των δεδομένων και την επιστροφή των αποτελεσμάτων που παρήχθησαν.



Εικόνα 4.3 Αρχιτεκτονική Spark

Όπως φαίνεται στην παραπάνω εικόνα, στον master κόμβο υπάρχει το driver πρόγραμμα, το οποίο κατευθύνει την εφαρμογή. Στην ουσία ο κώδικας παίρνει το ρόλο του driver προγράμματος ή αν χρησιμοποιείται το διαδραστικό shell, τότε αυτό αναλαμβάνει αυτά τα καθήκοντα.

Μέσα στο driver πρόγραμμα δημιουργείται το Spark context. Το Spark context είναι μία πύλη για όλες τις λειτουργίες του Spark. Είναι παρόμοιο με την σύνδεση σε μία βάση δεδομένων. Ότι εντολή εκτελείται στη βάση περνάει μέσω της σύνδεσης σε αυτήν. Έτσι και στο Spark οποιαδήποτε λειτουργία πραγματοποιείται διαρρέει μέσω του Spark context.

Το Spark context συνεργάζεται με τον cluster manager για τη διαχείριση διάφορων jobs. Ένα job χωρίζεται σε tasks που κατανέμονται στους υπολογιστές του cluster. Κάθε φορά που ένα RDD δημιουργείται στο Spark context κατανέμεται επίσης στους διάφορους κόμβους και μπορεί επίσης να αποθηκευτεί στην cache τους.

Οι worker κόμβοι είναι οι εργάτες του cluster, που η δουλειά τους είναι να εκτελούν tasks. Αφού εκτελεστούν αυτά τα tasks σε κάποιο RDD, επιστρέφεται το αποτέλεσμα πίσω στο Spark context.

Αν αυξηθεί ο αριθμός των worker κόμβων, θα αυξηθεί και η μνήμη κι έτσι θα μπορούν να δημιουργηθούν περισσότερα partitions, κάτι που σημαίνει ότι τα jobs μπορούν να κατατμηθούν περισσότερο και να εκτελεστούν παράλληλα σε πολλούς κόμβους. Σε τελική ανάλυση, όσο

περισσότεροι κόμβοι υπάρχουν σε έναν cluster, τόσο γρηγορότερα θα πραγματοποιείται η επεξεργασία των δεδομένων.

Κλείνοντας όσα αφορούν το Spark, το μόνο που μένει να περιγράψουμε είναι τα DAGs (Directed Acylic Graph). Ένα DAG είναι ένας γράφος που διατηρεί ένα αρχείο με τις λειτουργίες που έχουν εφαρμοστεί στα RDDs.

Στη συνέχεια, θα εξετάσουμε πως το Spark κατασκευάζει ένα DAG και πως αυτά βοηθούν στην ανοχή στα λάθη. Θα δούμε επίσης και τα πλεονεκτήματα τους που βοηθούν να ξεπεραστούν κάποια μειονεκτήματα του MapReduce.

Directed: Συμβολίζει την άμεση σύνδεση από τον ένα κόμβο στον άλλο.
Acyclic: Ορίζει την ανυπαρξία κύκλου ή βρόγχου. Όταν γίνει ένα transformation, δεν μπορεί να ανακληθεί στην προηγούμενη κατάσταση
Graph: Από την θεωρία γράφων, είναι ένας συνδυασμός από ακμές και άκρα.

Σε αυτούς τους γράφους οι άκρες αντιστοιχούν στα RDDs και οι ακμές στις λειτουργίες που εφαρμόζονται σε αυτά.

Σύμφωνα με το όνομα του, κατευθύνεται προς μία πλευρά από την νωρίτερη στην αργότερη. Όταν καλείται μια λειτουργία, το DAG που δημιουργείται κατατίθεται στον DAG Scheduler. Αυτό διαχωρίζει παραπάνω τον γράφο στα στάδια των jobs.

Είναι το στρώμα του χρονοπρογραμματισμού στο Spark που υλοποιεί χρονοπρογραμματισμό σε στάδια. Όταν κληθεί κάποια ενέργεια, το Spark απευθύνεται στον DAG Scheduler και εκτελεί τα tasks που έχουν προγραμματιστεί από αυτόν.

Τα παρακάτω βήματα περιγράφουν το πώς δουλεύει ο DAG Scheduler:

Ολοκληρώνει τον υπολογισμό και την εκτέλεση κάθε σταδίου του job. Επίσης κρατάει αρχείο με τα RDDs και τρέχει τα jobs

Αποφασίζει τις προτιμώμενες τοποθεσίες, στις οποίες μπορεί να τρέξει το κάθε task ξεχωριστά. Αυτό γίνεται δυνατό με τον task scheduler ο οποίος παίρνει πληροφορίες για την κατάσταση της cache

Διαχειρίζεται τα RDDs, τα οποία αποθηκεύονται στην cache για την αποφυγή αχρείαστων υπολογισμών. Με αυτόν τον τρόπο χειρίζεται και τις πιθανές αποτυχίες, καθώς θυμάται σε τι στάδιο βρίσκεται κάθε το κάθε αρχείο δεδομένων εξόδου και επαναφέρει την ζημία που έχει προκληθεί.

Τα παρακάτω βήματα περιγράφουν την διαδικασία με την οποία το spark δημιουργεί ένα DAG:

Πρώτον, ο χρήστης καταθέτει μία εφαρμογή στο spark

Το driver πρόγραμμα αναλαμβάνει την εφαρμογή από την πλευρά του Spark

Το driver πρόγραμμα βοηθά στην αναγνώριση των transformations και actions που είναι παρόντα στην εφαρμογή

Αυτές οι λειτουργίες κανονίζονται περεταίρω σε μία λογική σειρά που αποτελεί το DAG

Τέλος, ο γράφος μετατρέπεται σε μία φυσική ροή λειτουργιών που περιλαμβάνει στάδια.

Όπως ξέρουμε ένα DAG κρατά αρχείο με τις λειτουργίες που εκτελέστηκαν στα RDDs. Κρατά δηλαδή κάθε λεπτομέρεια των tasks που εκτελέστηκαν σε διαφορετικά partitions του Spark RDD. Έτσι σε περίπτωση αποτυχίας, μπορούν να ανακτηθούν τα δεδομένα με τη βοήθεια του DAG.

Τέλος, θα αναφέρουμε κάποια πλεονεκτήματα των DAG στο Spark, που δίνουν λύσεις σε κάποιες από τις αδυναμίες του MapReduce.

Είναι δυνατό να εκτελεστούν πολλά queries ταυτόχρονα μέσω των DAG. Επίσης μπορούμε να κάνουμε χρήση της SQL. Τα παραπάνω είναι κάποια σημεία στα οποία το MapReduce υστερεί.

Σε σύγκριση με το MapReduce, τα DAG προσφέρουν καλύτερη γενικότερη βελτιστοποίηση σε κάθε επιμέρους κομμάτι του Spark

Κεφάλαιο 5ο: Οδηγός εγκατάστασης Hadoop & Spark

5.1 Εγκατάσταση Hadoop

Η παρακάτω διαδικασία αποτελεί έναν οδηγό βήμα-βήμα για την χειροκίνητη εγκατάσταση του Apache Hadoop. Αφού περιγραφεί η χειροκίνητη διαδικασία, στη συνέχεια θα δοθεί και το bash script με το οποίο αυτοματοποιείται αυτή η εργασία.

Βήμα_1 : Έλεγχος για την ύπαρξη της java που είναι προαπαιτούμενο για το hadoop
Δίνω την εντολή **java –version**
Αν δεν εμφανιστεί κάποιο μήνυμα όπως το παρακάτω , θα πρέπει να εγκατασταθεί η java με την εντολή **wget {java_website_download_path}**

```
hadoopadm@192.168.6.74:~$ java -version
openjdk version "1.8.0_151"
OpenJDK Runtime Environment (build 1.8.0_151-8u151-b12-0ubuntu0.17.04.2-b12)
OpenJDK 64-Bit Server VM (build 25.151-b12, mixed mode)
```

Αποτελέσματα εντολών 5.1 Αποτέλεσμα της εντολής java -version

Βήμα_2: Κατέβασμα και αποσυμπίεση του hadoop
Δίνω την εντολή sudo wget {hadoop_webiste_download_path} για το κατέβασμα
Δίνω την εντολή **tar –xvf hadoop_v.xxx.tar.gz** για αποσυμπίεση σε φάκελο της επιλογής μου
Αλλάζω το όνομα του φακέλου hadoop-2.x.x, δίνοντας **mv hadoop-2.x.x/ hadoop**
Αλλάζω τα δικαιώματα και το ownership από τον κατάλογο hadoop

Βήμα_3: Επεξεργασία του αρχείου .bashrc (σε όλα τα VMs)

vi .bashrc και γράφω τα παρακάτω

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_HOME=/home/hadoopadm/hadoop-2.10.1
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
```

Κώδικας 5.1 Περιεχόμενα του .bashrc αρχείου

Δίνω την εντολή **source .bashrc** για να ανανεωθεί το αρχείο

Βήμα_4: Έλεγχος για την εγκατάσταση του hadoop

Αν όλα έχουν γίνει σωστά θα πρέπει δίνοντας την εντολή **hadoop version** να εμφανιστεί ένα μήνυμα του τύπου:

```
hadoopusr@192.168.6.74:~$ hadoop version
Hadoop 2.10.1
Subversion https://github.com/apache/hadoop -r 1827467c9a56f133025f28557bfc2c562d78e816
Compiled by centos on 2020-09-14T13:17Z
Compiled with protoc 2.5.0
From source with checksum 3114edef868f1f3824e7d0f68be03650
This command was run using /home/hadoopusr/hadoop-2.10.1/share/hadoop/common/hadoop-common-2.10.1.jar
```

Αποτελέσματα εντολών 5.2 Αποτέλεσμα εντολής hadoop version

Βήμα_5: Μεταφορά των αρχείων του hadoop στους slaves
Από τον master δίνω την εντολή : scp /home/hadoopadm/hadoop-x.x.x.tar.gz hadoopadm@192.168.6.82:/home/hadoopadm
scp .bashrc hadoopadm@192.168.6.82:/home/hadoopadm
Ακολουθώ το **Βήμα_2** για αποσυμπίεση στους slaves

Βήμα_8: Επεξεργασία των conflict files για το configuration του hadoop

Μεταφέρομαι στον κατάλογο όπου έχω εγκαταστήσει το hadoop (π.χ. /usr/local/hadoop/etc/hadoop)
Θα πρέπει να γίνει επεξεργασία των πέντε παρακάτω αρχείων:

1. core –site.xml
2. hdfs-site.xml
3. yarn-site.xml
4. mapred-site.xml
5. slaves

Στα παρακάτω πλαίσια φαίνεται το configuration κάθε αρχείου για τον Name Node (master)

```
<configuration>
        <property>
                <name>fs.defaultFS</name>
                <value>hdfs://hadoop-master:9000</value>
        </property>
</configuration>
```

Κώδικας 5.2 Configuration του αρχείου core-site.xml

```
<configuration>
<property>
                 <name>mapreduce.framework.name</name>
                 <value>yarn</value>
</property>
</configuration>
```

Κώδικας 5.3 Configuration του αρχείου mapred-site.xm

```
<configuration>
        <property>
                <name>dfs.replication</name>
                <value>2</value>
         </property>

         <property>
                <name>dfs.namenode.name.dir</name>
                <value>/home/hadoopusr/hadoop-2.10.1/hdfs/namenode</value>
                <final>true</final>
         </property>
         <property>
                <name>dfs.namenode.name.dir</name>
                <value>/home/hadoopusr/hadoop-
2.10.1/hdfs/datanode1</value>
               <final>true</final>
        </property>


         <property>
                <name>dfs.namenode.http-address</name>
                <value>192.168.6.74:50070</value>
         </property>
</configuration>
```

Κώδικας 5.4 Configuration του αρχείου hdfs-site.xml

```
hadoop-slave1
hadoop-slave2
```

Κώδικας 5.5 Configuration του αρχείου slaves

```
<configuration>

         <property>
                   <name>yarn.resourcemanager.address</name>
                   <value>hadoop-master:9001</value>
         </property>

         <property>
                   <name>yarn.resourcemanager.resource-
      tracker.address</name>
                   <value>hadoop-master:8031</value>
         </property>

         <property>
                  <name>yarn.nodemanager.aux-
      services.mapreduce_shuffle.class</name>

<value>org.apache.hadoop.mapred.ShuffleHandler</value>
         </property>

         <property>
                   <name>yarn.nodemanager.aux-services</name>
                   <value>mapreduce_shuffle</value>
        </property>

</configuration>
```

Κώδικας 5.6 Configuration του αρχείου yarn-site.xml

Ακολουθούν screenshots που δείχνουν το configuration κάθε αρχείου για τους Slaves (όσα δεν περιλαμβάνονται παρακάτω παραμένουν ίδια)

```
<configuration>
        <property>
                 <name>dfs.replication</name>
                 <value>2</value>
         </property>

         <property>
                 <name>dfs.datanode.data.dir</name>
                 <value>/home/hadoopusr/hadoop-2.10.1/hdfs/datanode</value>
                 <final>true</final>
         </property>

         <property>
                 <name>dfs.namenode.http-address</name>
                 <value>hadoop-master:50070</value>
         </property>
</configuration>
```

Κώδικας 5.7 Configuration του αρχείου hdfs-site.xml στον slave

Στο αρχείο hdfs-site του κάθε κόμβου αναγράφεται το path που θα αποθηκευτούν τα απαραίτητα αρχεία (metadata στην περίπτωση του master και πραγματικά δεδομένα όσον αφορά τους slaves). Ο κατάλογος που περιέχει αυτά τα δεδομένα (στην περίπτωση μας είναι ο /home/hadoopadm/hadoop/hdfs) θα πρέπει να δημιουργηθεί και να δοθούν σε αυτόν το κατάλληλο ownership και τα ανάλογα δικαιώματα ως εξής:

```
sudo mkdir /home/hadoopadm/hdfs
sudo chown –R hadoopadm:hadoopadm /home/hadoopadm/hdfs
chmod 750 /home/hadoopadm/hdfs
```

Βήμα_9: Format του hdfs και εκκίνηση/τερματισμός του cluster

Με την εντολή **hdfs namenode format** αρχικοποιώ το κατανεμημένο σύστημα αρχείων του hadoop. Αυτή η εντολή δίνεται μόνο μία φορά στον master κόμβο.
ΠΡΟΣΟΧΗ ! Σε περίπτωση που δοθεί ξανά, όλα τα meta data (που βρίσκεται το κάθε block αρχείου που έχει αποθηκευτεί, κτλ) θα χαθούν, καθιστώντας το cluster πρακτικά άχρηστο. Θα πρέπει δηλαδή να επεξεργαστούν και να αποθηκευτούν από την αρχή όποια δεδομένα προϋπήρχαν.
Αφού ολοκληρωθεί η παραπάνω διαδικασία για να εκκινήσω τον cluster δίνω **start-all.sh** από τον master κόμβο.
Αν όλα έχουν πάει καλά δίνοντας την εντολή **jps** (java related processes) θα πρέπει να δω τα παρακάτω ανάλογα με τον τύπο του μηχανήματος. (Master, Secondary Name Node, Slave)

```
hadoopusr@192.168.6.74:~$ jps
11356 NameNode
18932 Jps
3237 NodeManager
11911 ResourceManager
```

Αποτελέσματα εντολών 5.3 Αποτέλεσμα της εντολής jps

```
hadoopusr@192.168.6.74:~$ jps
22539 Jps
22424 NodeManager
22276 DataNode
```

Αποτελέσματα εντολών 5.4 Αποτέλεσμα της εντολής jps στον slave

Βήμα_10: Τελικός έλεγχος για σωστό configuration

Με την εντολή **hdfs dfsadmin –report** ελέγχω αν οι κόμβοι του cluster είναι στην επιθυμητή κατάσταση καθώς και σημαντικές πληροφορίες για καθέναν από αυτούς.

```
Live datanodes (2):
Name: 192.168.6.28:50010 (hadoop-slave1)
Hostname: localhost
Decommission Status : Normal
Configured Capacity: 30829943712 (28.71 GB)
DFS Used: 24576 (24 KB)
Non DFS Used: 22840217600 (21.27 GB)
```

```
DFS Remaining: 6399172608 (5.96 GB)
DFS Used%: 0.00%
DFS Remaining%: 20.76%
Configured Cache Capacity: 0 (B)
Cache Used: 0 (B)
Cache Remaining: 0 (OB)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sat Feb 05 13:20:19 EET 2022
Last Block Report: Sat Feb 05 13:18:19 EET 2022
```

Αποτελέσματα εντολών 5.5 Αποτέλεσμα της εντολής dfsadmin -report

5.2 Εγκατάσταση του Spark

Βήμα 1: Κατέβασμα του tar αρχείου από το site της apache με την εντολή **wget <download link>**

Βήμα 2: Αποσυμπίεση του αρχείου με την εντολή tar –xzf spark-3.2.0-bin-hadoop2.7.tgz

Βήμα 3: Μεταφορά στον φάκελο του spark με την εντολή cd /home/hadoopusr/spark-3.2.0-bin-hadoop2.7/conf

Δημιουργία του αρχείου spark-env.sh από το template αρχείο με την εντολή **cp spark-env.sh.template spark-env.sh**

Βήμα 4: Επεξεργασία του αρχείου spark-env.sh με την εντολή **vi spark-env.sh**

Με το vi προσθέτουμε τις γραμμές που φαίνονται στην παρακάτω εικόνα

```
#Options for the daemons used in the standalone deploy mode
#SPARK_MASTER_HOST, to bind the master to a different IP address or hostname
#SPARK_MASTER_HOST='192.168.6.74'
#SPARK_MASTER_PORT / SPARK_MASTER_WEBUI_PORT, to use non-default ports for the master
#SPARK_MASTER_OPTS, to set config properties only for the master (e.g. "-Dx=y")
#SPARK_WORKER_CORES, to set the number of cores to use on this machine
#SPARK_WORKER_MEMORY, to set how much total memory workers have to give executors (e.g. 1000m, 2g)
#SPARK WORKER PORT / SPARK WORKER WEBUI PORT, to use non-default ports for the worker
```

Αποτελέσματα εντολών 5.6 Περιεχόμενα του spark-env.sh

```
# Options for native BLAS, like Intel MKL, OpenBLAS, and so on.
# You might get better performance to enable these options if using native BLAS (see SPARK-21305).
# - MKL_NUM_THREADS=1 Disable multi-threading of Intel MKL
# - OPENBLAS_NUM_THREADS=1 Disable multi-threading of OpenBLAS
export HADOOP_CONF_DIR=/home/hadoopusr/hadoop-2.10.1/etc/hadoop
export YARN_CONF_DIR=/home/hadoopusr/hadoop-2.10.1/etc/hadoop
76,31
Bot
```

Αποτελέσματα εντολών 5.7 Περιεχόμενα του spark-env.sh

Βήμα 5: Δημιουργία του αρχείου slaves από το template αρχείο με την εντολή **cp workers.template workers**

Όπως και στο προηγούμενο βήμα, με την εντολή **vi** προσθέτω τις IP των workers του cluster

ΣΗΜΕΙΩΣΗ : Θα πρέπει να έχει ρυθμιστεί η επικοινωνία μέσω ssh χωρίς κωδικό με κάθε IP που υπάρχει στο συγκεκριμένο αρχείο

Βήμα 6: Μεταφορά home directory του χρήστη δίνοντας cd

Επεξεργασία του .bashrc με το vi προσθέτοντας τις παρακάτω γραμμές

```
export SPARK_HOME=/home/hadoopadm/spark-3.2.0-bin-hadoop2.7/
export PATH=$PATH:SPARK_HOME
```

Εικόνα 5.9 Περιεχόμενα του .bashrc

Για εκκίνηση του interactive spark shell μετακινούμαι στον bin κατάλογο του spark και δίνω **./spark-shell**

Για εκκίνηση του master κόμβου δίνω την εντολή ./start-master.sh ,μέσα από τον υποκατάλογο sbin του spark καταλόγου

```
hadoopusr@192.168.6.74:~$ jps
16352 Jps
16196 Master
16312 Worker
```

Αποτελέσματα εντολών 5.8 Αποτέλεσμα της εντολής jps στον master

Για εκκίνηση του worker κόμβου δίνω την εντολή ./start-worker.sh spark://<master_ip>:7077,μέσα από τον υποκατάλογο sbin του spark καταλόγου

```
hadoopusr@192.168.6.74:~$ jps
24021 Jps
24070 Master
```

Αποτελέσματα εντολών 5.9 Αποτέλεσμα της εντολής jps στον slave

Εναλλακτικά, μπορεί να χρησιμοποιηθεί η εντολή ./start-all.sh

Κεφάλαιο 6ο: ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Tom White , Hadoop: Definitive Guide,4th edition 2015.

[2] https://techvidvan.com/tutorials/apache-spark-dag-directed-acyclic-graph/

[3] https://blog.insightdatascience.com/simply-install-spark-cluster-mode-341843a52b88

[4]https://www.tutorialkart.com/apache-spark/how-to-setup-an-apache-spark-cluster/?fbclid=IwAR2n438fO0yIY0-STUsvjdwAoNyuL03d8OmI5RvsfUbkCvdD1NcTkslg-3g

[5]Chitresh Verma , Rajiv Pandey, "Comparative Analysis of GFS and HDFS: Technology and Architectural landscape, Published in 2018 10th International Conference on Computational Intelligence and Communication Networks (CICN)

[6]Sajal Tyagi , Shipra Saraswat, "Scheduling Options in YARN", Published in : 2016 International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE)

[7] https://www.geeksforgeeks.org/mapreduce-architecture/

[8] https://www.geeksforgeeks.org/hadoop-history-or-evolution/

[9] https://data-flair.training/blogs/install-apache-spark-multi-node-cluster/