ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΠΜΣ ΔΙΔΑΚΤΙΚΗΣ ΤΗΣ ΤΕΧΝΟΛΟΓΙΑΣ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΚΑΤΕΥΘΥΝΣΗ: ΨΗΦΙΑΚΕΣ ΕΠΙΚΟΙΝΩΝΙΕΣ ΚΑΙ ΔΙΚΤΥΑ

Αποδοτική επεξεργασία ερωτημάτων κατάταξης στο Map/Reduce

Οικονομάκης Σπυρίδων

Επιβλέπων Καθηγητής:

Δουλκερίδης Χρήστος

Πειραιάς 2014

ΠΕΡΙΛΗΨΗ

Η παρούσα Διπλωματική εργασία έχει ως στόχο την αποδοτική επεξεργασία ερωτημάτων με κατάταξη (γνωστά και ως Top-K) με τη μέθοδο Map/Reduce.

Στις εφαρμογές που διαχειρίζονται τεράστιο όγκο δεδομένων, η εκτέλεση υπολογισμών ή Top-K ερωτημάτων πρέπει να πραγματοποιηθεί με έναν κατανεμημένο τρόπο καθώς και με παράλληλη επεξεργασία ώστε να είναι γρήγορη και αποδοτική. Για να επιτευχθεί αυτό, χρησιμοποιήθηκε το σύστημα Hadoop και το προγραμματιστικό μοντέλο του Map/Reduce σε κατανεμημένα περιβάλλοντα. Τα μεγαλύτερα πλεονεκτήματα του Hadoop για την ανάπτυξη κατανεμημένων εφαρμογών είναι η παράλληλη επεξεργασία των δεδομένων σε ένα σύνολο κόμβων ενός συμπλέγματος καθώς και η δυνατότητα να διαχειρίζεται αστοχίες υλικού, καθώς το σύστημα ανιχνεύει τις διεργασίες που έχουν αποτύχει και τις επαναδρομολογεί σε άλλους κόμβους του συμπλέγματος. Έτσι η αξιοπιστία διασφαλίζεται σε επίπεδο λογισμικού και δεν εξαρτάται από την ποιότητα του υλικού. Η σημαντικότερη αδυναμία όμως του Map/Reduce σε περιπτώσεις ερωτημάτων κατάταξης (Top-K) είναι ότι για να εξάγει το τελικό αποτέλεσμα, είναι αναγκασμένο να διαβάσει όλα τα δεδομένα, κάτι το οποίο όμως δεν είναι καθόλου αποδοτικό.

Στην εργασία, μέσα από το πειραματικό μέρος και την εκτέλεση τριών διαφορετικών αλγορίθμων θα αναδειχθούν οι αδυναμίες της προκαθορισμένης λειτουργίας του προγραμματιστικού μοντέλου Map/Reduce σε Top-K ερωτήματα καθώς και η προτεινόμενη λύση και η αποδοτική επεξεργασία τέτοιου τύπου ερωτημάτων. Θα αντιμετωπιστούν δύο από τις κυριότερες αδυναμίες που εμφανίζονται, τόσο αυτή του πρόωρου τερματισμού (EarlyTermination), όσο και η δίκαιη και ισομερής κατανομή του φορτίου των δεδομένων (Load Balancing).

ABSTRACT

The present Thesis aims to process efficiently ranked queries (also known as Top-K) by the Map/Reduce method.

In applications that manage huge volumes of data, the execution of computations or Top-K queries must be carried out in a distributed way, as well as with parallel processing so that it will be quick and efficient. In order to achieve that, the Hadoop system and the Map/Reduce programming model were used in distributed environments. The major advantages of Hadoop in the development of distributed applications is the parallel data processing in a set of cluster nodes, as well as the capability to manage machine failures, while the system detects the tasks that have failed and reroutes them to other nodes of the cluster. In this way, the reliability is ensured in terms of software and it does not depend on the quality of the hardware. However, the most important shortcoming of the Map/Reduce in cases of ranked queries (Top-K) is that in order to extract the final result, it is obliged to read the whole amount of data, a procedure that is not efficient at all.

This study, through the experimental part and the execution of three different algorithms, aims to show the disadvantages of the default operation of the Map/Reduce programming model in Top-K queries, as well as the recommended solution and the effective processing of such query types. Two of the major shortcomings that occur will be managed, namely the Early Termination and the Load Balancing.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 1: Εισαγωγή

1.1 Πρόλογος

Σήμερα είμαστε περικυκλωμένοι από χιλιάδες δεδομένα. Οι άνθρωποι ανεβάζουν video, τραβούν φωτογραφίες με τα κινητά τους, ανανεώνουν το status τους στο Facebook, αφήνουν σχόλια στο διαδίκτυο, κάνουν κλικ σε διαφημίσεις και πολλά άλλα. Τα μηχανήματα επίσης παράγουν και αποθηκεύουν ολοένα και περισσότερα δεδομένα. Επιπλέον οι επιχειρήσεις αποκτούν τεράστιο όγκο δεδομένων από διαφορετικές πηγές και προσπαθούν να αξιοποιήσουν τα δεδομένα αυτά με τον όσο το δυνατόν καλύτερο τρόπο, έτσι ώστε να πάρουν σημαντικές αποφάσεις, είτε για μελλοντικά στρατηγικά πλάνα, είτε για να μετρήσουν αποδοτικά την ικανοποίηση των χρηστών του προϊόντος που χρησιμοποιούν, είτε ακόμα και για την ανάλυση της οικονομικής τους πορείας και φυσικά για πολλούς άλλους λόγους [6].

Η εκθετική αυτή αύξηση δεδομένων παρουσίασε τις πρώτες προκλήσεις στις εταιρείες αιχμής, όπως η Google, η Yahoo, η Amazon και η Microsoft. Οι εταιρείες αυτές αναγκάστηκαν να διαχειριστούν terabytes και petabytes δεδομένων ώστε να υπολογίσουν ποιες ιστοσελίδες είναι οι πιο δημοφιλείς, ποια βιβλία ήταν σε ζήτηση και τι είδους διαφημίσεις έχουν απήχηση στους χρήστες [6].

Βασική απαίτηση για την ανάλυση και αξιοποίηση μεγάλου όγκου δεδομένων είναι η επεκτασιμότητά τους από τεχνικής πλευράς. Παρ' όλα αυτά, η επεξεργασία, η ανάλυση και η εξόρυξη πληροφορίας από μεγάλο όγκο δεδομένων καθιστά την επεκτασιμότητα αδύνατη ή μη οικονομικά εφικτή από ένα μεγάλο εύρος επιχειρήσεων. Όπως γίνεται αντιληπτό, τα υπάρχοντα εργαλεία έχουν αρχίσει να ανεπαρκούν για την επεξεργασία μεγάλου όγκου δεδομένων. Η Google ήταν η πρώτη που παρουσίασε το Map/Reduce, σαν σύστημα με το οποίο μπορεί να καλύψει τις ανάγκες της για την επεκτασιμότητα των δεδομένων της [6].

Το σύστημα αυτό προκάλεσε μεγάλο ενδιαφέρον, γιατί πολλές εταιρείες που αντιμετώπιζαν αντίστοιχο πρόβλημα δεν είχαν την δυνατότητα να δημιουργήσουν από την αρχή ένα δικό τους αντίστοιχο εργαλείο. Ο Doug Cutting είδε μία ευκαιρία

και ανέπτυξε μία έκδοση ανοικτού κώδικα του Map/Reduce συστήματος, το οποίο ονόμασε Hadoop [6].

Σήμερα το Hadoop αποτελεί ένα βασικό μέρος της υπολογιστικής υποδομής μεγάλων εταιρειών διαδικτύου, όπως το Yahoo, το Facebook, το LinkedIn και το Twitter. Επίσης πολλές ακόμα παραδοσιακές επιχειρήσεις. όπως αυτές των μέσων μαζικής ενημέρωσης και των τηλεπικοινωνιών έχουν αρχίσει να υιοθετούν αυτό το σύστημα. Το Hadoop όμως αντιμετωπίζει ένα απλό πρόβλημα: ενώ οι δυνατότητες αποθήκευσης των σκληρών δίσκων έχουν αυξηθεί μαζικά με την πάροδο των ετών, οι ταχύτητες πρόσβασης, δηλαδή οι ταχύτητες με τις οποίες τα δεδομένα μπορούν να διαβαστούν από δίσκους, δεν συμβαδίζουν . Ο προφανής τρόπος για να μειωθεί ο χρόνος πρόσβασης και ανάκτησης δεδομένων είναι να γίνεται η ανάγνωση από πολλούς δίσκους ταυτόχρονα. Δουλεύοντας παράλληλα, θα ήταν δυνατή η ανάγνωση των δεδομένων σε λιγότερο από δύο λεπτά [6].

Το Hadoop δίνει λύση σε βασικά προβλήματα:

- Αστοχία υλικού

  Μόλις ξεκινάει η χρήση του υλικού σε πολλά κομμάτια, τότε η πιθανότητα κάποιου να αποτύχει είναι πολύ μεγάλη. Μία κοινή μέθοδος να αποφύγουμε την απώλεια δεδομένων είναι η αντιγραφή. Το Hadoop σύστημα αρχείων (HDFS) χρησιμοποιεί μία διαφορετική προσέγγιση [6].

- Συσχέτιση δεδομένων

  Δεδομένα από έναν δίσκο είναι ανάγκη να συνδυαστούν με δεδομένα από άλλους δίσκους. Τα περισσότερα κατανεμημένα συστήματα επιτρέπουν στα δεδομένα να συνδυαστούν με άλλα από διαφορετικές πηγές. Το Map/Reduce παρέχει ένα προγραμματιστικό μοντέλο, το οποίο επιλύει το πρόβλημα ανάγνωσης και εγγραφής στο δίσκο, μετατρέποντας το πρόβλημα σε έναν υπολογισμό από ένα σύνολο keys και values [6].

- Ευκολία ανάπτυξης παράλληλων προγραμμάτων από τον προγραμματιστή

Ο κάθε προγραμματιστής το μόνο που έχει να κάνει είναι να υλοποιήσει δύο συναρτήσεις, την Map και την Reduce, μέσα στο πρόγραμμα του. Αυτό που κερδίζει μέσα από τη χρήση του προγραμματιστικού μοντέλου Map/Reduce είναι η απλότητα του κώδικα ανάπτυξης καθώς και του αλγορίθμου. Από την πλευρά του δε χρειάζεται να ασχοληθεί με την εκτέλεση του προγράμματος με τρόπο παράλληλο, γιατί αυτό εξαρτάται από την πλευρά του συστήματος [6]

- Φορητότητα και ευελιξία προγραμμάτων

  Όπως αναφέρθηκε προηγουμένως, λόγω της ευκολίας ανάπτυξης Map/Reduce προγραμμάτων από την πλευρά του προγραμματιστή, τα προγράμματα είναι πολύ εύκολα φορητά σε διαφορετικά συστήματα, αφού η Map μπορεί να εκτελεστεί με τα ίδια δεδομένα σε πολλές μηχανές. Από την πλευρά της η Reduce λαμβάνοντας τα δεδομένα της Map, μπορεί να δημιουργηθεί σε πολλά αντίγραφα με τα ίδια δεδομένα ή ακόμα και να μοιραστούν τα δεδομένα στα διάφορα αντίγραφα και να εκτελεστούν παράλληλα [6].

Η προσέγγιση που υιοθετείται από το Map/Reduce φαίνεται σαν brute-force. Το Map/Reduce εκτελεί λειτουργίες επεξεργασίας με τρόπο batch-query και διαθέτει την ικανότητα να εκτελεί ad-hoc ερωτήματα στο σύνολο των δεδομένων και να επιστρέφει τα αποτελέσματα σε ένα ικανό χρονικό διάστημα [6].

1.2 Σύγκριση Map/Reduce με τις παραδοσιακές Βάσεις Δεδομένων

Ποιος είναι ο λόγος που δεν μπορούμε να χρησιμοποιήσουμε και τις παραδοσιακές βάσεις δεδομένων με ένα μεγάλο σύνολο από δίσκους που εκτελούν μεγάλης επεκτασιμότητας batch ανάλυση και χρειαζόμαστε το Map/Reduce;

Η απάντηση στο παραπάνω ερώτημα προέρχεται από μια άλλη τάση στους σκληρούς δίσκους:

- Η ταχύτητα αναζήτησης βελτιώνεται πιο δύσκολα από την ταχύτητα μεταφοράς στους σκληρούς δίσκους.

Αν το σχέδιο πρόσβασης στα δεδομένα κυριαρχείται από αναζητήσεις, θα χρειαστεί περισσότερος χρόνος για να διαβάσει ή να γράψει μεγάλα τμήματα του συνόλου των δεδομένων. Από την άλλη όμως, για την ανανέωση ενός μικρού ποσοστού εγγραφών σε μία βάση δεδομένων, τα παραδοσιακά B-Trees λειτουργούν πολύ καλά. Το B-Tree είναι λιγότερο αποδοτικό από το Map/Reduce το οποίο χρησιμοποιεί Sort/Merge για την ανοικοδόμηση της βάσης δεδομένων [11].

Το Map/Reduce είναι κατάλληλο για προβλήματα στα οποία πρέπει να γίνει ανάλυση ολόκληρου του συνόλου δεδομένων σε κομμάτια, ιδίως για ad-hoc ανάλυση. Το Map/Reduce ταιριάζει σε εφαρμογές όπου τα δεδομένα είναι γραμμένα μία φορά και εκτελούνται λειτουργίες για ανάγνωση πολλές φορές, ενώ μια σχεσιακή βάση δεδομένων είναι κατάλληλη για σύνολα δεδομένων που ενημερώνονται συνεχώς [11].

1.3 Κριτική

Το Map/Reduce διακρίνεται για τα πλεονεκτήματα του από πλευράς επεκτασιμότητας καθώς και ανοχής σφαλμάτων όπως αναφέρθηκαν και παραπάνω. Παρ' όλα αυτά υπάρχουν σημαντικά προβλήματα που παρουσιάζονται από τη φύση του πλαισίου του Map/Reduce από πλευράς απόδοσης [3].

Παρά τα προφανή πλεονεκτήματα τα οποία θα περιγραφούν αναλυτικότερα παρακάτω, το Map/Reduce έχει και πολλές αδυναμίες καθώς και μη ικανοποιητική απόδοση για διάφορες εργασίες επεξεργασίας δεδομένων. Μερικές από τις σημαντικότερες αδυναμίες που εμφανίζονται με τη χρήση του είναι οι εξής [3]:

- Πρόσβαση στα δεδομένα εισόδου

  Για ορισμένους τύπους ερωτημάτων ανάλυσης θα αρκούσε να υπήρχε η δυνατότητα πρόσβασης μόνο σε ένα υποσύνολο δεδομένων για να παραχθεί το αποτέλεσμα. Άλλοι τύποι ερωτημάτων μπορεί να απαιτούν εστιασμένη πρόσβαση σε μερικές μόνο πλειάδες που ικανοποιούν κάποια προϋπόθεση, οι οποίες δεν μπορούν να παρέχονται χωρίς την πρόσβαση και την επεξεργασία όλων των πλειάδων δεδομένων εισόδου [3].

- Υψηλό κόστος επικοινωνίας

  Όταν όλες οι διεργασίες επεξεργασίας Map έχουν ολοκληρωθεί, μερικά επιλεγμένα δεδομένα αποστέλλονται στις διεργασίες Reduce για περαιτέρω επεξεργασία (το λεγόμενο Shuffling). Ανάλογα με τον τύπο του ερωτήματος και με τον τύπο της επεξεργασίας κάθε φορά, κατά τη διάρκεια της φάσης Map, το μέγεθος της εξόδου της φάσης Map μπορεί να είναι αρκετά μεγάλο και η μετάδοση του μπορεί να καθυστερήσει το συνολικό χρόνο εκτέλεσης της εργασίας. Ένα κλασικό παράδειγμα είναι τα ερωτήματα Join, για το οποία η φάση Map δεν είναι ικανή να περιορίσει τα δεδομένα που θα αποσταλούν στη φάση Reduce [3].

- Περιττή και πολυδάπανη επεξεργασία

  Αρκετά συχνά πολλά Jobs του Map/Reduce αρχίζουν με επικαλυπτόμενα χρονικά διαστήματα και είναι αναγκαίο να υποβάλλονται σε επεξεργασία για το ίδιο σύνολο δεδομένων. Σε τέτοιες περιπτώσεις, είναι πιθανό ότι δύο ή περισσότερα Jobs πρέπει να εκτελέσουν την ίδια επεξεργασία πάνω από τα ίδια δεδομένα. Στο Map/Reduce, οι εργασίες αυτές εκτελούνται σε επεξεργασία ξεχωριστά, με αποτέλεσμα περιττή επεξεργασία [3].

- Επαναϋπολογισμός

  Οι εργασίες Map/Reduce που εκτελούνται σε ένα σύμπλεγμα παράγουν αποτελέσματα εξόδου που αποθηκεύονται στο δίσκο για να εξασφαλιστεί η ανοχή σε σφάλματα κατά την επεξεργασία της εργασίας. Αυτό αποτελεί έναν μηχανισμό ελέγχου που επιτρέπει στις εργασίες που απαιτούν μεγάλο χρόνο εκτέλεσης να ολοκληρωθούν σε περίπτωση αποτυχίας, χωρίς να χρειάζεται να γίνει επανεκτέλεση από την αρχή. Ωστόσο, το Map/Reduce στερείται ενός μηχανισμού για τη διαχείριση και τη μελλοντική επαναχρησιμοποίηση των αποτελεσμάτων της εξόδου. Έτσι, δεν υπάρχει καμία δυνατότητα για την επαναχρησιμοποίηση των αποτελεσμάτων που παράγονται από τα προηγούμενα ερωτήματα, πράγμα που σημαίνει ότι σε ένα μελλοντικό

ερώτημα που απαιτεί το αποτέλεσμα ενός προηγούμενου, θέτει το ερώτημα εκ νέου, με αποτέλεσμα να πρέπει να επαναϋπολογιστούν όλα από την αρχή [3].

- Έλλειψη πρόωρου τερματισμού (Early Termination)

  Το Early Termination έχει ως σκοπό να παρέχει γρήγορα και σωστά αποτελέσματα, για ερωτήματα ανάλυσης στο Map/Reduce, χωρίς την επεξεργασία ολόκληρων των δεδομένων εισόδου [3].

- Έλλειψη επανάληψης

  Σε εφαρμογές ανάλυσης δεδομένων προκύπτουν επαναληπτικοί υπολογισμοί και ερωτήματα αναδρομικά, περιλαμβάνοντας υπολογισμούς PageRank ή HITS, ομαδοποίησης, ανάλυσης κοινωνικών δικτύων, αναδρομικών ερωτημάτων SQL κ.λπ. Ωστόσο, σε ένα Map/Reduce, ο προγραμματιστής πρέπει να γράψει μια σειρά από Map/Reduce Jobs και να συντονίσει την εκτέλεσή τους, προκειμένου να εφαρμόσει μια απλή επαναληπτική επεξεργασία [3].

- Έλλειψη γρήγορης ανάκτησης προσεγγιστικών αποτελεσμάτων

  Σε περιπτώσεις διερευνητικών ερωτημάτων το Map/ Reduce, δεν παρέχει έναν σαφή τρόπο για να υποστηρίξει τη γρήγορη ανάκτηση ενδεικτικών αποτελεσμάτων από επεξεργασία σε αντιπροσωπευτικά δείγματα εισόδου[3].

- Μη δυνατή εξισορρόπηση φορτίου (Load Balancing)

  Συστήματα παράλληλης διαχείρισης δεδομένων προσπαθούν να ελαχιστοποιήσουν το χρόνο εκτέλεσης μίας σύνθετης εργασίας , με την προσεκτική στεγανοποίηση των δεδομένων εισόδου και τη διανομή του φορτίου επεξεργασίας στα διαθέσιμα μηχανήματα. Μέρος του παραπάνω προβλήματος είναι ο καταμερισμός των δεδομένων με δίκαιο τρόπο [3].

- Έλλειψη διαδραστικής ή σε πραγματικό χρόνο επεξεργασίας

Το Map/Reduce έχει σχεδιαστεί ως ένα σύστημα εξαιρετικά ανθεκτικό στην ανοχή σφαλμάτων, ωστόσο δεν έχει τη δυνατότητα να χρησιμοποιηθεί αποτελεσματικά για τη διαδραστική ή σε πραγματικό χρόνο επεξεργασία, η οποία απαιτεί γρήγορο χρόνο εκτέλεσης. Αυτό συμβαίνει διότι για να εξασφαλιστεί η ανοχή σφαλμάτων ενός Map/Reduce προγράμματος, εισάγεται επιπλέον φόρτος που επηρεάζει αρνητικά την απόδοση του [3].

- Έλλειψη υποστήριξης για τις n-way λειτουργίες

  Η επεξεργασία n-way λειτουργιών σε δεδομένα που προέρχονται από πολλαπλές πηγές δεν υποστηρίζεται από ένα Map/Reduce πρόγραμμα [3].

1.4 Στόχος Εργασίας

Τα batch-based συστήματα, όπως είναι το Hadoop, υπόσχονται παράλληλη επεξεργασία μεγάλης κλίμακας, χωρίς όμως να έχουν τη δυνατότητα να βελτιστοποιήσουν ένα Ranked ερώτημα, εφόσον δε λαμβάνουν υπ'όψιν τους παραλλαγές στη χρησιμότητα των δεδομένων [2].

Με στόχο να αποφευχθεί περιττή επεξεργασία, αυτά τα συστήματα επεξεργασίας δεδομένων έχουν ανάγκη να κάνουν χρήση της διεργασίας partitioning καθώς επίσης και στρατηγικών κατανομής πόρων (Load Balancing) που μπορούν να περιορίσουν εγγραφές, οι οποίες δεν είναι αναγκαίο να ληφθούν υπ' όψιν [2].

Σκοπός της παρούσης Διπλωματικής εργασίας είναι η υλοποίηση Map/Reduce προγραμμάτων με αποδοτικό τρόπο και η αντιμετώπιση δύο εκ των ανωτέρω προβλημάτων που παρουσιάζονται στο Map/Reduce [2]:

- Αποδοτική εκτέλεση Top-K ερωτημάτων

  Πληροφοριακά συστήματα διαφορετικών τύπων χρησιμοποιούν διάφορες τεχνικές για να ταξινομήσουν τις απαντήσεις σε ερωτήματα χρηστών. Σε πολλούς τομείς εφαρμογής, οι τελικοί χρήστες ενδιαφέρονται περισσότερο για τα πιο σημαντικά (Top-K) αποτελέσματα στα ερωτήματα που εκτελούν. Διαφορετικές αναδυόμενες εφαρμογές απαιτούν αποτελεσματική υποστήριξη για ερωτήματα Top-K. Για παράδειγμα, στο πλαίσιο του Web, η

αποτελεσματικότητα και η αποδοτικότητα της μετα-αναζήτησης, η οποία συνδυάζει βαθμολογίες από διαφορετικές μηχανές αναζήτησης, είναι άρρηκτα συνδεδεμένες με αποτελεσματικές μεθόδους κατάταξης συνόλων. [5]

Μέσα σε αυτόν τον τεράστιο όγκο πληροφορίας λοιπόν, απαιτείται η αποτελεσματική και σε μικρό χρόνο εκτέλεση Top-K ερωτημάτων. Ένα από τα σημαντικά προβλήματα που εμφανίζονται στο Map/Reduce είναι τέτοιου είδους ερωτήματα, μιας και για την εξαγωγή αποτελεσμάτων τέτοιου τύπου είναι αναγκασμένο να διατρέξει όλα τα δεδομένα (π.χ εκατοντάδες εκατομμύρια) για να παράγει αποτελέσματα. Σκοπός της εργασίας είναι με την αντιμετώπιση κάποιων αδυναμιών (έλλειψη πρόωρου τερματισμού) του Hadoop να επιτευχθεί η αποτελεσματική και αποδοτική εκτέλεση Top-K ερωτημάτων [2].

- Έλλειψη πρόωρου τερματισμού (Early Termination)

Το Early Termination έχει ως σκοπό να παράγει γρήγορα και σωστά αποτελέσματα για ερωτήματα ανάλυσης στο Map/Reduce, χωρίς την επεξεργασία ολόκληρων των δεδομένων εισόδου. Όπως αναφέρθηκε παραπάνω, η προκαθορισμένη λειτουργία του Map/Reduce είναι η επεξεργασία όλων των δεδομένων εισόδου ακόμα και όταν ένα μεγάλο μέρος αυτών δεν είναι χρήσιμα για τα αποτελέσματα [2].

Στην εργασία η υλοποίηση του Early Termination χρησιμοποιεί ενιαίες διαδικασίες δειγματοληψίας και λειτουργεί επαναληπτικά για να υπολογίσει μεγαλύτερα δείγματα μέχρι να επιτευχθεί ένα συγκεκριμένο επίπεδο ακρίβειας και να παράγει συγκεκριμένα όρια με βάση τα δεδομένα εισόδου. Η υλοποίηση στηρίζεται τόσο στα ιστογράμματα της κατανομής των δεδομένων, αφού από αυτά θα υπολογιστούν τα όρια για το Early Termination, όσο και από τον τρόπο με τον οποίο εκτελείται το Map/Reduce κατά την διάρκεια τροφοδότησης της Map με τα δεδομένα εισόδου, έτσι ώστε να τροφοδοτηθεί μόνο με εγγραφές των δεδομένων που είναι χρήσιμες για την παραγωγή αποτελεσμάτων. Με το Early Termination επιτυγχάνεται και η αποδοτική επεξεργασία Top-K δεδομένων [2].

- Μη δυνατή εξισορρόπηση φορτίου (Load Balancing)

  Συστήματα παράλληλης διαχείρισης δεδομένων προσπαθούν να ελαχιστοποιήσουν τον χρόνο εκτέλεσης μίας σύνθετης εργασίας , με την προσεκτική στεγανοποίηση των δεδομένων εισόδου και τη διανομή του φορτίου επεξεργασίας στα διαθέσιμα μηχανήματα. Μέρος του παραπάνω προβλήματος είναι ο καταμερισμός των δεδομένων με δίκαιο τρόπο. [3]

  Στην εργασία η υλοποίηση του Load Balancing βασίζεται στον αλγόριθμο LPT (Long Processing Time) έτσι ώστε να δημιουργηθεί ένας κατάλληλος Partitioner στο Map/Reduce που θα κατανέμει τα κομμάτια δεδομένων δίκαια δίνοντας προτεραιότητα στους Reducers που έχουν τον μικρότερο φόρτο εργασίας.

1.5 Δομή της Διπλωματικής εργασίας

Η Διπλωματική αυτή εργασία αποτελείται από 6 κεφάλαια.

Το *κεφάλαιο 1* αποτελεί την εισαγωγή της εργασίας καθώς και την παρουσίαση του σκοπού της.

Το *κεφάλαιο 2* αποτελεί μία πλήρη και λεπτομερή παρουσίαση και περιγραφή με διαγράμματα και παραδείγματα της αρχιτεκτονικής και της λειτουργίας του Hadoop.

Το *κεφάλαιο 3* έχει σκοπό την ανάδειξη και την παρουσίαση του προβλήματος που θα αντιμετωπιστεί.

Το *κεφάλαιο 4* αποτελεί την περιγραφή της υλοποίησης που δημιουργήθηκε για την αντιμετώπιση του προβλήματος.

Το *κεφάλαιο 5* αποτελεί το πειραματικό μέρος και τη σύγκριση τριών αλγορίθμων ξεχωριστά:

- Naive αλγόριθμος, ο οποίος αποτελεί το κλασικό Map/Reducer πρόγραμμα
- Early Termination αλγόριθμος, ο οποίος αποτελεί βελτιστοποίηση της έλλειψης πρόωρου τερματισμού(περιγράφεται αναλυτικά στο κεφάλαιο 4).
- Early Termination & Load Balancing, ο οποίος αποτελεί βελτιστοποίηση της έλλειψης πρόωρου τερματισμού και μη δυνατής εξισορρόπησης φορτίου (περιγράφεται αναλυτικά στο κεφάλαιο 4).

Το *κεφάλαιο 6* αποτελεί τα συμπεράσματα της εργασίας καθώς και τη μελλοντική εργασία.

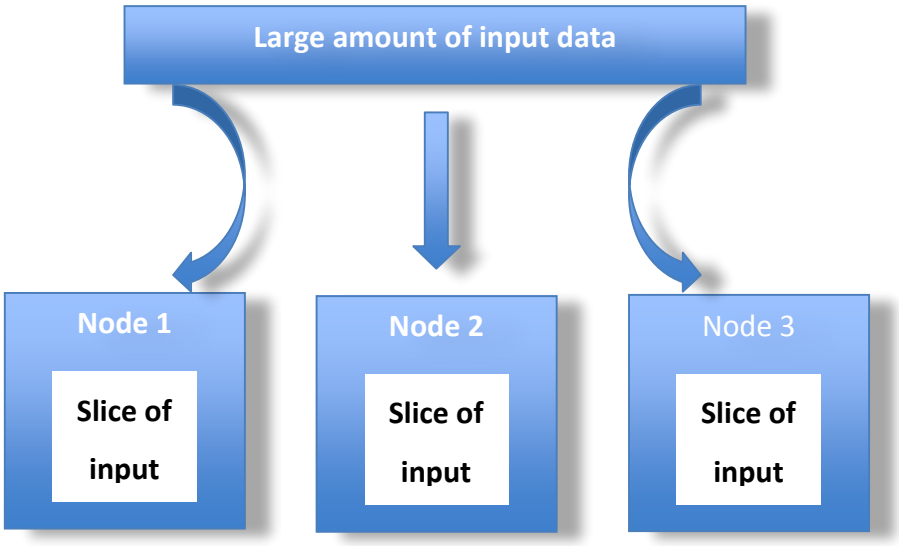
ΚΕΦΑΛΑΙΟ 2: Αρχιτεκτονική και Λειτουργία του Hadoop

2.1 Ορισμός

Το Hadoop είναι μια κατανεμημένη υποδομή μεγάλης κλίμακας για την επεξεργασία κατά δεσμίδες. Το Hadoop έχει τη δυνατότητα να χρησιμοποιηθεί σε ένα μοναδικό μηχάνημα, αλλά η πραγματική του ισχύς έγκειται στην ικανότητα του να αναπτύσσεται σε εκατοντάδες ή χιλιάδες υπολογιστές, όπου ο καθένας διαθέτει ένα πλήθος επεξεργαστικών πυρήνων. Τέλος, το Hadoop διαθέτει την ικανότητα να κατανέμει αποτελεσματικά μεγάλες ποσότητες διεργασιών σε ένα σύνολο μηχανημάτων, τα οποία κάνουν παράλληλη επεξεργασία [10].

Σε ένα σύμπλεγμα από κόμβους Hadoop, τα δεδομένα είναι κατανεμημένα σε όλους τους κόμβους. Το Hadoop Distributed File System (HDFS) θα χωρίσει τα μεγάλα αρχεία δεδομένων σε κομμάτια, τα όποια βρίσκονται σε διαφορετικούς κόμβους μέσα στο σύμπλεγμα από Hadoop εξυπηρετητές. Επιπλέον, το κάθε κομμάτι θα αντιγραφεί σε διαφορετικά μηχανήματα τα οποία ανήκουν στο σύμπλεγμα Hadoop, έτσι ώστε αν σε ένα μοναδικό μηχάνημα προκληθεί σφάλμα τα δεδομένα να είναι διαθέσιμα [10].

Τα δεδομένα στο Hadoop είναι εννοιολογικά προσανατολισμένα την έννοια του αρχείου. Τα μεμονωμένα αρχεία εισόδου είναι χωρισμένα σε γραμμές ή σε άλλες μορφές δεδομένων. Το πλαίσιο Hadoop προγραμματίζει τις παραπάνω διεργασίες εγγύς στην τοποθεσία των δεδομένων/εγγραφών, έχοντας τη γνώση από το κατανεμημένο σύστημα αρχείων. Καθώς τα αρχεία διαμοιράζονται κατά μήκος του κατανεμημένου συστήματος αρχείων σε κομμάτια, κάθε υπολογιστική διεργασία, η οποία εκτελείται σε έναν κόμβο, ενεργεί σε ένα υποσύνολο δεδομένων. Τα δεδομένα διαβάζονται από τον τοπικό δίσκο κατευθείαν μέσα στη CPU, “ανακουφίζοντας” το εύρος δικτύου και προλαμβάνοντας την περιττή μεταφορά δεδομένων μέσω δικτύου [10]..



2.2 Map/Reduce: Ατομικές διεργασίες

Το Hadoop περιορίζει το μέγεθος της επικοινωνίας που μπορεί να εκτελεσθεί από τις διεργασίες, καθώς μία μεμονωμένη εγγραφή εκτελείται από μία διεργασία ξεχωριστά από μία άλλη. Αυτή η διαδικασία περιορισμού της επικοινωνίας κάνει το πλαίσιο ανάπτυξης (framework) πολύ πιο αξιόπιστο και αυτό το μοντέλο ανάπτυξης ονομάζεται Map/Reduce [10].

Οι εγγραφές εκτελούνται ατομικά από διεργασίες οι οποίες ονομάζονται Mappers. Η έξοδοι που προέρχονται από τους Mappers συγκεντρώνονται σε ένα δεύτερο σύνολο διεργασιών, οι οποίες ονομάζονται Reducers, όπου εκεί τα αποτελέσματα από διαφορετικούς Mappers μπορούν συγχωνεύονται για να σταλούν στην έξοδο δεδομένων [10].

Διαφορετικοί κόμβοι μέσα σε ένα σύμπλεγμα Hadoop εξακολουθούν να επικοινωνούν ο ένας με τον άλλον. Κομμάτια δεδομένων μπορούν να επισημανθούν με βασικά ονόματα, τα οποία ενημερώνουν το Hadoop για τον τρόπο με τον οποίο πρέπει να στείλει bits πληροφοριών σε έναν κοινό κόμβο προορισμού. Το Hadoop διαχειρίζεται εσωτερικά όλη τη διαδικασία μεταφοράς δεδομένων, καθώς και προβλήματα της τοπολογίας του συμπλέγματος [10].

Περιορίζοντας την επικοινωνία μεταξύ των κόμβων, το Hadoop κάνει το κατανεμημένο σύστημα πολύ πιο αξιόπιστο. Μεμονωμένα τα σφάλματα των κόμβων μπορούν να εργαστούν για την επανεκκίνηση διαδικασιών σε άλλα μηχανήματα [10].

2.3 Το Hadoop Distributed File System (HDFS)

Το Hadoop κατανεμημένο σύστημα αρχείων σχεδιάστηκε έτσι ώστε να διατηρεί δεδομένα μεγάλου όγκου (terabytes ή petabytes) και να παρέχει υψηλό throughput πρόσβασης στην πληροφορία. Τα αρχεία είναι αποθηκευμένα με έναν πλεονάζοντα τρόπο σε πολλαπλά μηχανήματα, έτσι ώστε να εξασφαλίζεται η ανθεκτικότητα σε σφάλματα καθώς και η υψηλή διαθεσιμότητα τους σε παράλληλες εφαρμογές [7] [8].

Το HDFS είναι ανθεκτικό σε έναν μεγάλο αριθμό προβλημάτων, σε αντίθεση με άλλα κατανεμημένα συστήματα αρχείων που είναι ευάλωτα, όπως το σύστημα αρχείων δικτύου (NFS). Ειδικότερα [8]:

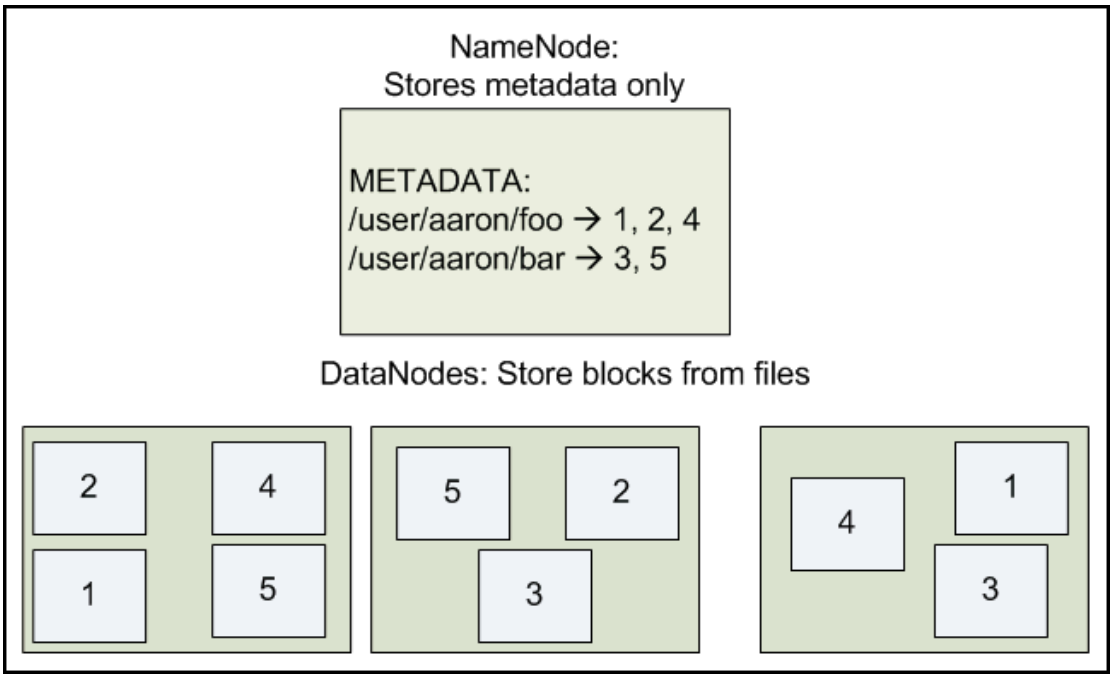
- Σχεδιάστηκε για την αποθήκευση μεγάλου όγκου δεδομένων, τα οποία είναι διαμοιρασμένα σε ένα σύνολο από μηχανήματα.
- Αν σε ένα μεμονωμένα από τα μηχανήματα του συμπλέγματος προκληθεί ένα σφάλμα τα δεδομένα θα πρέπει να είναι διαθέσιμα. Το HDFS εξασφαλίζει με αξιοπιστία ότι τα δεδομένα θα παραμείνουν διαθέσιμα.
- Παρέχει γρήγορα τη δυνατότητα επέκτασης πρόσβασης στην πληροφορία. Έχει την δυνατότητα να εξυπηρετήσει περισσότερους πελάτες, προσθέτοντας περισσότερα μηχανήματα στο σύμπλεγμα.
- Τέλος, μπορεί να διασυνδεθεί με το πλαίσιο ανάπτυξης Map/Reduce, έτσι ώστε να δίνεται η δυνατότητα να πραγματοποιηθεί η ανάγνωση και η εκτέλεση υπολογισμών στα δεδομένα τοπικά.

Το HDFS είναι σχεδιασμένο με βάση το σύστημα αρχείων της Google.

Το HDFS είναι ένα σύστημα αρχείων δομημένο σε blocks: ξεχωριστά αρχεία διασπώνται σε blocks συγκεκριμένου μεγέθους. Αυτά τα blocks αποθηκεύονται κατά μήκος ενός συμπλέγματος από ένα ή περισσότερα μηχανήματα με δυνατότητα αποθήκευσης δεδομένων. Ξεχωριστά μηχανήματα στο σύμπλεγμα αναφέρονται ως DataNodes. Ένα αρχείο μπορεί να αποτελείται από πολυάριθμα blocks, τα οποία δεν αποθηκεύονται απαραίτητα στο ίδιο μηχάνημα. Τα μηχανήματα-στόχοι που διατηρούν κάθε block επιλέγονται τυχαία σε μια block-by-block βάση. Έτσι η πρόσβαση σε ένα αρχείο μπορεί να απαιτεί τη συνεργασία

πολλαπλών μηχανημάτων, αλλά υποστηρίζει μεγέθη αρχείων κατά πολύ μεγαλύτερα από ένα DFS εγκατεστημένο σε ένα και μοναδικό μηχάνημα. Ξεχωριστά αρχεία μπορεί να απαιτούν περισσότερο χώρο από όσο θα μπορούσε να υποστηρίξει ένας και μόνο σκληρός δίσκος [7] [8]..

Εάν στη διαδικασία παροχής ενός αρχείου πρέπει να εμπλακούν περισσότερα του ενός μηχανήματα, τότε κάποιο αρχείο μπορεί να καταστεί μη διαθέσιμο σε περίπτωση απώλειας ενός εξ αυτών των μηχανημάτων. Το HDFS ξεπερνά αυτό το πρόβλημα αναπαράγοντας κάθε block κατά μήκος ενός αριθμού μηχανημάτων [7] [8]..



Εικόνα 1. **Τα DataNodes κρατούν τα blocks πολλαπλών αρχείων με έναν συντελεστή αντιγραφής = 2 . Ο NameNode αντιστοιχεί τα ονόματα αρχείων με τα block ids** [10]

Τα περισσότερα συστήματα αρχείων δομημένων σε blocks χρησιμοποιούν ένα μέγεθος block των 4 ή 8 KB. Αντίθετα το προκαθορισμένο μέγεθος block στο HDFS είναι 64 MB - πολλές τάξεις μεγέθους μεγαλύτερο. Αυτό επιτρέπει στο HDFS να μειώνει την ποσότητα αποθήκευσης μετα-δεδομένων που απαιτείται ανά αρχείο (η λίστα των blocks ανά αρχείο θα είναι μικρότερη καθώς το μέγεθος του κάθε block αυξάνεται) [7].

Το HDFS αποθηκεύει τα αρχεία ως ένα σετ από μεγάλα blocks κατά μήκος πολλών μηχανημάτων. Αυτά τα αρχεία δεν είναι μέρος ενός συνηθισμένου συστήματος αρχείων. Το HDFS εκτελείται σε ένα ξεχωριστό namespace, απομονωμένο από τα περιεχόμενα των τοπικών αρχείων. Τα αρχεία μέσα στο HDFS (ή πιο σωστά τα blocks από τα οποία αποτελούνται) αποθηκεύονται σε έναν συγκεκριμένο κατάλογο, τον οποίο διαχειρίζεται η υπηρεσία DataNode, τα αρχεία όμως θα ονομαστούν μόνο με ids των blocks. Η αλληλεπίδραση με τα αποθηκευμένα αρχεία HDFS χρησιμοποιώντας συνηθισμένα εργαλεία τροποποίησης των Linux δεν είναι δυνατή. Παρόλα αυτά, το HDFS διαθέτει τα δικά του εργαλεία για τη διαχείριση αρχείων, παρόμοια με τα οικεία μας. Είναι πολύ σημαντικό για το σύστημα αρχείων, να αποθηκεύονται τα μετα-δεδομένα αξιόπιστα. Επιπλέον, ενώ τα δεδομένα ενός αρχείου είναι προσβάσιμα για εγγραφή μία φόρα και για διάβασμα πολλές φορές, οι δομές των μετα-δεδομένων μπορούν να τροποποιηθούν από έναν μεγάλο αριθμό πελατών ταυτόχρονα. Είναι πάρα πολύ σημαντικό οι πληροφορίες να είναι απόλυτα συγχρονισμένες. Γι αυτόν ακριβώς τον λόγο όλα τα διαχειρίζεται ένα και μόνο μηχάνημα, το οποίο ονομάζεται NameNode [8]..

Ο NameNode είναι το μηχάνημα που αποθηκεύει όλα τα μετα-δεδομένα για το σύστημα αρχείων. Η πληροφορία που αποθηκεύεται στα μετα-δεδομένα είναι η εξής:

- Τα ονόματα των αρχείων
- Τα δικαιώματα των αρχείων
- Η τοποθεσία του κάθε block για κάθε αρχείο

Λόγω του μικρού όγκου μετα-δεδομένων ανά αρχείο, όλη αυτή η πληροφορία μπορεί να αποθηκευτεί στη μνήμη του μηχανήματος του NameNode, δίνοντας έτσι τη δυνατότητα γρήγορης πρόσβασης στα μετα-δεδομένα[7] [8]..

Για το άνοιγμα ενός αρχείο, ο πελάτης επικοινωνεί με τον NameNode και ανακτά μια λίστα από τοποθεσίες για τα blocks που περιλαμβάνει το αρχείο. Με αυτήν τη λίστα με τις τοποθεσίες, γίνεται δυνατή η αναγνώριση των DataNodes που κρατούν το κάθε block. Ο κάθε πελάτης διαβάζει απευθείας τα δεδομένα των

αρχείων από τους DataNode διακομιστές, με παράλληλο τρόπο. Ο NameNode δεν εμπλέκεται άμεσα σε αυτήν τη μαζική μεταφορά δεδομένων, διατηρώντας με αυτόν τον τρόπο τον φόρτο εργασίας του σε πολύ χαμηλά επίπεδα. Πρέπει να σημειωθεί ότι η πληροφορία που βρίσκεται στον NameNode θα πρέπει να διατηρηθεί ακόμα και σε περίπτωση σφάλματος του μηχανήματος [7] [8]..

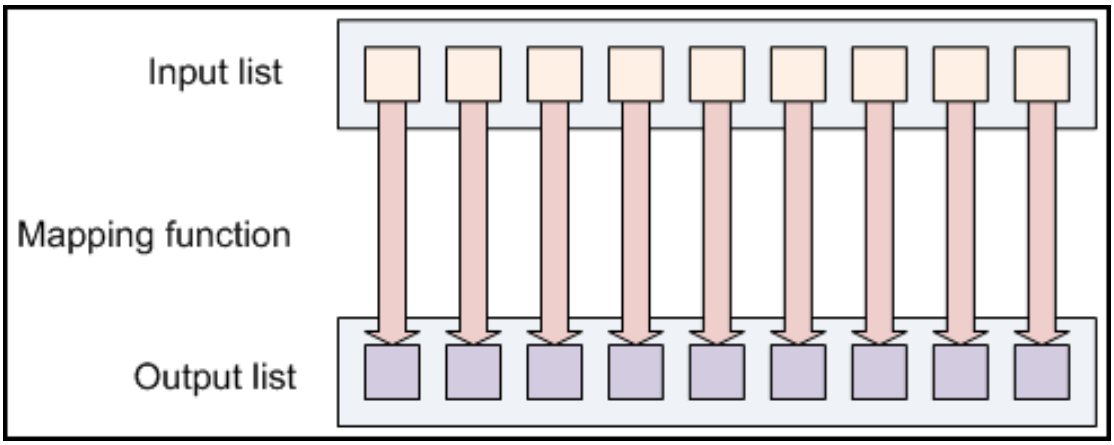
2.4 Βασικές αρχές του Map/Reduce

Ένα Map/Reduce πρόγραμμα έχει σχεδιαστεί για τον υπολογισμό μεγάλου όγκου δεδομένων που πραγματοποιείται όμως με παράλληλο τρόπο. Για να επιτευχθεί το παραπάνω απαιτείται η διαίρεση του φόρτου εργασίας σε έναν μεγάλο αριθμό μηχανημάτων. Αυτό το μοντέλο δεν έχει τη δυνατότητα να επεκταθεί σε πολύ μεγάλα συμπλέγματα (εκατοντάδες ή χιλιάδες κόμβους), εφόσον τα δεδομένα έχουν τη δυνατότητα να μοιραστούν στους κόμβους αυθαίρετα. Αντίθετα, στο Map/Reduce όλα τα στοιχεία δεδομένων είναι αμετάβλητα, κάτι το οποίο σημαίνει ότι δεν υπάρχει η δυνατότητα να πραγματοποιηθεί οποιαδήποτε ενημέρωση των δεδομένων του αρχείου. Αν σε μια διεργασία που ανήκει στον Mapper αλλάξουμε μία είσοδο (ζεύγος [key,value]), τότε αυτή η αλλαγή δεν θα επηρεάσει τα αρχεία εισόδου. Η επικοινωνία γίνεται μόνο με τη δημιουργία νέων ζευγών εξόδου ([key, value]), τα οποία διαβάζονται στη συνέχεια από το σύστημα Hadoop στην επόμενη φάση εκτέλεσης [3].

2.4.1 Επεξεργασία λιστών

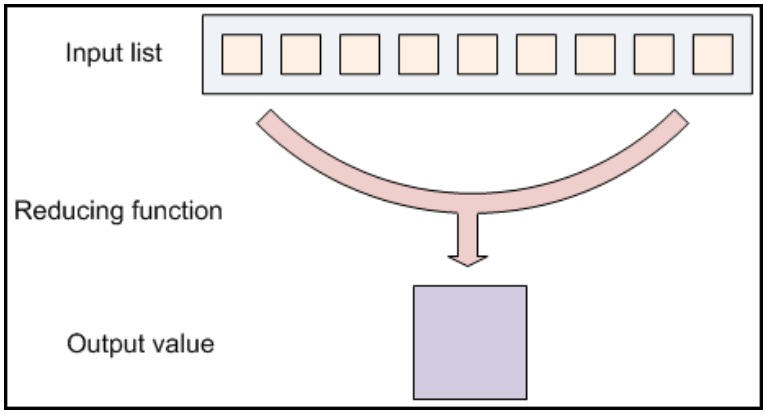
Κάθε πρόγραμμα Map/Reduce μετατρέπει τις λίστες εισόδου δεδομένων σε λίστες εξόδου δεδομένων. Αυτή η διαδικασία που περιγράφηκε θα εκτελεστεί δύο φορές μέσα σε ένα πρόγραμμα Map/Reduce, μία φορά κατά τη διαδικασία της Map και μία κατά τη Reduce [10].

Η πρώτη φάση, όπως αναφέρθηκε, ονομάζεται Map, στην οποία κάθε στιγμή παρέχεται μία λίστα από στοιχεία των δεδομένων, η οποία μετατρέπει κάθε στοιχείο ξεχωριστά σε ένα στοιχείο δεδομένων εξόδου [10].



Εικόνα 2. Η Map δημιουργεί μια νέα λίστα εξόδου εκτελώντας μια λειτουργία σε ξεχωριστά στοιχεία μιας λίστας εισόδου [10]

Η Reduce σου επιτρέπει να αθροίζεις τις τιμές μαζί. Ο Reducer λαμβάνει σαν είσοδο ένα σύνολο στοιχείων από μία λίστα εισόδου. Στη συνέχεια συνδυάζει αυτές τις τιμές, επιστρέφοντας μία μοναδική τιμή εξόδου [10].



Εικόνα 3. Η Reduce λαμβάνει ως είσοδο μία λίστα με values και παράγει ένα σύνολο τιμών ως έξοδο [10]

Το Reduce συχνά χρησιμοποιείται για να παράγει μία "περίληψη" δεδομένων, μετατρέποντας έναν μεγάλο όγκο δεδομένων σε μια μικρή περίληψη. Για παράδειγμα, να επιστρέψει το άθροισμα μιας λίστας με τιμές εισόδου [10].

Το πλαίσιο Hadoop Map/Reduce χρησιμοποιεί αυτό το πλάνο για να επεξεργαστεί μεγάλους όγκους πληροφοριών. Ένα πρόγραμμα Map/Reduce έχει δύο συστατικά:

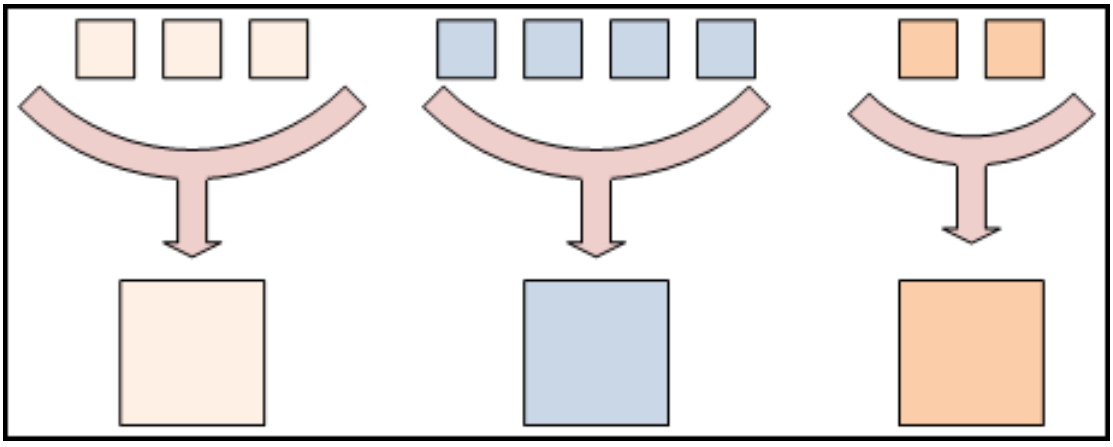
- Mapper
- Reducer

Keys-Values: Στο Map/Reduce μια value δεν υπάρχει μόνη της. Κάθε value έχει ένα key σχετικό με αυτήν. Τα keys προσδιορίζουν σχετικές values. Για παράδειγμα, καταγραφές ταχυμέτρου από διαφορετικά αυτοκίνητα θα μπορούσαν να πάρουν ένα key με βάση τον αριθμό πινακίδας και θα προέκυπταν τα εξής [10] :

```
AAA-123   65mph,
    12:00pm
ZZZ-789   50mph,
    12:02pm
AAA-123   40mph,
```

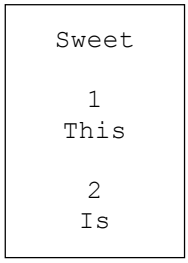
Οι λειτουργίες Map και Reduce δε λαμβάνουν απλώς values, αλλά key-value ζεύγη. Η έξοδος της καθεμιάς από αυτές τις λειτουργίες είναι η ίδια: και ένα key και ένα value πρέπει να παραδοθούν στην επόμενη λίστα στη ροή των δεδομένων [10] .

Τα keys διαιρούν το χώρο Reduce: Μια λειτουργία Reduce μετατρέπει μια μεγάλη λίστα από τιμές σε μία ή σε λίγες τιμές εξόδου. Στο Map/Reduce συνήθως οι τιμές εξόδου δεν μειώνονται όλες μαζί. Όλες οι τιμές με το ίδιο key παρουσιάζονται σε έναν Reducer μαζί. Αυτό γίνεται ανεξάρτητα από άλλες λειτουργίες Reduce που συμβαίνουν σε άλλες λίστες τιμών με διαφορετικά keys [10] .



Εικόνα 4. Διαφορετικά χρώματα αντιπροσωπεύουν διαφορετικά keys. Όλα τα values με το ίδιο key παρουσιάζονται σε μια μοναδική διεργασία Reduce [10]

Ένα παράδειγμα μίας τέτοιας εφαρμογής είναι η καταμέτρηση του αριθμού των διαφορετικών λέξεων που υπάρχουν σε μία σειρά από αρχεία. Ας υποθέσουμε ότι έχουμε τα αρχεία foo.txt και bar.txt., το αποτέλεσμα ενός Map/Reduce θα έπρεπε να είναι το εξής [10] :



Η δομή ενός προγράμματος Map/Reduce από υψηλότερο επίπεδο είναι της παρακάτω μορφής:

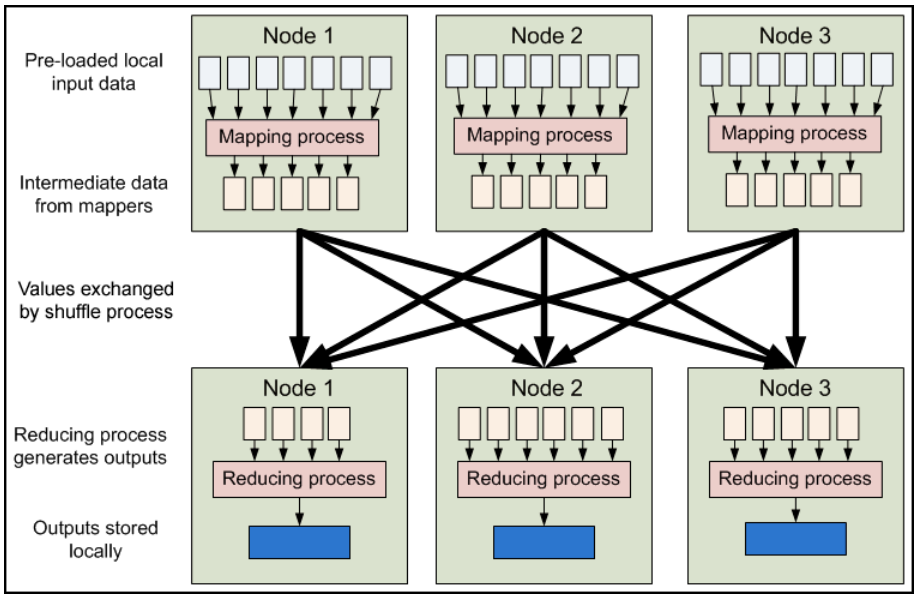
```
mapper (filename, file-contents):

for each word in file-contents:

 emit (word, 1)

reducer (word, values):

sum = 0

for each value in values:

sum = sum + value

emit (word, sum)
```

Πολλά στιγμιότυπα της λειτουργίας του Mapper δημιουργούνται σε διαφορετικά μηχανήματα μέσα στο σύμπλεγμα. Κάθε στιγμιότυπο λαμβάνει διαφορετικό αρχείο εισόδου (στην περίπτωση που έχουμε πολλά αρχεία εισόδου). Τα ζεύγη εξόδου των Mappers (π.χ. [word, 1]) στη συνέχεια προωθούνται στους Reducers. Αντίστοιχα και στην περίπτωση των Reducers, πολλά στιγμιότυπα της λειτουργίας του Reducer δημιουργούνται σε διαφορετικά μηχανήματα. Ο κάθε Reducer είναι υπεύθυνος για την επεξεργασία των τιμών που σχετίζονται με διαφορετική λέξη κάθε φορά. Στη συνέχεια ο Reducer τα συνοψίζει όλα αυτά σε μια τελική καταμέτρηση που συνδέεται με μία μόνο λέξη. Τέλος, ο Reducer στέλνει στην έξοδο το τελικός ζεύγος

με την καταμέτρηση [word, count], η οποία καταγράφεται σε ένα αρχείο εξόδου [10] [3].

2.4.2 Διάγραμμα ροής δεδομένων Map/Reduce

Τώρα που έχουμε δει τα συστατικά μέρη από τα οποία αποτελείται μια βασική διεργασία Map/Reduce, μπορούμε να δούμε πώς δουλεύουν όλα μαζί σε ένα υψηλότερο επίπεδο:



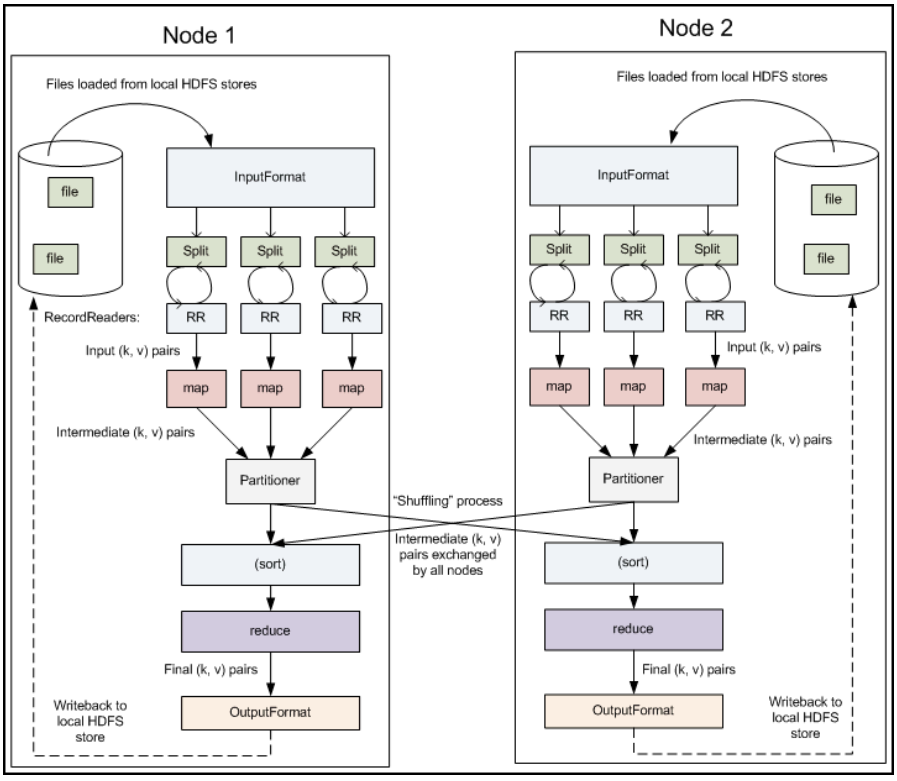
Εικόνα 5. Γενική όψη της ροής των δεδομένων σε ένα Map/Reduce [10]

Οι είσοδοι του Map/Reduce τυπικά προέρχονται από αρχεία εισόδου, τα οποία έχουν φορτωθεί στο HDFS του συμπλέγματος όπου γίνεται η επεξεργασία. Αυτά τα αρχεία είναι ισομερώς κατανεμημένα κατά μήκος όλων των κόμβων μας. Η εκτέλεση ενός προγράμματος Map/Reduce περιλαμβάνει την εκτέλεση διεργασιών Map σε πολλούς ή όλους τους κόμβους στο σύμπλεγμά μας. Αυτές οι διεργασίες Map είναι ισοδύναμες: κανένας Mapper δεν διαθέτει συγκεκριμένη “ταυτότητα” σχετική με αυτόν. Εξαιτίας αυτού, ο κάθε Mapper μπορεί να διαχειριστεί οποιοδήποτε αρχείο εισόδου. Κάθε Mapper φορτώνει ένα σύνολο αρχείων που βρίσκονται πιο κοντά στο μηχάνημα και τα επεξεργάζεται [10].

Όταν η φάση Map έχει ολοκληρωθεί, τα ενδιάμεσα ζεύγη key-value πρέπει να ανταλλαχθούν ανάμεσα στα μηχανήματα, ώστε να σταλούν όλες οι values με το ίδιο key στον ίδιο Reducer. Οι διεργασίες Reduce είναι διαμοιρασμένες κατά μήκος των

ίδιων κόμβων στο σύμπλεγμα με τους Mappers. Αυτό είναι το μόνο βήμα επικοινωνίας στο Map/Reduce. Η κάθε διεργασία Map δεν ανταλλάσσει πληροφορίες με άλλη, ούτε γνωρίζει η μία την ύπαρξη της άλλης. Με τον ίδιο τρόπο, διαφορετικές διεργασίες Reduce δεν επικοινωνούν μεταξύ τους. Ο χρήστης δεν επιλέγει ποτέ την πληροφορία που θα μεταφερθεί από ένα μηχάνημα σε άλλο. Όλη η μεταφορά δεδομένων πραγματοποιείται από την ίδια την πλατφόρμα Map/Reduce, οδηγούμενη από τα διαφορετικά keys σχετιζόμενα με τα values. Αυτό είναι ένα θεμελιώδες στοιχείο της αξιοπιστίας του Hadoop Map/Reduce. Εάν κάποιοι κόμβοι στο σύμπλεγμα αποτύχουν, θα πρέπει οι διεργασίες να έχουν τη δυνατότητα επανεκκίνησης. Εάν προκύψουν ανεπιθύμητες ενέργειες, για παράδειγμα επικοινωνία προς τα έξω, τότε η κατάσταση που μοιράζεται θα πρέπει να αποκατασταθεί σε μια διεργασία που θα επανεκκινηθεί. Περιορίζοντας την επικοινωνία και τις ανεπιθύμητες ενέργειες, οι επανεκκινήσεις μπορούν να γίνουν πιο ομαλά [10].

Στο παρακάτω διάγραμμα περιγράφεται αναλυτικά ένα παράδειγμα ενός Map/Reduce προγράμματος για την καταμέτρηση λέξεων:



Εικόνα 6. Λεπτομερής ροή δεδομένων Hadoop Map/Reduce [10]

Input Files: Σε αυτό εδώ το σημείο αποθηκεύονται τα δεδομένα για ένα Map/Reduce για πρώτη φορά. Συγκεκριμένα τα δεδομένα θα βρίσκονται στο HDFS [10].

InputFormat: Ο τρόπος με τον οποίο τα αρχεία εισόδου θα διαιρεθούν καθορίζεται από το InputFormat. Το InputFormat είναι μία κλάση του Hadoop πλαισίου η οποία παρέχει την εξής λειτουργία [10]:

- Επιλέγει τα αρχεία ή άλλα αντικείμενα που θα πρέπει να χρησιμοποιηθούν για είσοδο
- Καθορίζει τα InputSplits με τα οποία θα σπάσει το αρχείο σε διεργασίες
- Παρέχει σύνολο από RecordReader αντικείμενα που διαβάζουν τα αρχεία εισόδου

Ένα πλήθος από InputFormats παρέχονται από το πλαίσιο Hadoop, τα οποία είναι τα εξής:

InputFormat:	Περιγραφή:	Key:	Value:
TextInputFormat	Διάβασμα γραμμών αρχείου	Η μετατόπιση του byte της γραμμής	Το περιεχόμενο των γραμμών
KeyValueInputFormat	Ανάλυση των γραμμών σε ζευγάρια [key,value]	Όλη η πληροφορία μέχρι τον πρώτο χαρακτήρα tab	Το υπόλοιπο της γραμμής
SequenceFileInputFormat	Μία δυαδική μορφή υψηλής απόδοσης του Hadoop	Επιλεγμένο από τον χρήστη	Επιλεγμένο από τον χρήστη

Ο προκαθορισμένος τύπος αρχείου εισόδου είναι ο TextInputFormat, ο οποίος είναι χρήσιμος για μη δεδομένα όπως αρχεία καταγραφής (logs). Ο τύπος KeyValueInputFormat είναι χρήσιμος για την ανάγνωση της εξόδου μίας Map/Reduce διεργασίας, έτσι ώστε να χρησιμοποιηθεί ως είσοδος σε μία άλλη Map/Reduce διεργασία. Τέλος ο τύπος SequenceFileInputFormat διαβάζει ειδικά δυαδικά αρχεία που προορίζονται συγκεκριμένα και μόνο για χρήση στο Hadoop, τα οποία προσφέρουν την δυνατότητα της γρήγορης ανάγνωσης δεδομένων από τους Mappers [10].

InputSplits: Ο διαχωρισμός εισόδου, περιγράφει μία μονάδα εργασίας η οποία περιλαμβάνεται σε ένα Map/Reduce πρόγραμμα. Ένα Map/Reduce πρόγραμμα εφαρμόζεται σε ένα σύνολο από δεδομένα στα οποία εκτελούνται διάφορες διεργασίες. Μία Map διεργασία μπορεί να περιλαμβάνει την ανάγνωση ενός ολόκληρου αρχείου, συνήθως όμως γίνεται ανάγνωση ενός μέρους του αρχείου. Το κάθε αρχείο διαχωρίζεται σε κομμάτια (blocks) των 64MB, τα οποία επιτρέπουν σε πολλούς Mappers να μπορούν να επεξεργάζονται ένα αρχείο με παράλληλο τρόπο. Επίσης, δεδομένου ότι τα διάφορα κομμάτια (blocks) που συνθέτουν το αρχείο μπορούν να εξαπλωθούν σε διαφορετικούς κόμβους του συμπλέγματος, επιτρέπεται στις διεργασίες να γίνουν με τρόπο προγραμματισμένο σε κάθε έναν από τους διαφορετικού κόμβους. Τα επιμέρους τμήματα έχουν όλα υποστεί επεξεργασία σε τοπικό επίπεδο, αντί να χρειάζεται να μεταφερθούν από τον έναν κόμβο στον άλλον [3].

RecordReader: Ο RecordReader έχει φορτώσει τα δεδομένα και τα μετατρέπει σε ζεύγη [key,value], έτσι ώστε να είναι κατάλληλα για ανάγνωση από τον Mapper. Ο προκαθορισμένος τύπος εισόδου (TextInputFormat) παρέχει έναν LineRecordReader, ο οποίος αντιμετωπίζει κάθε γραμμή του αρχείου σαν μία νέα τιμή. Ο RecordReader μπορεί να εκτελεστεί επανειλημμένα μέχρι όλα τα κομμάτια εισόδου (InputSplits) να φτάσουν στο τέλος [10].

Mapper: Ο Mapper εκτελεί την καθορισμένη εργασία που έχει ορίσει ο χρήστης και είναι η πρώτη φάση του Map/Reduce προγράμματος. Η μέθοδος Map του Mapper είναι αυτή που θα προωθήσει τα δοθέντα ζευγάρια [key,value] στους Reducers. Επίσης δεν παρέχεται καμία μέθοδος επικοινωνίας μεταξύ των μεμονωμένων

Mappers. Αυτό επιτρέπει την αξιοπιστία της κάθε διεργασίας Map να διέπεται αποκλειστικά από την αξιοπιστία του τοπικού μηχανήματος [3].

Partition & Shuffle: Μετά το πέρας των διεργασιών Map, ο κάθε κόμβος πιθανόν να εκτελεί μερικές ακόμα διεργασίες Map. Επίσης παράλληλα ξεκινάει η ανταλλαγή των ενδιάμεσων εξόδων από τις διεργασίες Map στους Reducers. Αυτή η διαδικασία μεταφοράς της εξόδου των Maps προς τους Reducers ονομάζεται shuffling. Ένα διαφορετικό υποσύνολο από ενδιάμεσα keys ανατίθεται σε κάθε Reduce κόμβο (τα υποσύνολα είναι γνωστά ως partitions) και αποτελεί την είσοδο στις διεργασίες Reduce. Όλα τα values για το ίδιο key, πάντα εκτελούνται σε Reduce διεργασία μαζί ανεξάρτητα από ποιον Mapper προέρχεται. Παρόλα αυτά πρέπει όλοι οι Map κόμβοι να συμφωνήσουν για το πού θα σταλούν τα διαφορετικά κομμάτια των ενδιάμεσων δεδομένων. Ο Partitioner αποφασίζει ποιο κομμάτι ενός ζευγαριού [key,value] θα πάει που [3].

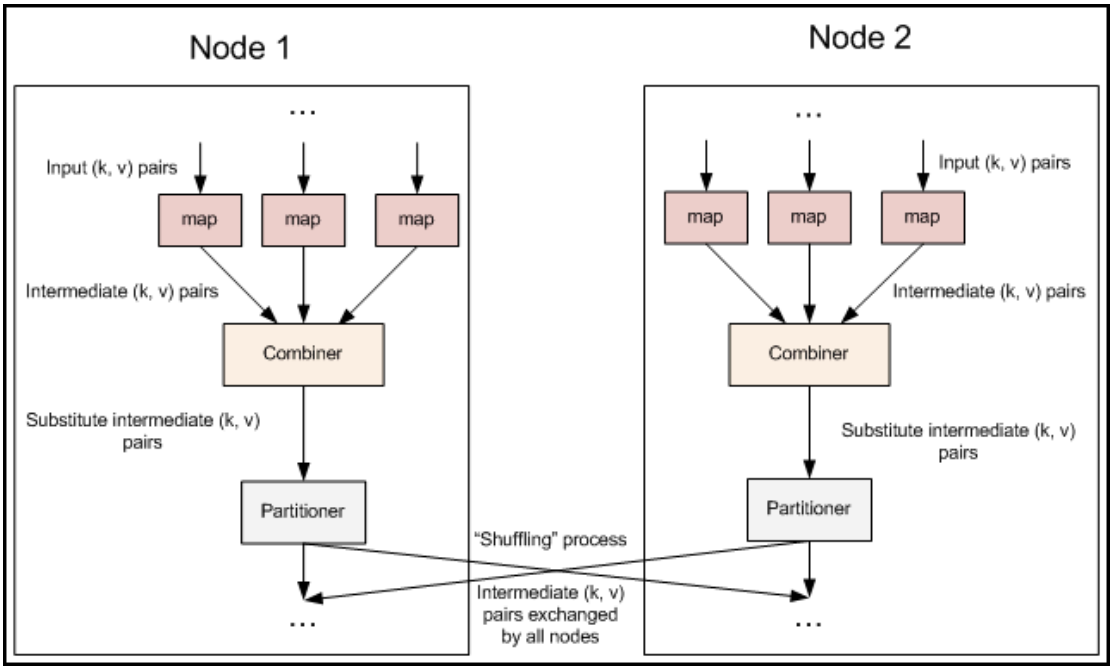
Sort: Κάθε Reduce διεργασία είναι υπεύθυνη για το Reduce των values που συνδέονται με ένα σύνολο από ενδιάμεσα keys. Το σύνολο των ενδιάμεσων keys ταξινομείται αυτόματα από το Hadoop πριν αυτά εμφανιστούν στον Reducer [3].

Reducer: Ο Reducer εκτελεί την καθορισμένη εργασία που έχει ορίσει ο χρήστης. Κάθε key που βρίσκεται μέσα στο partition ανατίθεται σε έναν Reducer. Αυτός λαμβάνει ένα key καθώς και ένα σύνολο από values που σχετίζονται με αυτό το key. Τέλος τα αρχεία εξόδου του Reducer καταγράφονται στο HDFS [3].

2.4.3 Επιπρόσθετη λειτουργία του Map/Reduce

Combiner: Η παραπάνω ροή που περιγράψαμε παραλείπει ένα βήμα στην επεξεργασία, το οποίο μπορεί να χρησιμοποιηθεί ώστε να βελτιστοποιηθεί η χρήση του Map/Reduce. Αυτό το σημείο εκτελείται μετά τον Mapper και πριν τον Reducer. Η χρήση του Combiner είναι προαιρετική. Εάν αυτό το πέρασμα είναι κατάλληλο για τον χρήστη, στιγμιότυπα από την κλάση του Combiner εκτελούνται σε κάθε κόμβο, στον οποίον έχουν εκτελεστεί διεργασίες Map. Ο Combiner θα λάβει σαν είσοδο όλα τα δεδομένα που εκπέμπονται από τον Mapper σε έναν δεδομένο κόμβο. Τα εξερχόμενα από τον Combiner στη συνέχεια στέλνονται στους Reducers, αντί

αυτών από τους Mappers. Ο Combiner είναι μια "mini-reduce" επεξεργασία, η οποία λειτουργεί μόνο με δεδομένα που δημιουργούνται σε ένα μηχάνημα [10].



Εικόνα 7. Προσθήκη Combiner στη ροή δεδομένων Map/Reduce [10]

Η καταμέτρηση λέξεων είναι ένα σαφές παράδειγμα, όπου ένας Combiner είναι χρήσιμος. Το πρόγραμμα καταμέτρησης λέξεων σε λίστες 1-3 εκπέμπει ένα ζεύγος (λέξη, 1) για κάθε στιγμιότυπο κάθε λέξεως που βλέπει. Συνεπώς εάν το ίδιο έγγραφο περιέχει τη λέξη "cat" 3 φορές, το ζεύγος ("cat", 1) εκπέμπεται 3 φορές. Όλες αυτά στη συνέχεια στέλνονται στον Reducer. Χρησιμοποιώντας έναν Combiner, αυτά τα ζεύγη μπορούν να συμπυκνωθούν σε ένα ("cat", 3), το οποίο θα σταλεί στον Reducer. Τώρα κάθε κόμβος στέλνει μία τιμή στον Reducer για κάθε λέξη, μειώνοντας δραστικά το συνολικό εύρος ζώνης που απαιτείται για τη διαδικασία shuffle και επιταχύνοντας την διεργασία. Το σημαντικό είναι ότι δεν χρειάζεται επιπλέον κώδικας για να λειτουργήσει αυτό το σημείο. Εάν μια λειτουργία Reduce είναι ταυτόχρονα " commutative" και "associative" τότε μπορεί να λειτουργήσει επιπρόσθετα και ως Combiner [10].

2.4.4 Ανοχή σε σφάλματα

Ένα από τα πλεονεκτήματα του Hadoop είναι η μεγάλη του ανοχή σε σφάλματα. Ακόμη και σε ένα μεγάλο σύμπλεγμα, όπου ο κάθε κόμβος ή το κάθε συστατικό του

δικτύου μπορεί να εμφανίζουν μεγάλο βαθμό αποτυχίας, το Hadoop μπορεί να οδηγήσει τις διεργασίες στην επιτυχημένη ολοκλήρωση [10].

Ο βασικός μηχανισμός με τον οποίο το Hadoop επιτυγχάνει την ανοχή στα σφάλματα είναι μέσω της επανεκκίνησης διεργασιών. Ο κάθε κόμβος διεργασίας (TaskTracker) είναι σε συνεχή επικοινωνία με τον κεντρικό κόμβο (JobTracker). Εάν ένας TaskTracker αποτύχει να επικοινωνήσει με τον JobTracker για κάποιο χρονικό διάστημα (προκαθορισμένο στο ένα λεπτό), ο JobTracker θα θεωρήσει ότι ο TaskTracker έχει καταρρεύσει. Ο JobTracker γνωρίζει ποιες Map και ποιες Reduce διεργασιες είχαν ανατεθεί στον κάθε TaskTracker [10].

Εάν η διεργασία βρίσκεται ακόμα στη φάση του Map , τότε θα ζητηθεί σε άλλους TaskTrackers να επανεκτελέσουν όλες τις Map διεργασίες, που προηγουμένως εκτελούνταν από τον TaskTracker που απέτυχε. Εάν η διεργασία βρίσκεται στη φάση Reduce, τότε σε άλλους TaskTrackers θα ανατεθεί να επανεκτελέσουν τις διεργασίες που ήταν σε εξέλιξη στον TaskTracker που απέτυχε [10].

Οι διεργασίες Reduce, αφού ολοκληρωθούν, γράφονται πίσω στο HDFS. Με αυτόν τον τρόπο, εάν ένας TaskTracker έχει ήδη ολοκληρώσει δύο από τις τρεις διεργασίες Reduce που του είχαν ανατεθεί, μόνον η τρίτη διεργασία θα πρέπει να εκτελεστεί αλλού. Οι διεργασίες Map είναι ελαφρώς πιο περίπλοκες: ακόμη και αν ένας κόμβος έχει ολοκληρώσει δέκα διεργασίες Map, οι Reducers μπορεί να μην έχουν όλοι αντιγράψει τα εισερχόμενά τους από τις εξόδους αυτών των Map διεργασιών. Εάν ένας κόμβος έχει καταρρεύσει, τότε τα εξερχόμενα του Mapper του είναι μη προσβάσιμα. Συνεπώς, οποιαδήποτε ολοκληρωμένη Map διεργασία πρέπει να επανεκτελεσθεί ώστε να είναι τα αποτελέσματά της διαθέσιμα στα υπόλοιπα μηχανήματα Reduce. Όλη αυτή η διαδικασία γίνεται αυτόματα από την πλατφόρμα Hadoop [10].

Αυτή η ανοχή στα σφάλματα υπογραμμίζει την ανάγκη η εκτέλεση του προγράμματος να μην περιέχει ανεπιθύμητες ενέργειες. Εάν οι Mappers και οι Reducers είχαν δικές τους ταυτότητες και επικοινωνούσαν μεταξύ τους, τότε η επανεκκίνηση μιας διεργασίας θα απαιτούσε από τους νέους κόμβους να επικοινωνούν με τα νέα στιγμιότυπα των Map και Reduce διεργασιών, και οι

διεργασίες που θα επανεκτελούνταν θα χρειαζόταν να αποκαταστήσουν την ενδιάμεσή τους κατάσταση. Αυτή η διαδικασία είναι προφανώς περίπλοκη και επιρρεπής σε σφάλματα. Το Map/Reduce απλοποιεί αυτό το πρόβλημα δραστικά περιορίζοντας τις ταυτότητες των διεργασιών ή την ικανότητα των συμμετεχόντων στις διεργασίες να επικοινωνούν μεταξύ τους. Μια μοναδική διεργασία βλέπει μόνον τα δικά της εισερχόμενα και γνωρίζει μόνον τα δικά της εξερχόμενα, ώστε να κάνει αυτήν τη διαδικασία αποτυχίας και επανεκκίνησης ανεξάρτητη και με διαφάνεια [10].

Speculative Execution: Ένα σημαντικό πρόβλημα του Hadoop είναι ότι διαιρώντας τις διεργασίες σε πολλούς κόμβους, είναι πιθανό να καθυστερήσει μερικούς κόμβους. Για παράδειγμα, αν κάποιος κόμβος έχει αργό ελεγκτή δίσκου, τότε είναι πολύ πιθανό να γίνει ανάγνωση των δεδομένων εισόδου στο 10% της ταχύτητας των άλλων κόμβων. Άρα όταν 99 Map διεργασίες είναι ολοκληρωμένες, το σύστημα περιμένει μέχρις ότου η τελευταία Map διεργασία εκτελεστεί, η οποία διαρκεί πολύ περισσότερο σε άλλους κόμβους [10].

Προκαλώντας την αναγκαστική εκτέλεση απομονωμένα η μία από την άλλη, η κάθε διεργασία δε γνωρίζει από πού έχει προέλθει η κάθε είσοδος. Οι διεργασίες θεωρούν ότι η πλατφόρμα Hadoop θα παραδώσει ακριβώς την κατάλληλη είσοδο σε αυτές. Ως εκ τούτου, η ίδια είσοδος μπορεί να δοθεί σε επεξεργασία πολλές φορές σε παράλληλο τρόπο για να εκμεταλλευτεί τις διαφορές στις δυνατότητες του κάθε μηχανήματος. Καθώς οι περισσότερες διεργασίες φτάνουν στο τέλος, η πλατφόρμα του Hadoop θα δημιουργήσει περιττά αντίγραφα από τις εναπομείνασες διεργασίες έτσι ώστε να τις μοιράσει στο σύνολο των κόμβων που δεν εκτελούν καμία διεργασία. Αυτή η διαδικασία ονομάζεται speculative execution. Όταν οι διεργασίες ολοκληρωθούν θα το ανακοινώσουν στον JobTracker. Όποιο αντίγραφο μίας διεργασίας τελειώσει πρώτα, γίνεται αυτόματα το οριστικό αντίγραφο [10].

2.5 Σχετικές ερευνητικές εργασίες

Υπάρχουν ερευνητικές εργασίες οι οποίες έχουν ασχοληθεί με την αντιμετώπιση προβλημάτων απόδοσης καθώς και ισομερούς κατανομής δεδομένων στο Map/Reduce. Η εργασία RanKloud είναι μία από αυτές που ασχολήθηκαν με το ίδιο αντικείμενο με τη παρούσα Διπλωματική εργασία. Η εργασία αυτή πραγματεύεται την αποδοτική επεξεργασία ερωτημάτων κατάταξης (Top-K Join). Ένας σημαντικός περιορισμός της είναι ότι δεν μπορεί να εγγυηθεί την ανάκτηση K αποτελεσμάτων και συγκεκριμένα είναι πιθανό να ανακτήσει λιγότερα αποτελέσματα από τα K που ζητήθηκαν [4]. Η εργασία On Saying "Enough Already!" in MapReduce κάνει μία προσπάθεια για την αντιμετώπιση του προβλήματος του EarlyTermination. Σε αυτήν προτείνεται η αποδοτική επεξεργασία ερωτημάτων κατάταξης (Top-K) χωρίς την εξοντωτική ανάγνωση του συνόλου των δεδομένων εισόδου για την επιστροφή αποτελεσμάτων [2]. Η εργασία A Survey of Large-Scale Analytical Query προτείνει ένα σύνολο από λύσεις για την αποδοτική επεξεργασία ερωτημάτων στο Map/Reduce, πέραν του Early Termination και του Load Balancing, οι οποίες όμως είναι ανάγκη να ερευνηθούν περαιτέρω [3].

ΚΕΦΑΛΑΙΟ 3: Παρουσίαση προβλήματος

3.1 Top-K Queries

Ο εντοπισμός δεδομένων υψηλής χρησιμότητας είναι γνωστός ως ranked / Top-K query processing. Ένα Top-K (q(k,f)) ερώτημα επιστρέφει τα K καλύτερα αποτελέσματα και είναι βασισμένο σε μία μονοτονική συνάρτηση βαθμολόγησης (scoring) f. Η πιο σημαντική και κοινή περίπτωση χρήσης συναρτήσεων βαθμολόγησης (scoring) είναι η γραμμική. Η συνάρτηση έχει ένα σχετικό βάρος w[i] βασισμένο στο ερώτημα για κάθε ένα από τα n χαρακτηριστικά βαθμολόγησης t[i] του αντικειμένου της βάσης δεδομένων και ο βαθμός (score) ενός αντικειμένου (ή πλειάδας) αποτελεί το άθροισμα των επιμέρους βαθμολογιών που δίνονται από τον παρακάτω τύπο [2]:

$$f(w) = \sum_{i=1}^{n} w[i] \cdot \tau[i]$$

Οι εφαρμογές στις οποίες οι χρησιμότητες των δεδομένων διαφέρουν, περιλαμβάνουν υποστήριξη αποφάσεων, ανάλυση κειμένου και media. Στην ανάλυση κειμένου για παράδειγμα, οι τιμές TF-IDF ή η συχνότητα των λέξεων μπορούν να θεωρηθούν ως τα αποτελέσματα χρησιμότητας των δεδομένων [4].

Ένα Top-K ερώτημα μπορεί πάντα να εκτελεστεί απλώς απαριθμώντας όλα τα αποτελέσματα και στη συνέχεια επιλέγοντας τα καλύτερα K. Όμως όταν ο αριθμός των υποψήφιων δεδομένων είναι μεγάλος και τα K είναι πολύ μικρότερα από τον αριθμό των υποψήφιων αποτελεσμάτων, είναι χρονοβόρο να βασιζόμαστε σε στρατηγικές επεξεργασίας που θα απαιτούσαν την απαρίθμηση όλων των πιθανών υποψήφιων δεδομένων από το σύστημα. Αντ'αυτού, για λειτουργίες ranked processing, όπως αναζήτηση k-nearest neighbor, k-best-nearest neighbor joins, Top-K joins, top-k group τα συστήματα επεξεργασίας δεδομένων είναι ανάγκη να χρησιμοποιήσουν δομές δεδομένων και αλγορίθμους που να αποκλείουν δεδομένα που δε χρειάζεται να συμπεριληφθούν, χωρίς να χρειάζεται να τα αξιολογούν [4].

Οι περισσότεροι από τους υπάρχοντες αλγορίθμους, οι οποίοι παρέχουν τις ανωτέρω λειτουργίες υποθέτουν ότι μία ή δύο από από τις ακόλουθες στρατηγικές πρόσβασης σε δεδομένα είναι διαθέσιμη:

- Πρόσβαση ταξινόμησης

  Το σύστημα έχει πρόσβαση στα δεδομένα εισόδου σταδιακά, για παράδειγμα, οι Top-K Join αλγόριθμοι, όπως οι FA, TA και ο NRA θεωρούν ότι τα δεδομένα εισόδου είναι διαθέσιμα κατά σειρά χρησιμότητας - η pipelined πρόσβαση στα ταξινομημένα δεδομένα βοηθάει ώστε να ταυτοποιηθούν τα υποψήφια δεδομένα και να εξαιρεθούν τα μη πιθανά αποτελέσματα πιο γρήγορα [9].

- Πρόσβαση ευρετηρίου / συμπλέγματος

  Για κάθε υποψήφιο αντικείμενο, το σύστημα μπορεί γρήγορα να ταυτοποιήσει το αντίστοιχο score χρησιμότητας, βασιζόμενο σε μια δομή ευρετηρίου για αποδοτική αναζήτηση του score χρησιμότητας. Η δομή ευρετηρίου επίσης βοηθάει στο να ταυτοποιηθούν παρόμοια αντικείμενα γρήγορα και στο να ξεχωρίσουν τα άσχετα αντικείμενα πιο γρήγορα. Παραδείγματος χάριν, οι αλγόριθμοι που αντιμετωπίζουν τα k-nearest neighbor και k-closest pair queries θεωρούν ότι τα δεδομένα ταξινομούνται χρησιμοποιώντας καμπύλες πλήρωσης χώρου ή ότι διαχωρίζονται χρησιμοποιώντας δομές ευρετηρίου (όπως τα R-trees, KD-trees και διαγράμματα Voronoi) ή χρησιμοποιώντας μικτές λειτουργίες. Οι αλγόριθμοι Skyline επιπλέον περιλαμβάνουν αλγορίθμους πρόσβασης ταξινόμησης και ευρετηρίου. Παρά ταύτα, η εφαρμογή αυτών των λειτουργιών σε ένα σύστημα όπως το Hadoop, που υποστηρίζει υψηλούς βαθμούς παραλληλισμού, απαιτεί περαιτέρω έλεγχο [9].

  Το πλαίσιο Map/Reduce του Hadoop μπορεί να εκτελεί joins ανάμεσα σε μεγάλα σύνολα δεδομένων χρησιμοποιώντας είτε join από την πλευρά του Map είτε join από την πλευρά του Reduce. [9]

3.2 Early Termination

Στο πλαίσιο Hadoop, από τον σχεδιασμό του, οι διεργασίες Map πρέπει να έχουν πρόσβαση στα δεδομένα εισόδου στο σύνολο τους, πριν οποιαδήποτε Reduce διεργασία ξεκινήσει την επεξεργασία. Παρ' όλα αυτά, το παραπάνω είναι αποδοτικό μόνο για πολύ συγκεκριμένα ερωτήματα προς τα δεδομένα έτσι ώστε να παράγουν το τελικό αποτέλεσμα. Σε ένα μεγάλο σύνολο ερωτημάτων προς τα δεδομένα, συνήθως μόνο ένα υποσύνολο των δεδομένων είναι απαραίτητο για να παραχθούν σωστά αποτελέσματα [3].

Ένα τυπικό παράδειγμα στο οποίο εμφανίζεται το παραπάνω, είναι τα Top-K ή Top-K Join ερωτήματα στα οποία δεν είναι απαραίτητο να διαβάσουμε όλα τα δεδομένα. Θα αρκούσε να είναι γνωστά κάποια όρια τιμών ή μια δειγματοληψία των τιμών για να παραχθεί το σωστό αποτέλεσμα γρηγορότερα χωρίς να είναι αναγκαία η ανάγνωση όλων των δεδομένων κατά τη διάρκεια της Map. Ένας τέτοιος τύπος ερωτήματος είναι σύνηθες να εμφανίζεται σε δεδομένα ανάλυσης και δεν είναι εφικτό να επεξεργαστούν αποδοτικά από το Map/Reduce [3].

Το συμπέρασμα λοιπόν είναι η έλλειψη ενός μηχανισμού πρόωρου τερματισμού των διεργασιών της Map, όταν ήδη υπάρχει αρκετή πληροφορία για την εξαγωγή των αποτελεσμάτων (π.χ ιστογράμματα, όρια, δειγματοληπτικές τιμές κ.λπ.) σε Ranking ερωτήματα (Top-K, Top-K Join etc.) [3].

3.2.1 Προτεινόμενη και υλοποιημένη λύση

Στη παρούσα εργασία για την επίλυση του Early Termination σε Top-K Join προβλήματα αναπτύχθηκε μία σειρά εργασιών. Σκοπός του Early Termination είναι να προσφέρει γρήγορα αποτελέσματα χωρίς να γίνει επξεργασία όλων των δεδομένων.

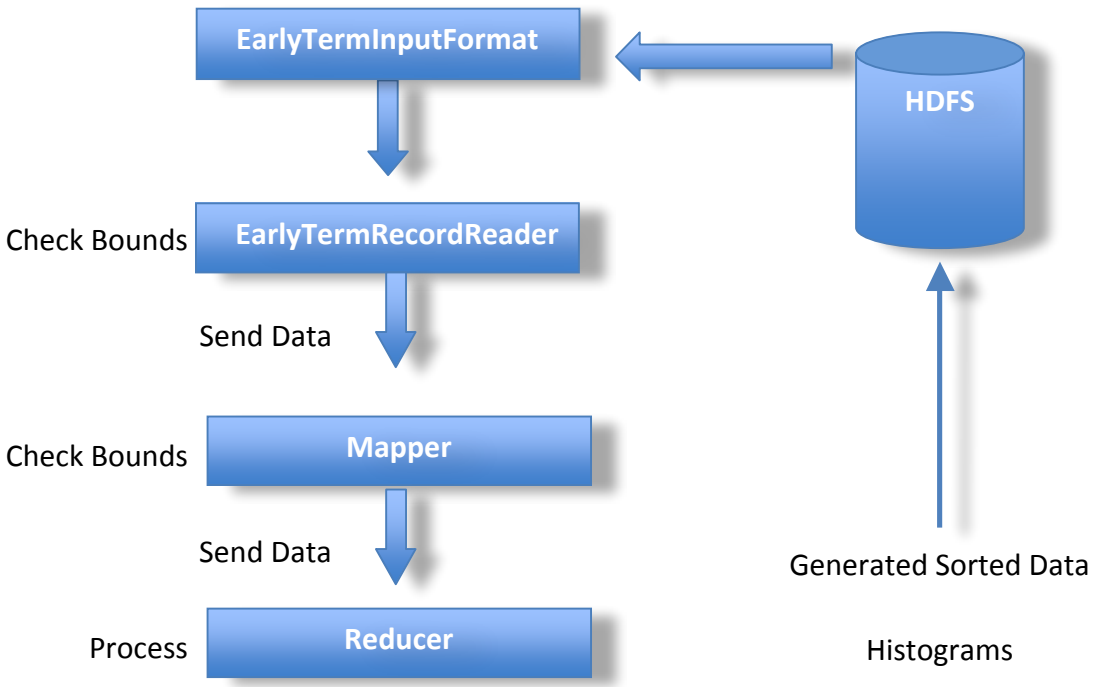
Για να επιτευχθεί το παραπάνω δημιουργήθηκαν μεγάλου όγκου δεδομένα τα οποία είναι ταξινομημένα σύμφωνα με τα scores των εγγραφών. Η υλοποίηση χρησιμοποιεί στατιστικά στοιχεία με τη μορφή ιστογραμμάτων έτσι ώστε να αντλήσει επαρκή όρια scoring που παράγουν τα σωστά Top-K αποτελέσματα. Τα όρια (εύρος τιμών των scores) καθορίζουν τις πλειάδες που πρέπει να ανακτηθούν από κάθε ένα από τα κομμάτια δεδομένων και τα όρια αποφασίζουν για το ποιες

πλειάδες πρέπει να προσπελαστούν, κάτι το οποίο θα αναλυθεί με συγκεκριμένο παράδειγμα στο επόμενο κεφάλαιο.

Έχοντας τα όρια των τιμών αυτών έγινε δυνατό κατά την εκτέλεση του RecordReader, που γίνεται πριν την Map, όπως αναφέρθηκε και προηγουμένως, να ελέγχουμε τα όρια τα οποία έχουν παραχθεί από τα ιστογράμματα και να μην στέλνουμε όλα τα δεδομένα για επεξεργασία στην Map και κατα συνέπεια και στην Reduce.

Η διαδικασία που ακολουθήθηκε είναι η εξής:

- Παραγωγή μεγάλου όγκου δεδομένων
- Ταξινόμηση με βάση τα scores
- Παραγωγή ιστογραμμάτων
- Υπολογισμός ορίων με βάση τις τιμές των ιστογραμμάτων
- Έλεγχος ορίων στον RecordReader
- Αποστολή στον Mapper
- Έλεγχος ορίων για αποστολή σε reducer



3.3 Load Balancing

Παράλληλα συστήματα διαχείρισης δεδομένων προσπαθούν να ελαχιστοποιήσουν τον χρόνο εκτέλεσης σύνθετων διεργασιών επεξεργασίας με το κατάλληλο Partitioning των δεδομένων εισόδου και τον διαμοιρασμό του φορτίου επεξεργασίας στα μηχανήματα. Αν τα δεδομένα διανέμονται με δίκαιο τρόπο, ο χρόνος εκτέλεσης της πιο αργής μηχανής θα υπερέχει στο συνολικό χρόνο εκτέλεσης. Το πιο σημαντικό , ακόμη και όταν τα δεδομένα κατανέμονται ισομερώς στα διαθέσιμα μηχανήματα, είναι ότι η ίση εκτέλεση επεξεργασίας μπορεί να μην είναι πάντα εγγυημένη. Άρα μέρος του προβλήματος, το οποίο εντοπίζεται στην επεξεργασία που θα εκτελεστεί από τους Reducers, είναι το Partitioning των δεδομένων με έντιμο τρόπο, έτσι ώστε σε κάθε μηχάνημα να γίνεται ανάθεση ισοδύναμου αριθμού δεδομένων [3].

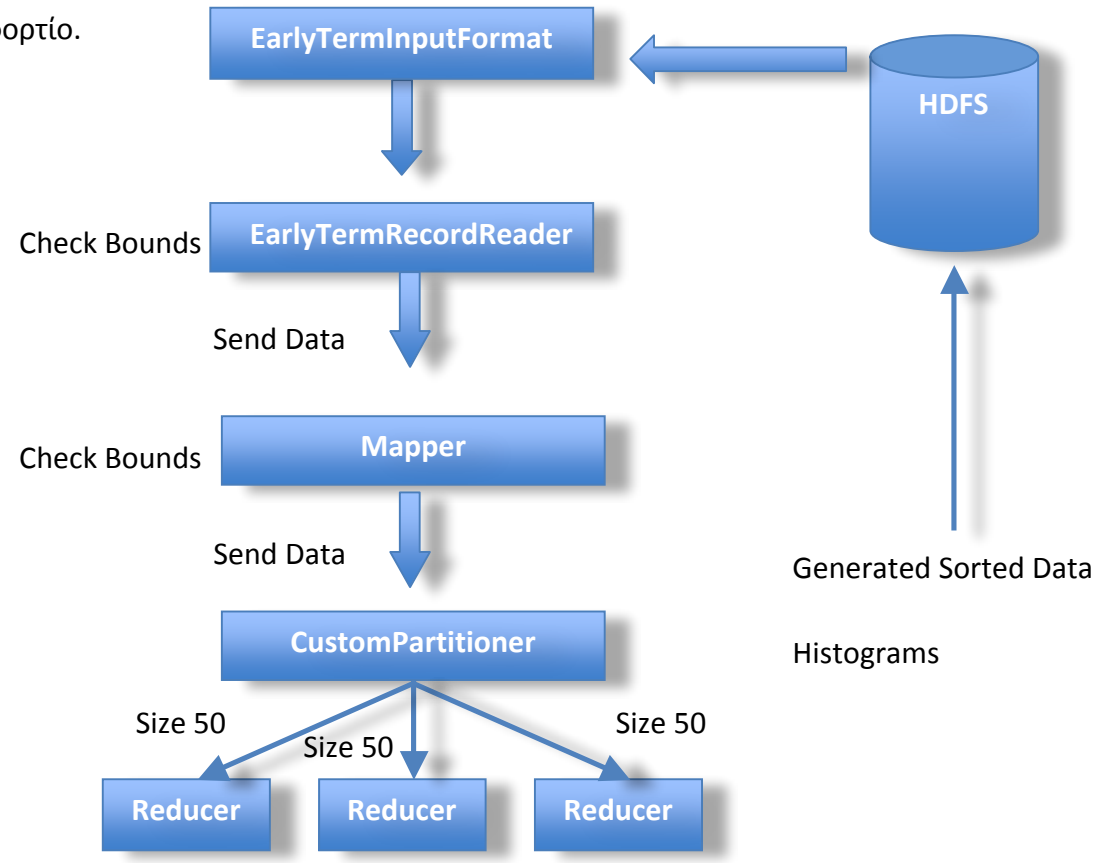
Το συμπέρασμα λοιπόν είναι ότι η παροχή προηγμένων μηχανισμών εξισορρόπησης φορτίου που στοχεύουν στην αύξηση της αποτελεσματικότητας της επεξεργασίας του ερωτήματος, εκχωρώντας ίσα μερίδια, χρήσιμα για επεξεργασία στα διαθέσιμα μηχανήματα, είναι μια αδυναμία του Map/Reduce που δεν έχει αντιμετωπιστεί ακόμη [3].

3.3.1 Προτεινόμενη και υλοποιημένη λύση

Στη παρούσα εργασία για την επίλυση του προβλήματος Load Balancing αναπτύχθηκε ένας custom Partitioner όπου πραγματοποιεί τον δίκαιο διαμοιρασμό των κομματιών των δεδομένων σύμφωνα με τον αριθμό των Reducers αλλά και το σύνολο που ο κάθε Reducer θα λάβει για να εξυπηρετήσει.

Πριν εκτελεστεί οποιαδήποτε εργασία για να ξεκινήσει το Map/Reduce, εφαρμόστηκε το παραπάνω. Εφόσον είναι γνωστός ο αριθμός των Reducers από την είσοδο του προγράμματος, είναι γνωστός ο αριθμός των εγγραφών για κάθε key, εφόσον υπολογίζεται από τα ιστογράμματα που αναφέραμε προηγουμένως, γίνεται χρήση ενός γνωστού αλγορίθμου οποίος ονομάζεται Longest Processing Time (LPT) με τον οποίο κατανέμουμε σε κάθε Reducer δίκαια και ισομερώς όλα τα keys.

Αυτό που επιτυγχάνεται με τον LPT είναι να φορτώνονται με κομμάτια δεδομένων ισόποσα όλοι οι Reducers με προτεραιότητα αυτούς που έχουν το μικρότερο φορτίο.



ΚΕΦΑΛΑΙΟ 4: Σχεδίαση και Υλοποίηση των εργασιών

4.1 Γενική Περιγραφή

Στην συνέχεια θα περιγραφεί η σχεδίαση και η υλοποίηση των εργασιών για όλα τα συστατικά από τα οποία αποτελείται ένα Top-K Join Map/Reduce πρόγραμμα. Όπως αναφέραμε προηγουμένως η υλοποίηση στηρίζεται στα εξής:

- Δημιουργία κατάλληλων δεδομένων για επεξεργασία
- Δημιουργία ιστογραμμάτων σύμφωνα με τα δεδομένα
- Ταξινόμηση των δεδομένων με βάση τα scores
- Υπολογισμό ορίων με βάση τις τιμές των ιστογραμμάτων
- Έλεγχο ορίων στον RecordReader
- Αποστολή στη φάση Map
- Έλεγχο ορίων στη φάση Map για αποστολή σε Reducer

4.2 Περιγραφή υλοποίησης

4.2.1 Περιγραφή δημιουργίας παραγωγής δεδομένων

Όπως θα αναλυθεί και θα παρουσιαστεί στο επόμενο κεφάλαιο, για το πείραμα δημιουργήθηκαν δύο πίνακες δεδομένων μεγάλου μεγέθους. Οι γεννήτριες παραγωγής δεδομένων χρησιμοποιούν Zipfian κατανομή έτσι ώστε η τυχαία παραγωγή των Join values να μην είναι ομοιόμορφη, για αυτό άλλωστε όπως θα δούμε και στην συνέχεια του επόμενου κεφαλαίου των πειραμάτων, μία παράμετρο αποτελεί το skew των δεδομένων. Στη συνέχεια ακολουθεί λεπτομερές παράδειγμα της χρήσης της Zipfian κατανομής στα δεδομένα που χρησιμοποιήθηκαν για να γίνει απόλυτα κατανοητό:

Αν η κατανομή ήταν ομοιόμορφη, δηλαδή skew = 0:

J1 ~100.000

J2 ~100.000

...

J10 ~100.000

=========

1.000.000

Αν η κατανομή δεν ήταν ομοιόμορφη, δηλαδή skew > 0:

J1 ~500.000

J2 ~200.000

...

J10 ~1.000

=========

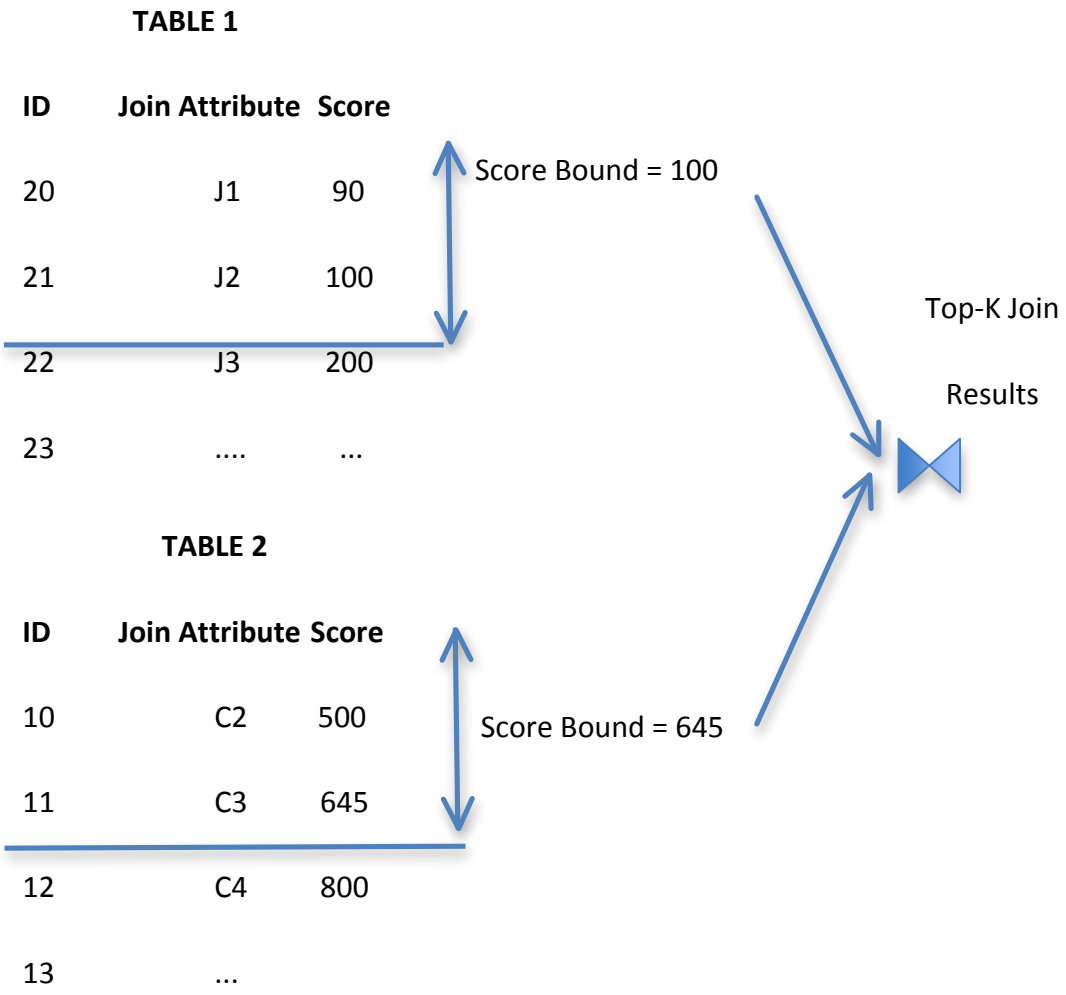
1.000.000

Βλέποντας το παραπάνω παράδειγμα, φαίνεται ότι στην πρώτη περίπτωση με skew = 0 όλα τα Join χαρακτηριστικά έχουν ακριβώς τον ίδιο αριθμό πλειάδων, ενώ με skew > 0, ο αριθμός πλειάδων δεν είναι ισομερώς κατανεμημένος, είναι ανομοιόμορφος για κάθε Join χαρακτηριστικό.

Σημαντικό είναι να σημειώσουμε, ότι στη συνέχεια τα δεδομένα ταξινομούνται με βάση τα scores και η ταξινόμηση των δεδομένων είναι ανεξάρτητη από το εκάστοτε ερώτημα, δηλαδή τα δεδομένα ταξινομούνται μόνο μία φορά και μετά είναι δυνατή η επεξεργασία οποιουδήποτε Top-K ερωτήματος.

4.2.2 Περιγραφή και κατανόηση ορίων

Για την καλύτερη κατανόηση των ορίων που θα παραχθούν από τα στατιστικά που έχουμε δημιουργήσει με τη μορφή ιστογραμμάτων, παρουσιάζεται ένα αναλυτικό παράδειγμα:



Βλέποντας το παραπάνω παράδειγμα είναι ξεκάθαρο πως τα όρια εγγυώνται ότι εάν διαβαστεί το Table 1 έως το Bound = 100 και το Table 2 έως το Bound = 645, τότε σίγουρα δεν υπάρχει καμία πλειάδα από τους 2 αυτούς πίνακες δεδομένων που να μπορεί να δώσει Top-K αποτελέσματα και να μην έχουν διαβαστεί.

4.2.3 Τρόπος υπολογισμού ορίων

Ο υπολογισμός των ορίων γίνεται βάσει στατιστικών με τη μορφή ιστογραμμάτων που έχουν παραχθεί από τα δεδομένα εισόδου. Τα ιστογράμματα των σχέσεων N του κάθε πίνακα διερευνώνται με Round-Robin τρόπο. Σε κάθε επανάληψη

ανακτούμε το κάθε επόμενο bin από τα ιστογράμματα του κάθε πίνακα ξεχωριστά. Το ανακτώμενο bin προστίθεται σε μία λίστα από ήδη ανακτημένα bins του κάθε πίνακα και στη συνέχεια το μόλις ανακτημένο bin συνδυάζεται με τα ανακτημένα bins των σχέσεων και οι έγκυροι Join συνδυασμοί από bins αποθηκεύονται σε μία ουρά. Κάθε φόρα για να ανακτήσουμε το score ελέγχουμε την ουρά με τους έγκυρους Join συνδυασμούς από bins και μας επιστρέφεται το score της K-οστής Join πλειάδας, τέτοιο ώστε ο αριθμός των πλειάδων των Joined bins που έχουν score μικρότερο ή ίσο του K-στού Join score να έχει άθροισμα μεγαλύτερο από το K. Το πραγματικό score της K-οστής Joined πλειάδας είναι μικρότερο ή ίσο του score της K-οστής Joined πλειάδας. Επίσης υπολογίζεται ο συνολικός αριθμός των πλειάδων όλων των Joined bins στην ουρά. Επιπλέον το κατά εκτίμηση όριο score για κάθε πίνακα έχει οριστεί να είναι ίσο με το υψηλότερο score του bin, υποδηλώνοντας ότι οι πλειάδες του τελευταίου bin του ιστογράμματος του κάθε πίνακα συνεισφέρουν στο Join αποτέλεσμα. Επίσης υπολογίζεται ένα κατώφλι το οποίο είναι το καλύτερα δυνατό score που μπορεί να παραχθεί από μη αναγνωσμένα bins. Η επανάληψη αυτή τερματίζεται όταν τα K Joined αποτελέσματα έχουν παραχθεί και κανένα από τα μη αναγνωσμένα bins δεν μπορεί να παράγει καμία πλειάδα με καλύτερο score από το καλύτερο K-οστό Joined αποτέλεσμα που έχει ανακτηθεί μέχρι τώρα.

4.2.4 Δημιουργία τροποποιημένου RecordReader και έλεγχος ορίων τιμών

Όπως αναφέρθηκε σε προηγούμενο κεφάλαιο το προκαθορισμένο **InputFormat** εκτελεί τις τρεις από τις προηγούμενες διαδικασίες που αναφέρθηκαν παραπάνω:

- Επιλέγει τα αρχεία ή άλλα αντικείμενα που θα πρέπει να χρησιμοποιηθούν για είσοδο.
- Καθορίζει τα InputSplits.
- Παρέχει ένα σύνολο από RecordReader αντικείμενα που διαβάζουν τα αρχεία εισόδου έτσι ώστε να αποσταλούν στην φάση Map.

Η υλοποίηση αρχικά επικεντρώνεται στο τρίτο βήμα και στην τροποποίηση της τεχνικής με την οποία γίνεται η ανάγνωση των αρχείων εισόδου από τον RecordReader καθώς και η τροφοδότηση της Map με συγκεκριμένες εγγραφές. Γι' αυτόν το λόγο δημιουργήθηκε ένας καινούριος τύπος **InputFormat** ο οποίος ονομάστηκε **EarlyTermInputFormat.**

Για να χρησιμοποιηθεί η υλοποίηση του **EarlyTermInputFormat** μέσα σε μία εργασία Map/Reduce, πρέπει κατά τη δημιουργία της να οριστεί η νέα κλάση προγράμματος που θα αποτελεί το νέο InputFormat. Για παράδειγμα:

```
job.setInputFormatClass(EarlyTermInputFormat.class);
```

Η υλοποίηση του EarlyTermInputFormat είναι η εξής:

```
public class EarlyTermInputFormat extends TextInputFormat {
    @Override
    public RecordReader<LongWritable, Text>
createRecordReader(InputSplit split, TaskAttemptContext context) {
      return new EarlyTermRecordReader();
    }
    @Override
    protected boolean isSplitable(JobContext context, Path filename)
{
      return true;
    }
}
```

Όπως αναφέρθηκε παραπάνω το InputFormat παρέχει ένα σύνολο από RecordReader αντικείμενα που διαβάζουν τα αρχεία εισόδου για να αποστείλουν τις πλειάδες των εγγραφών στη φάση Map. Η προκαθορισμένη λειτουργία του RecordReader είναι να αποστέλλει όλες τις πλειάδες εγγραφών στη φάση Map. Όπως έχει αναφερθεί σε προηγούμενο κεφάλαιο, σε Top-K ερωτήματα δεν είναι καθόλου αποδοτική η ανάγνωση όλων των εγγραφών, παρά μόνον αυτών που είναι χρήσιμες και ικανές να επιστρέψουν τα καλύτερα αποτελέσματα.

Για τον λόγο αυτόν υλοποιήθηκε ο **EarlyTermRecordReader**, ο οποίος αντικαθιστά την προκαθορισμένη υλοποίηση του RecordReader του Hadoop, έτσι ώστε να επιτευχθεί η τροφοδότηση της Map μόνο με εγγραφές που είναι κατάλληλες και χρήσιμες για την εξαγωγή των καλύτερων αποτελεσμάτων σε Top-K ερωτήματα. Αυτό επιτυγχάνεται με τον υπολογισμό ορίων, τα οποία έχουν υπολογιστεί από στατιστικά δεδομένα με τη μορφή ιστογραμμάτων, που θα αναλυθούν λεπτομερώς

στη συνέχεια. Ένα παράδειγμα ενός τροποποιημένου **RecordReader** είναι η υλοποίηση του **EarlyTermRecordReader**:

```
public class EarlyTermRecordReader extends
RecordReader<LongWritable, Text>{

	@Override
	public void initialize(InputSplit genericSplit,
TaskAttemptContext context)throws IOException, InterruptedException
{

	}

	@Override
	public boolean nextKeyValue() throws IOException,
InterruptedException {
	}
}
```

4.3 Επίτευξη στόχου

Με αυτόν τον τρόπο που περιγράφηκε αναλυτικά στο προηγούμενο παράδειγμα επιτεύχθηκε ο πρόωρος τερματισμός της εργασίας για ένα Top-K Join ερώτημα και κατά συνέπεια δεν πραγματοποιείται η ανάγνωση ολόκληρου του συνόλου των δεδομένων εισόδου.

Αφού εκτελεστούν τα παραπάνω, οι πλειάδες των εγγραφών που συγκεντρώθηκαν αποστέλλονται στην Map διεργασία. Μέσα στην Map για κάθε σχέση που υπάρχει στα Join δεδομένα γίνεται επανέλεγχος για το κατά πόσον το score της κάθε εγγραφής ξεπερνάει τα όρια που έχουν υπολογιστεί έτσι ώστε να αποσταλούν στον Reducer.

Όπως αναφέρθηκε στα προηγούμενα κεφάλαια, μετά την αποστολή των εγγραφών από την Map παρεμβάλλεται ο Partitioner, ο οποίος αποφασίζει ποια κομμάτια δεδομένων θα σταλούν σε κάθε reducer. Αυτή η διαδικασία δε γίνεται με απόλυτα αποδοτικό τρόπο από πλευράς εξισορρόπησης φορτίου δεδομένων που θα λάβει ο κάθε Reducer. Για να βελτιστοποιηθεί ο τρόπος με τον οποίο κατανέμονται τα κομμάτια δεδομένων στον κάθε Reducer, έτσι ώστε ο φόρτος εργασίας να είναι ισομερής για όλους, υλοποιήθηκε ένας νέος Partitioner.

Για να είναι δυνατό ο νέος Partitioner να αποφασίσει πού θα στείλει τα κάθε κομμάτια δεδομένων, θα πρέπει πριν από οποιαδήποτε εργασία Map/Reduce να έχει υπολογιστεί για κάθε Join σχέση ο συνολικός αριθμός εγγραφών και να προαποφασιστεί ο Reducer που θα του ανατεθεί το κάθε κομμάτι. Για να μοιραστεί αποδοτικά το φορτίο, αναλογικά με τον αριθμό των Reducers χρησιμοποιήθηκε ο Longest Processing Time αλγόριθμος. Ένα παράδειγμα υλοποίησης ενός τροποποιημένου Partitioner είναι ο παρακάτω:

```
public class LoadBalancePartitioner extends
Partitioner<IntWritable, MyKeyValuePair> implements Configurable{

	private Configuration configuration;

	@Override
	public int getPartition(IntWritable key, MyKeyValuePair
value, int numPartitions){

	}

	@Override
	public void setConf(Configuration configuration) {
		this.configuration = configuration;
	}

	@Override
	public Configuration getConf() {
		return configuration;
	}
}
```

Η χρήση του σε ένα Map/Reduce πρόγραμμα πρέπει να γίνει κατά τη δημιουργία ενός νέου Job:

```
job.setPartitionerClass(LoadBalancePartitioner.class);
```

Στη συνέχεια ο κάθε Reducer λαμβάνει τα δεδομένα που του ανατέθηκαν από τον Partitioner. Ο Reducer πραγματοποιεί το join των δεδομένων για κάθε σχέση και τοποθετεί σε μία λίστα προτεραιότητας τα δεδομένα, τα οποία αποτελούν τα Top-K Join αποτελέσματα σύμφωνα με το ερώτημα που τέθηκε.

ΚΕΦΑΛΑΙΟ 5: Πειραματικό μέρος

Ο παρακάτω πίνακας περιλαμβάνει τις τιμές και τις παραμέτρους οι οποίες χρησιμοποιήθηκαν για το πείραμα:

Παράμετροι	Τιμές
Κατανομή δεδομένων	Zipfian
Αριθμός Δεδομένων	1.000.000 /relation
Αριθμός Reducers	10,6
Αριθμός K results	10
Skew δεδομένων	0, 0.5, 1
Αριθμός Joining Attributes	10
Max τιμή για τα δεδομένα	10000
Sorting	By score
Histograms	10 bins

5.1 Περιβάλλον εκτέλεσης πειραμάτων

Τα πειράματα εκτελέστηκαν σε ένα cluster 8 μηχανημάτων με τα ακόλουθα χαρακτηριστικά:

Το κάθε ένα από τα 4 μηχανήματα διαθέτει:

Χαρακτηριστικά	Τιμή
Processor	Intel Xeon X5650 2.67GHz
Cores	6

Memory	128GB
Hard Disk	10TB

Το κάθε ένα από τα υπόλοιπα 4 μηχανήματα διαθέτει:

Processor	AMD Opteron(tm) Processor 4130
Cores	4
Memory	32GB
Hard Disk	10TB

5.2 Μορφή Δεδομένων

Τα δεδομένα στα οποία εκτελέστηκαν οι 3 αλγόριθμοι του πειράματος δημιουργήθηκαν με Zipfian κατανομή όπως αναφέρθηκε λεπτομερώς και στο προηγούμενο κεφάλαιο.

Οι γεννήτριες παραγωγής δεδομένων δημιουργούν δύο πίνακες με 1.000.000 εγγραφές στον καθένα (2.000.000 συνολικά) και αποτελούνται από 10 Joining attributes, πράγμα που σημαίνει το εξής:

Για παράδειγμα αν ο αριθμός των Joining Attributes = 10 τότε θα δημιουργηθούν:

J1, J2, J3 ….. J10

Αν ο αριθμός των Joining Attributes = 5 τότε θα δημιουργηθούν:

J1, J2, …. J5

Η τελική μορφή των δεδομένων είναι:

ID JoiningAttribute Score

1	J1	20
	…..	
10	J10	20

Επίσης στη δημιουργία των δεδομένων λαμβάνεται υπ'όψιν και το skew, δηλαδή το πόσο συμμετρική ή όχι είναι η κατανομή των εγγραφών του κάθε πίνακα για το κάθε Join Attribute. Επίσης πολύ σημαντικό είναι να σημειωθεί ότι τα δεδομένα είναι ταξινομημένα σύμφωνα με τη στήλη score.

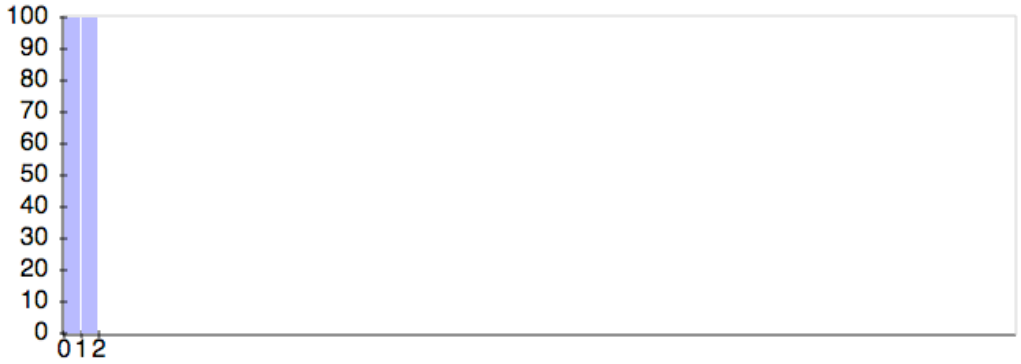
Παράλληλα με τη δημιουργία των δεδομένων, όπως έχει αναφερθεί και σε προηγούμενο κεφάλαιο, γίνεται και η παραγωγή ιστογραμμάτων έτσι ώστε να υπολογιστούν τα όρια για το Early Termination. Ο αριθμός των bins του ιστογράμματος που έχει επιλεχθεί είναι 10 σύμφωνα και με τον αριθμό των Joining Attributes. Πιο συγκεκριμένα η πληροφορία που περιλαμβάνουν τα ιστογράμματα είναι ο αριθμός των πλειάδων για κάθε join value μέσα σε κάθε εύρος τιμών.

5.3 Εκτέλεση Πειραμάτων

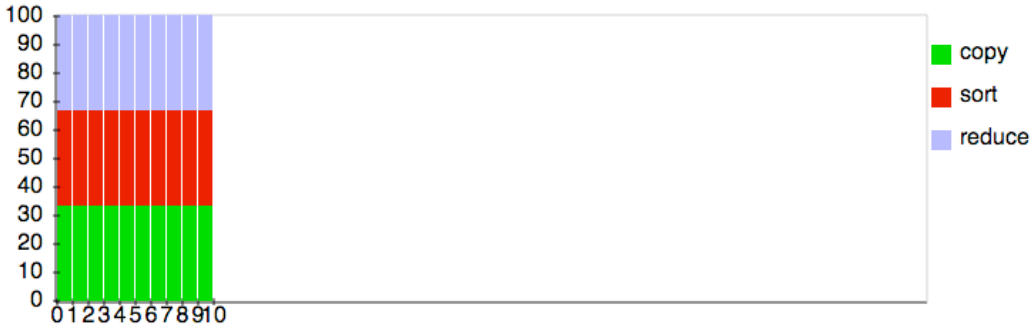
5.3.1 Εκτέλεση Naive Map/Reduce

Η εκτέλεση του προγράμματος Naive πραγματοποιήθηκε χωρίς καμία από τις λύσεις που προτείναμε παρά μόνο με τον έλεγχο για το αν τα scores ξεπερνούν τα όρια που έχουν υπολογιστεί και πρόκειται για ένα απλό Map/Reduce πρόγραμμα. Με αυτή την εκτέλεση θέλουμε να δείξουμε τη διαφορά τόσο στον χρόνο όσο και στον αριθμό των πλειάδων που θα διαβαστούν τόσο από την Map (όλες) όσο και από την Reduce (μόνο αυτές που δεν ξεπερνούν τα όρια).

Παρακάτω φαίνεται ο αριθμός των Mappers (3) και ο αριθμός των Reducers (10) που εκτελέστηκαν.



Εικόνα 8. Ο αριθμός των Mappers που εκτελέστηκαν για τον Naive με Skew = 0, Skew = 0.5, Skew = 1 και Reducers = 10



Εικόνα 9. Ο αριθμός των Reducers που εκτελέστηκαν για τον Naive με Skew = 0, Skew = 0.5, Skew = 1 και Reducers = 10

Για τιμή των scores skewed = 0 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Failed reduce tasks	0	0	1
	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	40
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	14,400
	Total time spent by all reduces in occupied slots (ms)	0	0	59,782
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 10. Χρόνοι εκτέλεσης Mappers και Reducers για τον Naïve με Skew = 0 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	2,000,000
	Map output records	0	0	2,000,000
	Map output bytes	0	0	62,667,436
	Input split bytes	0	0	278
	Combine input records	0	0	0
	Combine output records	0	0	0
	Spilled Records	0	0	4,000,000
	CPU time spent (ms)	0	0	12,040
	Physical memory (bytes) snapshot	0	0	1,211,322,368
	Virtual memory (bytes) snapshot	0	0	3,754,958,848
	Total committed heap usage (bytes)	0	0	1,999,241,216

Εικόνα 11. Naive αλγόριθμος Skew = 0 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι ίσος με όλες τις εγγραφές που υπήρχαν στα δεδομένα (2.000.000). Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάσουν στη φάση Reduce είναι 2.000.000. Ο χρόνος εκτέλεσης του προγράμματος είναι 42min 54sec.

Για τιμή των scores skewed = 0.5 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	20
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	15,586
	Total time spent by all reduces in occupied slots (ms)	0	0	16,226

Εικόνα 12. Χρόνοι εκτέλεσης Mappers και Reducers για τον Naïve με Skew = 0.5 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	2,000,000
	Map output records	0	0	2,000,000
	Map output bytes	0	0	62,774,514
	Input split bytes	0	0	280
	Combine input records	0	0	0
	Combine output records	0	0	0
	Spilled Records	0	0	4,000,000
	CPU time spent (ms)	0	0	15,720
	Physical memory (bytes) snapshot	0	0	1,279,148,032
	Virtual memory (bytes) snapshot	0	0	3,267,067,904
	Total committed heap usage (bytes)	0	0	1,680,932,864

Εικόνα 13. Naive αλγόριθμος με Skew = 0.5 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι ίσος με όλες τις εγγραφές που υπήρχαν στα δεδομένα (2.000.000). Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάσουν στη φάση Reduce είναι 2.000.000. Ο χρόνος εκτέλεσης του προγράμματος είναι 41min και 19sec.

Για τιμή των scores skewed = 1 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	20
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	17,257
	Total time spent by all reduces in occupied slots (ms)	0	0	16,400

Εικόνα 14. Χρόνοι εκτέλεσης Mappers και Reducers για τον Naïve με Skew = 1 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	2,000,000
	Map output records	0	0	2,000,000
	Map output bytes	0	0	62,774,514
	Input split bytes	0	0	278
	Combine input records	0	0	0
	Combine output records	0	0	0
	Spilled Records	0	0	4,000,000
	CPU time spent (ms)	0	0	17,330
	Physical memory (bytes) snapshot	0	0	1,341,820,928
	Virtual memory (bytes) snapshot	0	0	2,795,876,352
	Total committed heap usage (bytes)	0	0	1,362,493,440

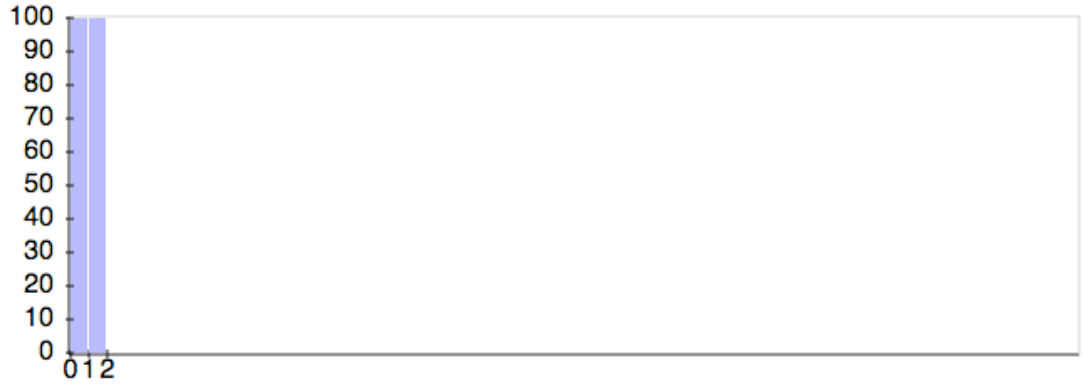
Εικόνα 15. Naive αλγόριθμος με Skew = 1 και Reducers = 10

Παρατηρώντας το πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι ίσος με όλες τις εγγραφές που υπήρχαν στα δεδομένα (2.000.000). Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάσουν στη φάση Reduce είναι 2.000.000. Ο χρόνος εκτέλεσης του προγράμματος είναι 43min και 50sec.

5.3.2 Εκτέλεση EarlyTermination Map/Reduce

Η εκτέλεση του προγράμματος ET πραγματοποιήθηκε με τη λύση του πρόωρου τερματισμού (EarlyTermination). Αυτό σημαίνει ότι από την Map και την Reduce δεν θα περάσουν όλες οι εγγραφές όπως συνέβη στο Naive Map/Reduce, παρά μόνον οι εγγραφές που το score τους δεν ξεπερνά τα όρια που υπολογίστηκαν από τα ιστογράμματα των δεδομένων.

Παρακάτω φαίνεται ο αριθμός των Mappers (3) και ο αριθμός των Reducers (10) που εκτελέστηκαν.



Εικόνα 16. Ο αριθμός των Mappers που εκτελέστηκαν για τον EarlyTermination με Skew = 0, Skew = 0.5, Skew = 1 και Reducers = 10

Εικόνα 17. Ο αριθμός των Reducers που εκτελέστηκαν για τον EarlyTermination με Skew = 0, Skew = 0.5, Skew = 1 και Reducers = 10

Για τιμή των scores skewed = 0 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	10
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	12,139
	Total time spent by all reduces in occupied slots (ms)	0	0	3,237,786
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 18. Χρόνοι εκτέλεσης Mappers και Reducers για τον EarlyTermination με Skew = 0 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	599,726
	Map output records	0	0	599,726
	Map output bytes	0	0	18,636,670
	Input split bytes	0	0	278
	Combine input records	0	0	0
	Combine output records	0	0	0
	Reduce input groups	0	0	10
	Reduce shuffle bytes	0	0	7,081,501
	Reduce input records	0	0	599,726
	Reduce output records	0	0	0
	Spilled Records	0	0	1,199,452
	CPU time spent (ms)	0	0	3,286,540
	Physical memory (bytes) snapshot	0	0	5,518,147,584
	Virtual memory (bytes) snapshot	0	0	21,771,546,624
	Total committed heap usage (bytes)	0	0	11,674,189,824

Εικόνα 19. EarlyTermination αλγόριθμο με Skew = 0 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι 599.726, μιας και αυτές ήταν οι εγγραφές οι οποίες δεν ξεπερνούσαν τα όρια που υπολογίστηκαν από τα ιστογράμματα. Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάνουν στη φάση Reduce είναι 599.726 όσες και στην Map. Ο χρόνος εκτέλεσης του προγράμματος είναι 7 mins και 22 secs.

Για τιμή των scores skewed = 0.5 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	10
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	10,490
	Total time spent by all reduces in occupied slots (ms)	0	0	1,028,412
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 20. Χρόνοι εκτέλεσης Mappers και Reducers για τον EarlyTermination με Skew = 0.5 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	330,018
	Map output records	0	0	330,018
	Map output bytes	0	0	10,262,803
	Input split bytes	0	0	280
	Combine input records	0	0	0
	Combine output records	0	0	0
	Reduce input groups	0	0	10
	Reduce shuffle bytes	0	0	4,090,197
	Reduce input records	0	0	330,018
	Reduce output records	0	0	0
	Spilled Records	0	0	660,036
	CPU time spent (ms)	0	0	1,002,220
	Physical memory (bytes) snapshot	0	0	5,432,545,280
	Virtual memory (bytes) snapshot	0	0	21,302,099,968
	Total committed heap usage (bytes)	0	0	11,231,232,000

Εικόνα 21. EarlyTermination αλγόριθμό με Skew = 0.5 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι 330.018, μιας και αυτές ήταν οι εγγραφές οι οποίες δεν ξεπερνούσαν τα όρια που υπολογίστηκαν από τα ιστογράμματα. Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα

φτάνουν στη φάση Reduce είναι 330.018 όσες και στην Map. Ο χρόνος εκτέλεσης του προγράμματος είναι 2 mins.

Για τιμή των scores skewed = 1 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	10
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	11,115
	Total time spent by all reduces in occupied slots (ms)	0	0	1,064,213
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 22 Χρόνοι εκτέλεσης Mappers και Reducers για τον EarlyTermination αλγόριθμο με Skew = 1 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	330,018
	Map output records	0	0	330,018
	Map output bytes	0	0	10,262,803
	Input split bytes	0	0	278
	Combine input records	0	0	0
	Combine output records	0	0	0
	Reduce input groups	0	0	10
	Reduce shuffle bytes	0	0	4,090,197
	Reduce input records	0	0	330,018
	Reduce output records	0	0	0
	Spilled Records	0	0	660,036
	CPU time spent (ms)	0	0	1,036,360
	Physical memory (bytes) snapshot	0	0	5,349,941,248
	Virtual memory (bytes) snapshot	0	0	20,339,269,632
	Total committed heap usage (bytes)	0	0	10,506,862,592

Εικόνα 23. EarlyTermination αλγόριθμος με Skew = 1 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι 330.018, μιας και αυτές ήταν οι εγγραφές οι οποίες δεν ξεπερνούσαν τα όρια που υπολογίστηκαν από τα ιστογράμματα. Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάνουν στη φάση Reduce είναι 330.018 όσες και στην Map. Ο χρόνος εκτέλεσης του προγράμματος είναι 2 mins 10 secs.

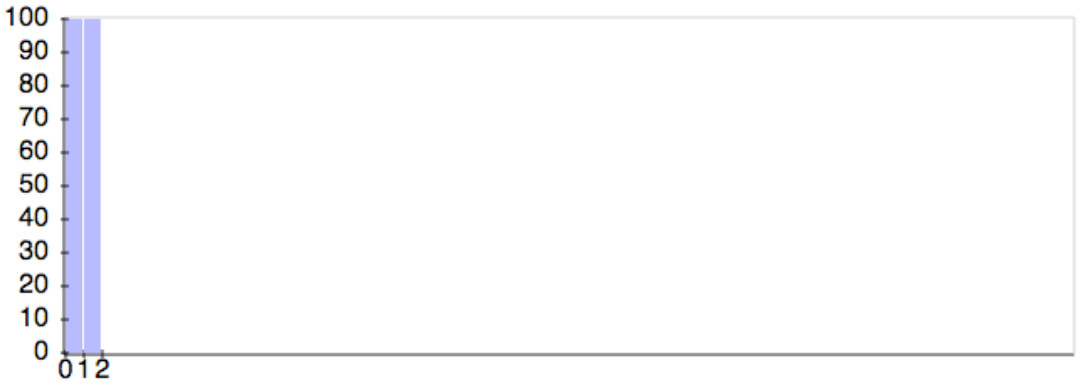
Το συμπέρασμα των παραπάνω είναι ότι για skewed = 0 ο χρόνος επιστροφής των αποτελεσμάτων για Top-K Join ερωτήματα είναι μεγαλύτερος σε σχέση με τις άλλες

περιπτώσεις skewed = 0.5 και skewed = 1. Παρόλα αυτά, ο συνολικός χρόνος είναι πάρα πολύ μικρότερος από το Naive μιας και ο αριθμός των εγγραφών που περνούν στην Map και στην Reduce για επεξεργασία είναι μόνο 330.018 από τις 2.000.000 εγγραφές λόγω του Early Termination.

5.3.3 Εκτέλεση EarlyTermination & Load Balancing Map/Reduce

Η εκτέλεση του προγράμματος ETLPT πραγματοποιήθηκε με τη λύση του πρόωρου τερματισμού (EarlyTermination) και τη βελτιστοποίηση εξισορρόπησης φορτίου ανά Reducer. Αυτό σημαίνει ότι από την Map και την Reduce δεν θα περάσουν όλες οι εγγραφές όπως συνέβη και στο ET Map/Reduce. Θα περάσουν μόνο οι εγγραφές που τα scores τους δεν ξεπερνά τα όρια που υπολογίστηκαν από τα ιστογράμματα των δεδομένων. Επιπλέον το φορτίο των δεδομένων θα μοιραστεί ισόποσα μιας και παρεμβάλλεται μετά την Map και την Reduce ο Partitioner.

Παρακάτω φαίνεται ο αριθμός των Mappers (3) και ο αριθμός των Reducers (10) που εκτελέστηκαν.



Εικόνα 24. Ο αριθμός των Mappers που εκτελέστηκαν για τον EarlyTermination και LoadBalancing με Skew = 0, Skew = 0.5, Skew = 1 και Reducers = 10

Εικόνα 25. Ο αριθμός των Reducers που εκτελέστηκαν για τον EarlyTermination και LoadBalancing με Skew = 0, Skew = 0.5, Skew = 1 και Reducers = 10

Για τιμή των scores skewed = 0 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	10
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	15,598
	Total time spent by all reduces in occupied slots (ms)	0	0	3,350,178
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 26. Χρόνοι εκτέλεσης Mappers και Reducers για τον EarlyTermination και LoadBalancing αλγόριθμο με Skew = 0 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	599,726
	Map output records	0	0	599,726
	Map output bytes	0	0	18,636,670
	Input split bytes	0	0	278
	Combine input records	0	0	0
	Combine output records	0	0	0
	Reduce input groups	0	0	10
	Reduce shuffle bytes	0	0	7,129,161
	Reduce input records	0	0	599,726
	Reduce output records	0	0	0
	Spilled Records	0	0	1,199,452
	CPU time spent (ms)	0	0	3,406,350
	Physical memory (bytes) snapshot	0	0	5,647,261,696
	Virtual memory (bytes) snapshot	0	0	21,428,908,032
	Total committed heap usage (bytes)	0	0	11,332,747,264

Εικόνα 27. EarlyTermination και LoadBalancing αλγόριθμος με Skew = 0 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι 599.726, μιας και αυτές ήταν οι εγγραφές οι

οποίες δεν ξεπερνούσαν τα όρια που υπολογίστηκαν από τα ιστογράμματα. Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάνουν στη φάση Reduce είναι 599.726 όσες και στην Map. Ο χρόνος εκτέλεσης του προγράμματος είναι 7 mins 6 secs.

Για τιμή των scores skewed = 0.5 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	10
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	13,541
	Total time spent by all reduces in occupied slots (ms)	0	0	1,071,872
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 28 Χρόνοι εκτέλεσης Mappers και Reducers για τον EarlyTermination και LoadBalancing με Skew = 0.5 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	330,018
	Map output records	0	0	330,018
	Map output bytes	0	0	10,262,803
	Input split bytes	0	0	280
	Combine input records	0	0	0
	Combine output records	0	0	0
	Reduce input groups	0	0	10
	Reduce shuffle bytes	0	0	4,090,631
	Reduce input records	0	0	330,018
	Reduce output records	0	0	0
	Spilled Records	0	0	660,036
	CPU time spent (ms)	0	0	1,048,480
	Physical memory (bytes) snapshot	0	0	5,615,452,160
	Virtual memory (bytes) snapshot	0	0	20,340,723,712
	Total committed heap usage (bytes)	0	0	10,220,601,344

Εικόνα 29. EarlyTermination και Load Balancing αλγόριθμος με Skew = 0.5 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι 330.018, μιας και αυτές ήταν οι εγγραφές οι οποίες δεν ξεπερνούσαν τα όρια που υπολογίστηκαν από τα ιστογράμματα. Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάνουν στη φάση Reduce είναι 330.018 όσες και στην Map. Ο χρόνος εκτέλεσης του προγράμματος είναι 2 mins 15 secs.

Για τιμή των scores skewed = 1 και Reducers = 10 φαίνονται τα παρακάτω:

Job Counters	Launched map tasks	0	0	2
	Launched reduce tasks	0	0	10
	Data-local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	13,798
	Total time spent by all reduces in occupied slots (ms)	0	0	1,039,895
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0

Εικόνα 30. Χρόνοι εκτέλεσης Mappers και Reducers για τον EarlyTermination και Load Balancing αλγόριθμο με Skew = 1 και Reducers = 10

Map-Reduce Framework	Map input records	0	0	330,018
	Map output records	0	0	330,018
	Map output bytes	0	0	10,262,803
	Input split bytes	0	0	278
	Combine input records	0	0	0
	Combine output records	0	0	0
	Reduce input groups	0	0	10
	Reduce shuffle bytes	0	0	4,090,631
	Reduce input records	0	0	330,018
	Reduce output records	0	0	0
	Spilled Records	0	0	660,036
	CPU time spent (ms)	0	0	1,017,420
	Physical memory (bytes) snapshot	0	0	5,575,630,848
	Virtual memory (bytes) snapshot	0	0	20,937,859,072
	Total committed heap usage (bytes)	0	0	10,776,215,552

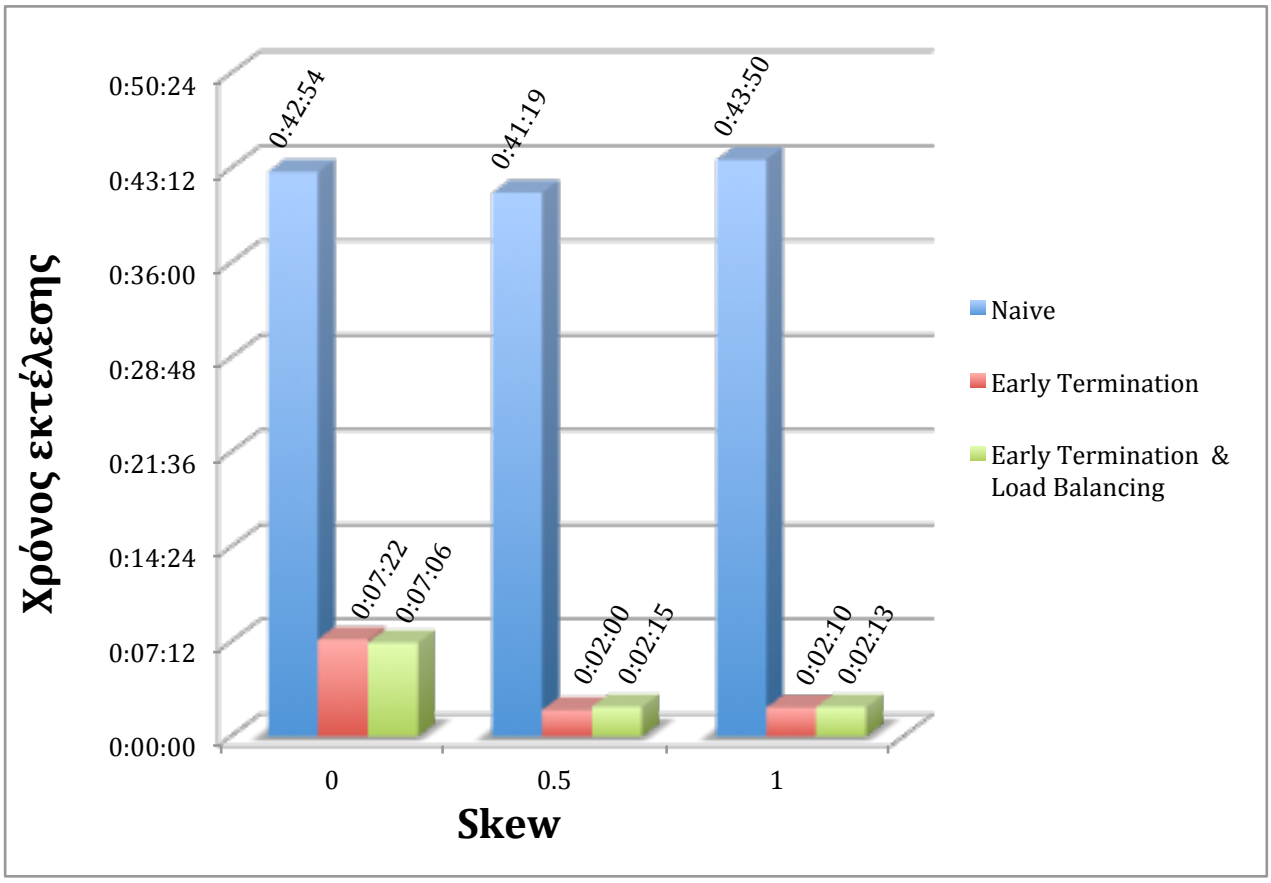
Εικόνα 31. EarlyTermination και Load Balancing αλγόριθμος με Skew = 1 και Reducers = 10

Παρατηρώντας τον πίνακα είναι ξεκάθαρο ότι ο αριθμός των πλειάδων που διαβάστηκαν από τη φάση Map είναι 330.018, μιας και αυτές ήταν οι εγγραφές οι οποίες δεν ξεπερνούσαν τα όρια που υπολογίστηκαν από τα ιστογράμματα. Στη συνέχεια παρατηρείται, όπως προαναφέραμε, ότι οι πλειάδες οι οποίες τελικά θα φτάνουν στη φάση Reduce είναι 330.018 όσες και στην Map. Ο χρόνος εκτέλεσης του προγράμματος είναι 2 mins 13 secs.

Το συμπέρασμα των παραπάνω είναι ότι για skewed = 0 ο χρόνος επιστροφής των αποτελεσμάτων για Top-K Join ερωτήματα είναι μεγαλύτερος σε σχέση με τις άλλες περιπτώσεις skewed = 0.5 και skewed = 1. Παρόλα αυτά, ο συνολικός χρόνος και σε αυτή την περίπτωση είναι πάρα πολύ μικρότερος από το Naive μιας και ο αριθμός των εγγραφών που περνούν στην Map και στην Reduce για επεξεργασία είναι μόνο

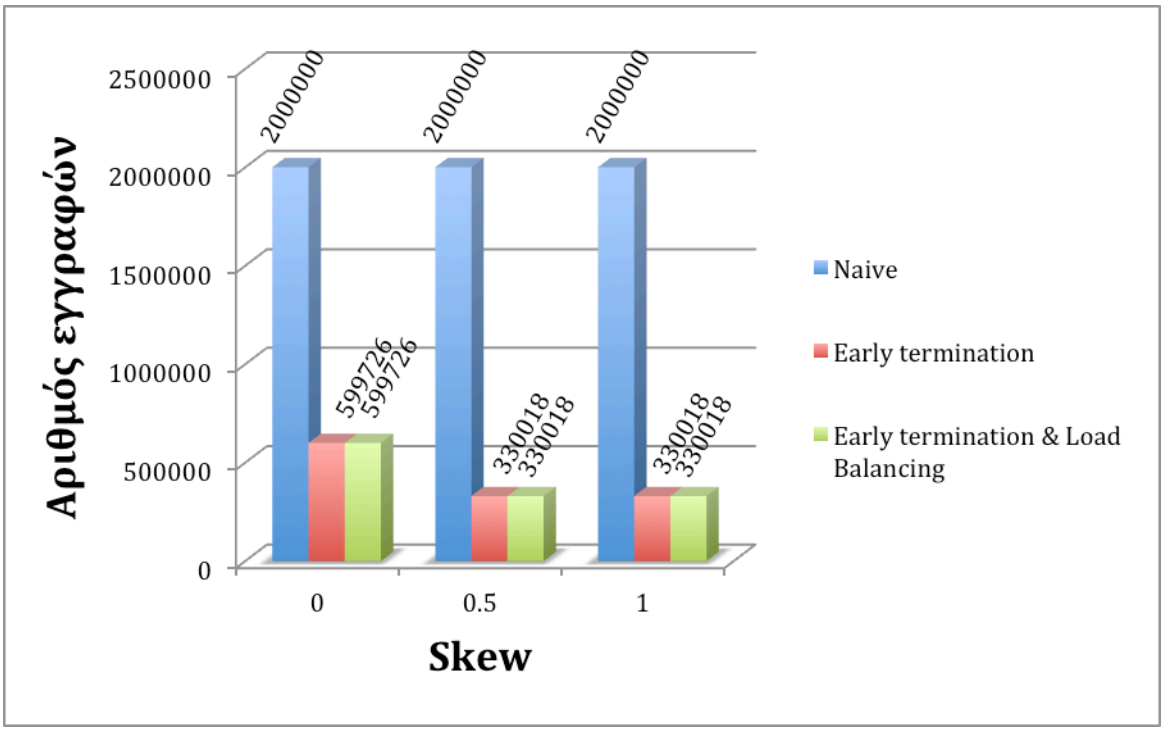
330.018 από τις 2.000.000 εγγραφές λόγω του Early Termination. Σε σύγκριση με τον Early Termination, ο χρόνος εκτέλεσης είναι σχεδόν ο ίδιος, μιας και όπως αναφέρθηκε και στην μορφή των δεδομένων, ο αριθμός των Reducers είναι ίσος με τον αριθμό των Joining χαρακτηριστικών και κατά κάποιο τρόπο τα δεδομένα κατανέμονται ισόποσα και στις δύο περιπτώσεις.

Παρακάτω απεικονίζεται διαγραμματικά ο χρόνος εκτέλεσης του κάθε αλγορίθμου ξεχωριστά για skew = 0, skew = 0.5 και skew = 1:



Εικόνα 32 Χρόνος εκτέλεσης κάθε αλγορίθμου για skew=0, skew=0.5, skew=1

Παρακάτω απεικονίζεται διαγραμματικά ο αριθμός εγγραφών που κατέληξαν στη φάση Reduce (αριθμός εγγραφών Map = Reduce):



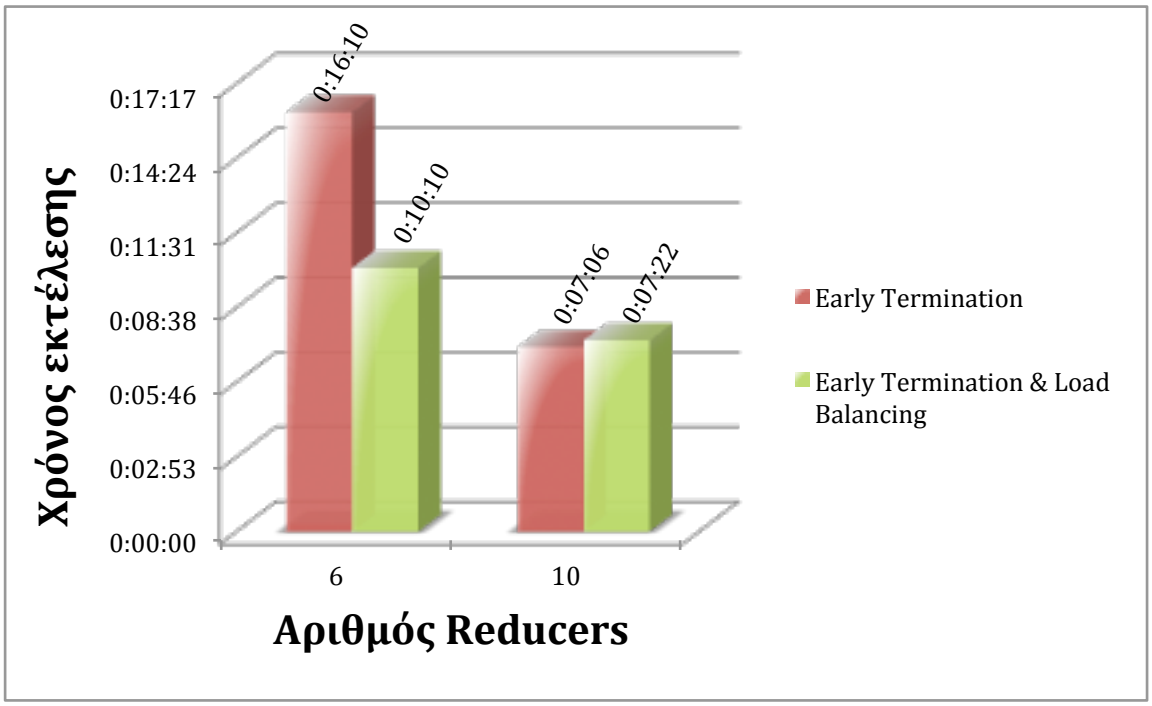
Εικόνα 33. Ο αριθμός των εγγραφών που κατέληξαν στη φάση Reduce

Αν εκτελέσουμε τα δυο τελευταία πειράματα με έναν αριθμό Reducers μικρότερο από τον αριθμό των Joining χαρακτηριστικών, για παράδειγμα Reducers = 6, θα δούμε ότι το EarlyTermination με LoadBalancing είναι σαφώς αποδοτικότερο από το απλό EarlyTermination μιας και η κατανομή των εγγραφών στους Reducers θα γίνει με τον βέλτιστο τρόπο σύμφωνα με τον LPT αλγόριθμο κατά 6 λεπτά γρηγορότερα.

Συγκεκριμένα εκτελέσαμε και τα δύο πειράματα με Reducers = 6 και οι χρόνοι εκτέλεσης ήταν οι εξής:

Map/Reduce:	No of Reducers:	Data Skewed:	Time:
Simple EarlyTemrination	6	0	16mins, 10secs
	10	0	7mins, 6secs
EarlyTermination & Load Balancing	6	0	10mins, 10secs
	10	0	7mins, 22secs

Παρακάτω απεικονίζεται διαγραμματικά η σύγκριση από πλευράς χρόνου απόκρισης των αλγορίθμων Early Termination και Early Termination & Load Balancing:



Εικόνα 34. Σύγκριση αλγορίθμων Early Termination με Early Termination & Load Balancing για Reducers = 6 και Skew = 0

ΚΕΦΑΛΑΙΟ 6: Γενικά Συμπεράσματα

6.1 Συμπεράσματα εργασίας

Το συμπέρασμα που προκύπτει από τα παραπάνω πειράματα είναι ότι η εκτέλεση ενός Naive προγράμματος Map/Reduce είναι μη αποδεκτή και καθόλου αποδοτική σε Top-K Join (ranked γενικότερα) ερωτήματα, διότι το κλασικό Map/Reduce διαβάζει όλα τα δεδομένα για να ανακτήσει το τελικό αποτέλεσμα. Συγκρίνοντας τα δύο επόμενα πειράματα που περιέχουν την υλοποίηση EarlyTermination, κρίνοντας από τους χρόνους εκτέλεσης, συμπεραίνουμε ότι είναι σχεδόν το ίδιο αποδοτικά, παρότι το δεύτερο πείραμα περιέχει και τη βελτιστοποίηση LoadBalancing .

Αυτό συμβαίνει γιατί ο αριθμός των Joining χαρακτηριστικών των δεδομένων είναι ίσος με τον αριθμό των Reducers που επεξεργάζονται τη διαδικασία Join των δεδομένων.

Στη συνέχεια με τη μείωση του αριθμού των Reducers στους αλγορίθμους EarlyTermination και EarlyTermination & Load Balancing, φαίνεται ξεκάθαρα από πλευράς χρόνων εκτέλεσης ότι με ο δεύτερος αλγόριθμός είναι αποδοτικότερος κατά έξι λεπτά. Αυτό οφείλεται στο γεγονός ότι η κατανομή των δεδομένων με την εφαρμογή του τροποποιημένου Partitioner, γίνεται με δίκαιο και ισομερή τρόπο. Επομένως ο κάθε Reducer αναλαμβάνει σχεδόν τον ίδιο φόρτο εργασίας για κάθε κομμάτι δεδομένων που του αναλογεί για επεξεργασία.

Σε αυτή την εργασία περιγράφθηκαν τεχνικές για την αποδοτική εκτέλεση Top-K ερωτημάτων χρησιμοποιώντας το Map/Reduce καθώς επίσης προτάθηκαν λύσεις σε συγκεκριμένα προβλήματα που εμφανίζονται στο Map/Reduce.

6.2 Μελλοντική εργασία

Επόμενα βήματα και εξέλιξη της παραπάνω εργασίας είναι η αυτοματοποίηση κάποιων διαδικασιών από την προκαθορισμένη λειτουργία του Hadoop. Στην εργασία παρουσιάστηκε η ταξινόμηση των δεδομένων με βάση το score σαν μία διαδικασία η οποία εκτελείται έξω από το περιβάλλον του Hadoop, όπως και η

παραγωγή ιστογραμμάτων για τον υπολογισμό των ορίων έτσι ώστε να επιτευχθεί το EarlyTermination.

Δεδομένων των παραπάνω, προτεινόμενη εργασία για το μέλλον είναι η αυτοματοποίηση των διαδικασιών από το ίδιο το Hadoop έτσι ώστε να επιτευχθεί ο βέλτιστος χρόνος απόκρισης, η εκμετάλλευση της ταχύτητας του Hadoop για επεξεγασία δεδομένων καθώς και του Map/Reduce σε ερωτήματα κατάταξης.

Όπως αναφέρθηκε στην εργασία, το Hadoop κατακερματίζει τα δεδομένα σε blocks των 64MB για να τα φορτώσει στο σύστημα αρχείων του Hadoop (HDFS). Θα ήταν δυνατή λοιπόν η ταξινόμηση του κάθε block δεδομένων ξεχωριστά, βασισμένη σε μία O(nlogn) συνάρτηση, έτσι ώστε τα blocks να φορτώνονται στο HDFS ήδη ταξινομημένα και παράλληλα να υπολογίζονται τα ιστογράμματα και τα όρια κατά τη φόρτωση των αρχείων στο HDFS. Έχοντας τα όρια από τα ιστογράμματα καθώς και τα ταξινομήμενα blocks είναι πάρα πολύ εύκολο να "πετάμε" blocks που είναι απρόσφορα για εξαγωγή αποτελεσμάτων. Ο συνδυασμός των παραπάνω με το EarlyTermination & Load Balancing ίσως επιτύχουν εξαιρετικά αποδοτικούς χρόνους από πλευράς Top-K και Top-K Join ερωτημάτων στο Hadoop.

Βιβλιογραφία

[1] Christos Doulkeridis, Akrivi Vlachou, Kjetil Nørvag, Yannis Kotidis and Neoklis Polyzotis, *Processing of Rank Joins in Highly Distributed Systems*, ICDE, 2012, Washington DC

[2] Christos Doulkeridis, Kjetil Nørv̊ag, *On Saying "Enough Already!" in MapReduce,* International Conference on Data Engineering, Cloud Intelligence, 2012, Istanbul Turkey,

[3] Christos Doulkeridis, Kjetil Nørv̊ag, *A Survey of Large-Scale Analytical Query Processing in MapReduce*, VLDB journal, 2013, Riva Del Grada, Trento

[4] K. Selçuk Candan, Parth Nagarkar, Mithila Nagendra, Renwei Yu, *RanKloud: A Scalable Ranked Query Processing Framework on Hadoop,* VLDB, 2013, Riva Del Grada, Trento

[5] IHAB F. ILYAS, GEORGE BESKALES, MOHAMED A. SOLIMAN, *A Survey of Top-k Query Processing Techniques in Relational Database Systems*, Journal ACM Computing Surveys (CSUR), 2008

[6] Chuck Lam, *Hadoop In Action,* 1st Edition, Manning Publications Co., 2011

[7] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, *The Google File System*, 19th ACM Symposium on Operating Systems Principles, 2003, Bolton Landing, NY, USA

[8] Borthakur, Dhruba. *The Hadoop Distributed File System: Architecture and Design,* 2007, The Apache Software Foundation.

[9] Ronald Fagin, Amnon Lotem, Moni Naor, *Optimal Aggregation Algorithms for Middleware*, 2001, PODS, Santa Barbara, California, USA

[10] Yahoo, *Yahoo! Hadoop Tutorial*, Available from: http://developer.yahoo.com/hadoop/tutorial/

[11] Tom White, Hadoop, 3d Edition, O'Reily, 2012

ΠΑΡΑΡΤΗΜΑ Α

Hadoop on Ubuntu in Pseudo-Distributed Mode

1. Πρσθήκη ενός λογαριασμού για το Hadoop:

```
sudo addgroup hadoop

ssh adduser --ingroup hadoop hadoop
```

2. Δημιουργία ενός SSH λογαριασμού για τον Hadoop χρήστη:

```
sudo apt-get install ssh

su - hadoop

ssh-keygen -t rsa -P ""

cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

3. Παίρνουμε από το site του Hadoop την έκδοση 0.20.2

4. Παραμετροποίηση του Hadoop

Ανοίγουμε το αρχείο hadoop-env.sh και κάνουμε export το Path της Java

```
export JAVA_HOME=/user/lib/jvm/java-6-openjdk
```

Στην συνέχεια πρέπει να επεξεργαστούμε 3 αρχεία στο Hadoop.

- core-site.xml
- hdfs-site.xml
- mapred-site.xml

conf/core-site.xml

```
<property>
 <name>hadoop.tmp.dir</name>
 <value>/app/hadoop/tmp</value>
 <description>A base for other temporary directories.</description>
</property>
<property>
 <name>fs.default.name</name>
 <value>hdfs://localhost:54310</value>
 <description>The name of the default file system.  A URI whose
 scheme and authority determine the FileSystem implementation.  The
 uri's scheme determines the config property (fs.SCHEME.impl) naming
 the FileSystem implementation class.  The uri's authority is used to
 determine the host, port, etc. for a filesystem.</description>
</property>
```

conf/mapred-site.xml

```
<property>
 <name>mapred.job.tracker</name>
 <value>localhost:54311</value>
 <description>The host and port that the MapReduce job tracker runs
 at.  If "local", then jobs are run in-process as a single map
 and reduce task.
 </description>
</property>
```

conf/hdfs-site.xml

```
<property>
 <name>dfs.replication</name>
 <value>1</value>
 <description>Default block replication.
 The actual number of replications can be specified when the file is created.
 The default is used if replication is not specified in create time.
 </description>
</property>
```

5. Πραγματοποιούμε format του HDFS:

```
bin/hadoop namenode -format
```

6. Εκκίνηση του cluster:

```
bin/start-dfs.sh && bin/start-mapred.sh
```

Εκτέλεση Πειραματικού μέρους

1. Παραγωγή δεδομένων με το jar GeneratorsRJ.jar

```
java –jar GeneratorsRJ.jar <outputpath> <numOfTables> <skew> <sizeOfData>
```

Το παραπάνω θα πάραγει 4 αρχεία αν επιλέξουμε για παράδειγμα 2 πίνακες: R0sorted.txt, R1sorted.txt, hist0.txt, hist1.txt. Όπου R0,R1 είναι οι πίνακες με τα δεδομένα και hist0, hist1 τα ιστογράμματα για τον υπολογισμό των ορίων που περιγράφηκαν στην εργασία.

2. Φόρτωση των δεδομένων στο HDFS:

```
hadoop dfs -copyFromLocal path-to-data-dir/ path-to-data-in-hdfs/

hadoop dfs -copyFromLocal path-to-hist-dir/ path-to-hist-in-hdfs/
```

3. Εκτέλεση των τριών αλγορίθμων με τα ακόλουθα jars που είναι στην διάθεση σας:

EarlyTermination & Load Balancing

```
hadoop jar thesis-etlpt.jar JoinETLPT /path/to/data/in/hdfs/
/path/to/results/in/hdfs/ path/to/hists/in/hdfs/ <numOfTopKResults>
<numOfReducers>
```

EarlyTermination & Load Balancing

```
hadoop jar thesis-et.jar JoinET /path/to/data/in/hdfs/
/path/to/results/in/hdfs/ path/to/hists/in/hdfs/ <numOfTopKResults>
<numOfReducers>
```

Naive

```
hadoop jar thesis-naive.jar JoinNaive /path/to/data/in/hdfs/
/path/to/results/in/hdfs/ path/to/hists/in/hdfs/ <numOfTopKResults>
<numOfReducers>
```

ΠΑΡΑΡΤΗΜΑ Β

Job Setup για εκτέλεση

```
public class JoinETLPT extends Configured implements Tool{

	public static HashMap<Integer, Integer> dataToReducers;
	public static Integer numOfReducers = 0;
	public final static List<Integer> logs = new
ArrayList<Integer>();
	public static Integer countMapTuples = 0,
countReduceTuples = 0;
	private static Logger LOGGER;
	private static Date startTime;
	private static Date endTime;
	private Configuration conf;

	private static void printUsage() {
		System.out.println("Usage: <inputPath> <outputPath>
<histPath> [<k_tuples = 50>]");
	}

	private static int[] computeBounds(Configuration conf,
String histPath) {
		int[] arBounds = new int[2];
		BufferedReader br0 = null;
		BufferedReader br1 = null;
		try {
			FileSystem hdfs = FileSystem.get(conf);
			Path path0 = new
Path(histPath+File.separatorChar+"hist0.txt");
			Path path1 = new
Path(histPath+File.separatorChar+"hist1.txt");
			br0 = new BufferedReader(new
InputStreamReader(hdfs.open(path0)));
			br1 = new BufferedReader(new
InputStreamReader(hdfs.open(path1)));
		} catch (IOException e) {
			e.printStackTrace();
		}

		BoundEstimatorHist b = new BoundEstimatorHist();
		b.initialize(10, 10, 10000, br0, br1);
		double[] w = new double[2];
		w[0] = 0.5;
		w[1] = 0.5;
```

```
            int nNrDistValues = 10;// as many as the #join
attributes for query type Q1
            int[] arHistId0 = new int[nNrDistValues];
            int[] arHistId1 = new int[nNrDistValues];
            for (int j=0 ; j < nNrDistValues ; j++) {
                  arHistId0[j] = j;
                  arHistId1[j] = j + nNrDistValues;
            }
            double[] arDepths = b.estimate(10, w, arHistId0,
arHistId1);
            arBounds[0] = (int)arDepths[0];
            arBounds[1] = (int)arDepths[1];

            // Assign datasources according to data
            if (b != null) {
                  LPT lpt = new LPT(numOfReducers);
                  dataToReducers =
lpt.assignDataToReducerHasMap(b.joinResultsPerJoinValue);
            }
            return arBounds;
      }

      public static void main(String[] args) throws
Exception {
         int ret = ToolRunner.run(new JoinETLPT(), args);
         System.exit(ret);
      }

      @SuppressWarnings("deprecation")
      @Override
      public int run(String[] args) throws Exception {
            conf= getConf();
            String[] otherArgs=new GenericOptionsParser(conf,
args).getRemainingArgs();
            String inputPath = otherArgs[0];
            String outputPath = otherArgs[1];
            String histPath = otherArgs[2];
            if (args.length > 4) {
                  numOfReducers = Integer.valueOf(otherArgs[4]);
            }

            int[] arBounds = computeBounds(conf, histPath);
            conf.setInt("bound1", arBounds[0]);
            conf.setInt("bound2", arBounds[1]);
            JSONObject jsonDataReducers = new JSONObject();

      jsonDataReducers.put("dataToReducers",dataToReducers);
            conf.set("dataToReducers",
```

```
jsonDataReducers.toString());

          if (args.length > 3) {
               try {
                    int kValue = Integer.parseInt(args[3]);
                    if (kValue <= 0) {
                         throw new Exception();
                    }
                    conf.setInt("kValue", kValue);
               } catch (Exception ex) {
                    System.out.println("Could not
successfully parse <k_tuples> info: Expecting positive integer
greater than zero.");
                    System.exit(1);
               }
          }

          Job job = new Job(conf, "joinETLPT");
          job.setJarByClass(JoinETLPT.class);
          job.setPartitionerClass(MyPartitioner.class);
          job.setMapperClass(Map.class);
          job.setReducerClass(Reduce.class);
          job.setOutputKeyClass(IntWritable.class);
          job.setOutputValueClass(MyKeyValuePair.class);

     job.setInputFormatClass(EarlyTermInputFormat.class);
          job.setOutputFormatClass(TextOutputFormat.class);
          if (args.length > 4) {
               job.setNumReduceTasks(numOfReducers);
          }


          FileInputFormat.setInputPaths(job, new
Path(inputPath));//new Path(args[0]));
          FileOutputFormat.setOutputPath(job, new
Path(outputPath));//new Path(args[1]));
          //JobClient.runJob(conf);

          startTime=new Date();
          System.out.println("Job started at : "+ startTime);
          if (job.waitForCompletion(true)) {
               endTime = new Date();
            LOGGER = new Logger(Constants.logs);
            LOGGER.write("------Simulation time for ETLPT----
--",
                    (endTime.getTime() - startTime.getTime())
/1000+ " seconds.", null, null);
            LOGGER.write("------Metrics------",
```

```
                        " Map read: "+Map.countMapTuples+"
tuples", null, null);
                LOGGER.write("",
                        " Reduce got input:
"+Reduce.countReduceTuples+" tuples", null, null);
                System.out.println("Job ended: " + endTime);
                System.out.println("The job took " +
(endTime.getTime() - startTime.getTime()) /1000 + " seconds.");
                return 1;
            }
            return 0;
        }
}
```

Customized EarlyInputFormat για την υλοποίηση EarlyTermination

```
public class EarlyTermInputFormat extends TextInputFormat {
    @Override
    public RecordReader<LongWritable, Text>
createRecordReader(InputSplit split, TaskAttemptContext
context) {
        return new EarlyTermRecordReader();
    }
    @Override
    protected boolean isSplitable(JobContext context, Path
filename) {
        return true;
    }
}
```

Customized EarlyTermRecordReader

```
public class EarlyTermRecordReader extends
RecordReader<LongWritable, Text>{
    private LineReader in;
    private LongWritable key;
    private Text value = new Text();
    private long start =0;
    private long end =0;
    private long pos =0;

    private int bound0;
    private int bound1;
    private int tag;
```

```
	private Text buffer = new Text();

	@Override
	public void close() throws IOException {
		if (in != null) {
			in.close();
		}
	}

	@Override
	public LongWritable getCurrentKey() throws
IOException,InterruptedException {
		return key;
	}

	@Override
	public Text getCurrentValue() throws IOException,
InterruptedException {
		return value;
	}

	@Override
	public float getProgress() throws IOException,
InterruptedException {
		if (start == end) {
			return 0.0f;
		}
		else {
			return Math.min(1.0f, (pos - start) /
(float)(end - start));
		}
	}

	@Override
	public void initialize(InputSplit genericSplit,
TaskAttemptContext context)throws IOException,
InterruptedException {
		FileSplit split = (FileSplit) genericSplit;
		final Path file = split.getPath();
		String filename = file.getName();
		if (filename.startsWith("R0")) {
			tag = 0;
		}
		else {
			tag = 1;
		}
		Configuration conf = context.getConfiguration();
		this.bound0 =
```

```
conf.getInt("bound1",Integer.MAX_VALUE);
            this.bound1 =
conf.getInt("bound2",Integer.MAX_VALUE);
            FileSystem fs = file.getFileSystem(conf);
            start = split.getStart();
            end= start + split.getLength();
            FSDataInputStream filein = fs.open(split.getPath());
            in = new LineReader(filein,conf);
            this.pos = start;
      }

      @Override
      public boolean nextKeyValue() throws IOException,
InterruptedException {
            int newSize = in.readLine(buffer);

            String line = buffer.toString();
            String[] split = line.split(" ");

            if (!split[0].equals("") && !split[1].equals("") &&
!split[2].equals("")) {
                  key   = new
LongWritable(Long.parseLong(split[0]));
                  value      = new Text(key+" "+split[1]+"
"+split[2]);
                  int scoreAttr = Integer.parseInt(split[2]);

                  int bound = ((tag == 0) ? bound0 : bound1);

                  if (scoreAttr > bound)
                        return false;
                  else
                        return true;
            }
            return false;
      }
}
```

Mapper

```
public class Map extends Mapper<LongWritable, Text,
IntWritable, MyKeyValuePair> {
     private final static MyKeyValuePair outval = new
MyKeyValuePair();
     private final static IntWritable outkey = new
IntWritable(0);
     private int bound1, bound2;
     public static Integer countMapTuples = 0;

     public void setup(Context context) {
          bound1 = context.getConfiguration().getInt("bound1",
0);
          bound2 = context.getConfiguration().getInt("bound2",
0);
     }

     public void map(LongWritable key, Text value, Context
context) throws IOException, InterruptedException {
          FileSplit fileSplit =
(FileSplit)context.getInputSplit();
          String filename = fileSplit.getPath().getName();
          int tag;

          if (filename.startsWith("R0")) {
               tag = 0;
          }
          else {
               tag = 1;
          }

          String line = value.toString();
          String[] split = line.split(" ");

          String join = split[1];

          Integer scoreAttr = Integer.parseInt(split[2]);
          if (((tag == 0) && (scoreAttr <= bound1)) ||
                    ((tag == 1) && (scoreAttr <= bound2))) {
               int nJAttrId =
Integer.parseInt(join.substring(1));
               outkey.set(nJAttrId);
               outval.setKey(tag);
               outval.setValue(line);
               outval.setTupleScore(scoreAttr);
               outval.setJoinAttr(nJAttrId);
               context.write(outkey, outval);
```

```
            }
      }
}
```

Reducer

```
public class Reduce extends Reducer<IntWritable,
MyKeyValuePair, IntWritable, MyKeyValuePair> {
      private int kValue = 0;
      private Logger logger = Logger.getLogger(Reduce.class);
      private int partition = 0;
      public static Integer countReduceTuples = 0;;

      public void setup(Context context) {
            kValue = context.getConfiguration().getInt("kValue",
50);
            context.setStatus("FINISH WITH REDUCE AND GO ON");
            context.progress();
      }

      private int applyFunction(int score0, int score1) {
            return score0 + score1;
      }

      public void reduce(IntWritable key,
Iterable<MyKeyValuePair> values, Context context) throws
IOException, InterruptedException
      {
            List<MyKeyValuePair> arValues = new
ArrayList<MyKeyValuePair>();
            ScoredJoinedTuple joinedTuple = null;
            int joinedScore = 0;

            for (Iterator<MyKeyValuePair> it0 =
values.iterator(); it0.hasNext();){
                  MyKeyValuePair tuple1 = it0.next();
                  MyKeyValuePair tupleNew = new
MyKeyValuePair(tuple1.getKey(), tuple1.getValue(),
tuple1.getJoinAttr(), tuple1.getTupleScore());
                  arValues.add(tupleNew);
                  tupleNew = null;
                  tuple1 = null;
            }
            values = null;
            StringBuilder sb = new StringBuilder();
```

```
            for (int i = 0; i < arValues.size(); i++) {
                  MyKeyValuePair tuple1 = arValues.get(i);
                  for (int j = i+1; j < arValues.size(); j++) {
                        MyKeyValuePair tuple2 = arValues.get(j);
                        if(tuple1.getKey() != tuple2.getKey() &&
tuple1.getJoinAttr() == tuple2.getJoinAttr()) {

      sb.append(tuple1).append("|").append(tuple2);
                              // Tuples DO join
                              joinedScore =
applyFunction(tuple1.getTupleScore(), tuple2.getTupleScore());
                              joinedTuple = new
ScoredJoinedTuple(joinedScore, sb.toString());
                              sb.delete(0, sb.length());
                              joinedTuple = null;
                        }
                        tuple2 = null;
                  }
                  arValues.remove(i);
                  tuple1 = null;
            }
            arValues = null;
            sb = null;
      }

}
```

Partitioner

```
public class MyPartitioner extends Partitioner<IntWritable,
MyKeyValuePair> implements Configurable{

      private Configuration configuration;
      private List<Reducer> dataToReducers;
      private JSONObject dataToReducersHashJSON = null;

      @Override
      public int getPartition(IntWritable key, MyKeyValuePair
value, int numPartitions){
            String jsonDataToReducers =
configuration.get("dataToReducers");
            try {
                  JSONObject jsonObject = new
JSONObject(jsonDataToReducers.toString());
                  dataToReducersHashJSON =
```

```
jsonObject.getJSONObject("dataToReducers");
            } catch (JSONException e) {
                  e.printStackTrace();
            }
            int relation = key.get();
            try {
                  return getReducerToSendFromJSON(relation);
            } catch (JSONException e) {
                  e.printStackTrace();
                  return 0;
            }
      }

      @Override
      public void setConf(Configuration configuration) {
            this.configuration = configuration;
      }

      @Override
      public Configuration getConf() {
            return configuration;
      }
      public Integer getReducerToSendFromJSON(Integer
relation) throws JSONException
      {
            if (dataToReducersHashJSON != null) {
                  // just iterate to validate
                  Iterator<?> keys =
dataToReducersHashJSON.keys();
                  while( keys.hasNext() ){
                        Integer keyRel =
Integer.parseInt((String) keys.next());
                        if (keyRel == relation) {
                              Integer reducerToSend =
dataToReducersHashJSON.getInt(keyRel.toString());
                              return reducerToSend;
                        }
                  }
            }
            return 0;
      }
      public Integer getReducerToSend(Integer relation)
      {
            ObjectInputStream is;
            try {
                  is = new ObjectInputStream(new
FileInputStream("loadBalancing"));
                  dataToReducers = (List<Reducer>)
```

```
is.readObject();
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            } catch (ClassNotFoundException e) {
                  e.printStackTrace();
            }
            if (dataToReducers != null && dataToReducers.size()
> 0) {
                  for (int j = 0; j < dataToReducers.size();
j++) {
                        Reducer reducer = dataToReducers.get(j);
                        for (int i = 0; i <
reducer.getRelation().size(); i++) {
                              if (reducer.getRelation().get(i) ==
relation) {
                                    return
reducer.getReducerToSend();
                              }
                        }
                  }
            }
            return 0;
      }
}
```