ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Κατανεμημένοι Αλγόριθμοι Ερωτημάτων Ένωσης με Εφαρμογές στην Ανάλυση Δεδομένων Δικτυακής Κίνησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος Α. Σαρλής

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2014

Εθνικο Μετσοβιο Πολυτεχνειο
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Κατανεμημένοι Αλγόριθμοι Ερωτημάτων Ένωσης με Εφαρμογές στην Ανάλυση Δεδομένων Δικτυακής Κίνησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος Α. Σαρλής

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Νοεμβρίου 2014.

.....	.....	.....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Νοέμβριος 2014.

.....

Δημήτριος Α. Σαρλής
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια έχει παρατηρηθεί ραγδαία αύξηση της κίνησης στο Internet, γεγονός που είναι περισσότερο εμφανές σε κόμβους ουδέτερης διασύνδεσης (IXPs) από τους οποίους πλέον διέρχονται έως και petabytes δεδομένων καθημερινά. Υπάρχει ανάγκη, λοιπόν, για επεξεργασία αυτού του τεράστιου όγκου δεδομένων με αποδοτικές τεχνικές, για να εξαχθούν στατιστικά για την κίνηση που διέρχεται από αυτούς τους κόμβους.

Στην παρούσα διπλωματική, ασχολούμαστε με τη σχεδίαση και υλοποίηση ενός συστήματος ανάλυσης δεδομένων δικτυακής κίνησης τύπου sFlow που θα χρησιμοποιεί τεχνικές κατανεμημένης επεξεργασίας, όπως το MapReduce σε αντίθεση με τις παραδοσιακές τεχνικές που χρησιμοποιούνται μέχρι τώρα. Το σύστημα αυτό θα είναι σε θέση να αντιμετωπίσει τη γενικότερη περίπτωση του log processing όπου έχουμε ένα βασικό σύνολο δεδομένων και θέλουμε να εξάγουμε πληροφορία από αυτό σε συνδυασμό με εξωτερικές πηγές επιπλέον πληροφορίας.

Για το σκοπό αυτό εξετάζουμε αποδοτικές τεχνικές με τις οποίες μπορεί να γίνει η συνένωση των πληροφοριών, όπως είναι η τεχνική του map join. Συνδυάζουμε αυτή τη μέθοδο με εξειδικευμένες συναρτήσεις UDF στο Hive για να επιτύχουμε καλύτερη απόδοση. Ακόμη, προτείνουμε ένα έξυπνο τρόπο για pre-partitioning των δεδομένων με τη χρήση ενός K-d tree, ώστε να μπορεί να γίνει γρήγορα η εκτέλεση ερωτημάτων που αφορούν περιορισμένο τμήμα των δεδομένων (με χρήση διάφορων φίλτρων). Στη συνέχεια εξετάζουμε την επίδραση διαφορετικών συστημάτων εκτέλεσης MapReduce στα ίδια ερωτήματα και συγκρίνουμε τα χαρακτηριστικά τους. Τέλος, παρουσιάζουμε τη δυνατότητα κλιμάκωσης του συστήματος που υλοποιήσαμε, καθώς αυξάνει ο αριθμός των διαθέσιμων κόμβων αλλά και το μέγεθος του συνόλου των δεδομένων. Σε κάθε περίπτωση η δική μας μέθοδος παρουσιάζει μία βελτίωση έως και 70% στο χρόνο εκτέλεσης σε σύγκριση με μία απλή βασική υλοποίηση.

Λέξεις κλειδιά

SFlow, Hadoop, MapReduce, Hive, Spark, Shark, HBase, NoSQL, K-d Tree, Map Join

Abstract

In recent years Internet traffic has increased rapidly, a fact that is more obvious in Internet eXchange Points (IXPs) which handle even petabytes of data daily. Therefore, a need for efficiently processing this vast amount of data emerges, in order to extract statistics concerning the traffic that comes through these nodes.

In this thesis, we design and implement a network monitoring data analysis system that utilizes distributed processing techniques, like MapReduce, unlike traditional approaches used until now. This system is able to deal with the more general case of log processing, where we have a basic dataset available and we want to extract information in combination with external data sources.

To achieve this, we investigate efficient techniques for joining the various information, like map join. We combine this method with specialized UDF functions in Hive for enhancing performance. Furthermore, we suggest a smart way of pre-partitioning the data with the use of a K-d tree, so we can efficiently run queries that refer to a limited portion of the dataset (using various filters). Moreover, we test the effect of different MapReduce engines in executing the same queries and we compare their performance characteristics. Finally, we study the system's scalability when we modify both the cluster size as well as the dataset size. In any case we noticed an improvement of 70% in execution time using our approach compared to the baseline.

Keywords

SFlow, Hadoop, MapReduce, Hive, Spark, Shark, HBase, NoSQL, K-d Tree, Map Join

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Καθηγητή Νεκτάριου Κοζύρη και αποτελεί το επιστέγασμα των προπτυχιακών σπουδών μου.

Αρχικά, θέλω να ευχαριστήσω τον Καθηγητή μου Νεκτάριο Κοζύρη, για τη δυνατότητα που μου έδωσε να ασχοληθώ με ένα σύγχρονο και πολύ ενδιαφέρον θέμα. Ιδιαίτερα θα ήθελα να ευχαριστήσω το μεταδιδακτορικό ερευνητή Γιάννη Κωνσταντίνου για τη συνεχή καθοδήγησή του και το χρόνο που αφιέρωσε για να με βοηθήσει όποτε χρειάστηκε. Ακόμα, θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα Νίκο Παπαηλίου για τις διάφορες συμβουλές που μου έδωσε και την βοήθεια του σε τεχνικά ζητήματα όποτε του το ζήτησα. Τέλος, θα ήταν παράλειψη μου να μην ευχαριστήσω το μεταδιδακτορικό ερευνητή Γεώργιο Σμαραγδάκη του Τεχνολογικού Πανεπιστημίου του Βερολίνου. Η διπλωματική αυτή στηρίχτηκε κατά ένα σημαντικό ποσοστό σε προηγούμενη ερευνητική του δουλειά και θέλω να τον ευχαριστήσω θερμά για τις συζητήσεις που κάναμε και οι οποίες υπήρξαν διαφωτιστικές για την περαιτέρω πορεία της παρούσας εργασίας.

Επίσης, θέλω να ευχαριστήσω όλους τους φίλους και συμφοιτητές μου για την παρουσία τους καθόλη τη διάρκεια των σπουδών μου, αλλά και εκτός των πραγμάτων της σχολής. Χωρίς αυτούς αυτά τα χρόνια δεν θα ήταν το ίδιο ωραία.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου, τους γονείς μου και τις δύο αδερφές μου, που είναι πάντα δίπλα μου και με υποστηρίζουν σε κάθε επιλογή μου. Τους ευχαριστώ για την υπομονή και την κατανόησή τους στις καλές αλλά και στις άσχημες στιγμές.

Δημήτρης Σαρλής,
Αθήνα, 5 Νοεμβρίου 2014

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Οι κόμβοι ουδέτερης διασύνδεσης (Internet eXchange Points - IXPs) είναι φυσικός εξοπλισμός (routers, switches κ.λπ.) που επιτρέπει την άμεση διασύνδεση παρόχων υπηρεσιών Internet (Internet Service Providers - ISPs) με σκοπό την ανταλλαγή κίνησης δεδομένων Internet μεταξύ των δικτύων τους (Autonomous Systems - AS). Στην Ελλάδα, ο κόμβος GR-IX διασυνδέει όλους τους ελληνικούς ISPs: με αυτό τον τρόπο, η επικοινωνία μεταξύ δύο ελληνικών ISPs γίνεται απευθείας μέσω του GR-IX χωρίς να απαιτείται η δρομολόγηση των πακέτων μεταξύ ενός τρίτου δικτύου που βρίσκεται π.χ. στο εξωτερικό.

Προφανώς, ένα IXP δρομολογεί κίνηση τάξης μεγέθους μεγαλύτερη σε σχέση με ένα συγκεκριμένο ISP . Πολλές σύγχρονες μελέτες έχουν δείξει ότι η δειγματοληπτική ανάλυση της κίνησης ενός IXP σε ένα βάθος χρόνου μερικών εβδομάδων μπορεί να εξάγει ενδιαφέροντα συμπεράσματα όχι μόνο για τα συγκεκριμένα AS που διασυνδέει, αλλά και για την κατάσταση ολόκληρου του διαδικτύου. Χρησιμοποιώντας το εργαλείο sFlow [5] , οι προγούμενες μελέτες συγκέντρωσαν ένα δείγμα των πακέτων που δρομολογήθηκαν. Η ανάλυση των δειγμάτων απέδειξε ότι ένα IXP έχει τέλεια "ορατότητα" του συνολικού διαδικτύου, καθώς μέσα από αυτό περνάει κίνηση προς όλα σχεδόν τα υπάρχοντα AS και για όλα τα προθέματα δημόσιων δικτύων [12].

Οι προηγούμενες μελέτες [12] χρησιμοποίησαν παραδοσιακές τεχνικές επεξεργασίας κειμένου δεδομένων σε συγκεντρωμένα log files: τα δεδομένα αποθηκεύονται σε απλά αρχεία στο filesystem ενός υπολογιστή και κατόπιν με τη χρήση κάποιων προγραμμάτων-scripts εξάγεται η απαραίτητη πληροφορία (π.χ. uniqueIPs, κίνηση μεταξύ διαφορετικών ASes, χώρες παραληπτών/αποστολέων, ταξινομημένες λίστες με βάση τον αριθμό των εμφανίσεων κ.λπ.). Είναι προφανές ότι η ισχύς του υπολογιστή αυτού τοποθετεί ένα όριο στις δυνατότητες κλιμάκωσης της επεξεργασίας σε μεγαλύτερο αριθμό δεδομένων: σε αυτήν την περίπτωση, ο χρόνος της ανάλυσης είναι

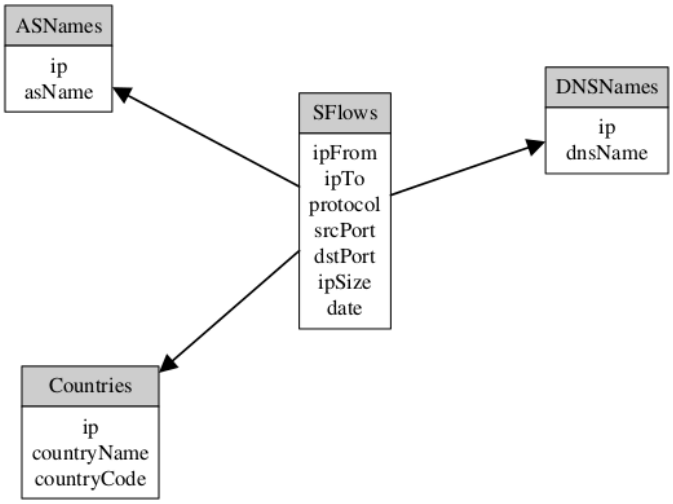
ανάλογος του μεγέθους των δεδομένων (π.χ. διπλάσια δεδομένα χρειάζονται διπλάσιο χρόνο για να αναλυθούν). Επίσης, το μέγιστο δυνατό μέγεθος των δεδομένων προς ανάλυση περιορίζεται από τις δυνατότητες του συγκεκριμένου υπολογιστή (συνήθως επηρεάζεται από το μέγεθος της φυσικής μνήμης του μηχανήματος).

Έχει παρατηρηθεί ότι τα δεδομένα που παράγονται τα τελευταία χρόνια έχουν αυξηθεί δραματικά [1]. Ως εκ τούτου, οι τεχνικές ανάλυσης μεγάλου όγκου δεδομένων (big data analytics) έχουν γνωρίσει ιδιαίτερη άνθηση [16, 19, 18]. Οι τεχνικές αυτές βασίζονται σε κατανεμημένες προσεγγίσεις, όπου συστοιχίες υπολογιστών χρησιμοποιούνται συνεργατικά για την επεξεργασία τεράστιου όγκου δεδομένων, με σκοπό την εύκολη κλιμάκωση της υποδομής καθώς αυξάνονται τα δεδομένα. Τεχνικές όπως το μοντέλο MapReduce της Google χρησιμοποιούνται κατά κόρον για το σκοπό αυτό.

1.2 Σκοπός και σύντομη περιγραφή της εργασίας

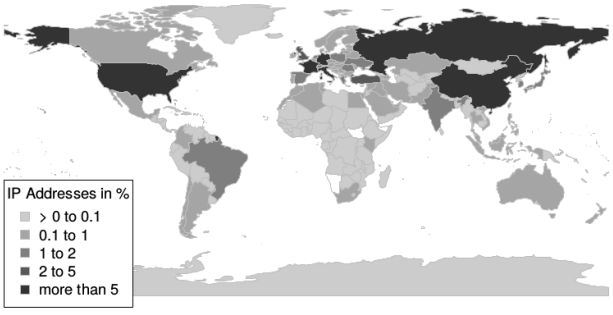
Σκοπός της παρούσας διπλωματικής εργασίας είναι η σχεδίαση και υλοποίηση ενός συστήματος επεξεργασίας μεγάλου όγκου δεδομένων που έχουν τη μορφή ενός κλασικού star schema [14], όπως για παράδειγμα είναι τα δεδομένα που προέρχονται από log files και δευτερεύουσες πληροφορίες που ενισχύουν αυτά τα δεδομένα με επιπλέον γνωρίσματα. Μία ειδική περίπτωση εφαρμογής αυτού του μοντέλου δεδομένων είναι και η ανάλυση πραγματικών δεδομένων κίνησης ενός IXP. Στην περίπτωση αυτή τα δεδομένα που προέρχονται από τον IXP είναι ο λεγόμενος fact table, ενώ οι δευτερεύουσες πληροφορίες που αφορούν τις διευθύνσεις IP των πακέτων που εξετάζουμε αποτελούν τους λεγόμενους dimension tables. Στο σχήμα 1.1 φαίνεται εποπτικά το μοντέλο δεδομένων που επιθυμούμε να εξετάσουμε.

Στόχος μας, λοιπόν, είναι χρησιμοποιώντας τεχνικές κατανεμημένης επεξεργασίας που βασίζονται στο μοντέλο MapReduce και προγράμματα του οικοσυστήματος Hadoop να επεξεργαστούμε αυτά τα δεδομένα, εκτελώντας ερωτήματα συνένωσης που θα συνδυάζουν την πληροφορία από τον πίνακα sFlows με τους δευτερεύοντες πίνακες που περιέχουν τη συμπληρωματική πληροφορία. Συγκεκριμένα, ενδιαφερόμαστε για την εκτέλεση ερωτημάτων που θα παράγουν στατιστική πληροφορία για τα κορυφαία ζεύγη αυτόνομων συστημάτων, χωρών προέλευσης ή ονομάτων DNS (αλλιώς top-K queries) [12]. Ένα παράδειγμα τέτοιας επεξεργασίας δείχνει το σχήμα 1.2 που παρουσιάζει για μια εβδομάδα τον αριθμό των ASes που ανταλλάσσουν κίνηση μέσω ενός μεγάλου ευρωπαϊκού IXP και την ποσοστιαία κατανομή των διευθύνσεων ανάλογα με τη χώρα προέλευσης.



Σχήμα 1.1: Μοντέλο δεδομένων

		week 45	educated guesses of ground-truth
Peering Traffic	IPs	232,460,635	unknown $< 2^{32}$
	#ASes	42,825	approx. 43K
	Subnets	445,051	450K+
	countries	242	250
Server Traffic	IPs	1,488,286	unknown
	#ASes	19,824	unknown
	Subnets	75,841	unknown
	Countries	200	250



Σχήμα 1.2: Παράδειγμα επεξεργασίας δεδομένων ενός IXP

Από τον πίνακα του σχήματος 1.2 αξίζει να επεξηγήσουμε τους όρους που χρησιμοποιούνται. Αρχικά, ο όρος peering traffic αναφέρεται στο τμήμα της συνολικής κίνησης που προκύπτει αν αφαιρέσουμε κάποια στοιχεία που δεν μας ενδιαφέρουν. Συγκεκριμένα, αφαιρούμε την κίνηση που:

- Δεν αποτελεί κίνηση που αφορά IPv4 πακέτα δεδομένων.
- Κίνηση που δεν ανταλλάσσεται μεταξύ δύο αυτόνομων συστημάτων που είναι μέλη του συγκεκριμένου IXP ή είναι τοπική (για παράδειγμα πακέτα διαχείρισης του IXP).
- Κίνηση που δεν αφορά πρωτόκολλα TCP ή UDP (για παράδειγμα πακέτα τύπου ICMP).

Ο όρος server traffic αφορά όπως είναι αναμενόμενο την κίνηση που σχετίζεται με web servers είτε εισερχόμενη είτε εξερχόμενη. Μπορούμε εξετάζοντας το πακέτο, να αναζητήσουμε για μεθόδους του πρωτοκόλλου HTTP (όπως GET, POST) ή να ανζητήσουμε τις συνηθισμένες θύρες που χρησιμοποιούνται για το πρωτόκολλο αυτό (80 και 443). Τέλος, στον πίνακα εκτός από τα στοιχεία που προέκυψαν ως αποτέλεσμα των πειραμάτων γίνεται και μια εκτίμηση του συνολικού αριθμού παγκοσμίως (educated guesses of ground-truth) για να γίνει αντιληπτό ότι ένας IXP συναλλάσσεται κίνηση που αφορά μεγάλο μέρος του συνολικού διαδικτύου.

Βασικά ερωτήματα που θα μας απασχολήσουν είναι αυτά που χρησιμοποιώντας την πληροφορία για τις διευθύνσεις IP που ανταλλάσσουν πληροφορία, θα μπορούν να τη μετατρέπουν σε αντίστοιχη πληροφορία ανάλογα με το δευτερεύοντα πίνακα που θα χρησιμοποιείται. Για παράδειγμα, από την πληροφορία για τις διευθύνσεις που ανταλλάσσουν μεταξύ τους κίνηση, θα παράγεται πληροφορία για τα αυτόνομα συστήματα που εμπλέκονται στην ανταλλαγή πακέτων. Για να γίνει αυτό πρέπει να λάβουμε υπόψη μας διάφορες παραμέτρους του προβλήματος, όπως ο αλγόριθμος που θα χρησιμοποιηθεί για τη συνένωση των πληροφοριών ή ο τρόπος που θα αποθηκεύονται τα δεδομένα ώστε να μπορεί να γίνει αποδοτικά η επεξεργασία τους. Οι κυριότερες συνεισφορές αυτής της εργασίας έγκεινται στα εξής:

- Προτείνουμε μια αποδοτική υλοποίηση για τον αλγόριθμο συνένωσης της πληροφορίας, που κάνει χρήση της τεχνικής του map join και εξειδικευμένες συναρτήσεις UDF.
- Παρουσιάζεται ένας έξυπνος τρόπος για pre-partitioning των δεδομένων ανάλογα με τις εγγραφές που περιέχουν, για να μπορεί να γίνει πιο αποδοτικά η επεξεργασία τους κατά τη διαδικασία της συνένωσης.
- Συνδυάζοντας τα δύο παραπάνω μπορούμε να εκτελέσουμε ειδικότερα ερωτήματα που αφορούν ένα περιορισμένο τμήμα των δεδομένων σε αρκετά μικρό χρονικό διάστημα έως μερικά λεπτά, όταν τα ερωτήματα που αφορούν όλο το dataset μπορεί να απαιτούν μερικές ώρες σε κάποιες περιπτώσεις.

1.3 Οργάνωση του κειμένου

Στο Κεφάλαιο 2 παρουσιάζεται το γενικό θεωρητικό υπόβαθρο που αφορά τα διάφορα frameworks και εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση της εργασίας. Περιγράφεται, ακόμα το σύνολο δεδομένων που εξετάσαμε και το cluster των υπολογιστών που χρησιμοποιήθηκε για την εκτέλεση των πειραμάτων.

Στο Κεφάλαιο 3 παρουσιάζεται η διαδικασία που ακολουθήθηκε για την υλοποίηση της συνένωσης των δεδομένων με τη δευτερεύουσα πληροφορία στην περίπτωση που το meta-dataset είναι σχετικά μικρό και μπορεί να χωρέσει στην κύρια μνήμη μίας διεργασίας map. Παρουσιάζονται οι διάφοροι αλγόριθμοι που εξετάστηκαν και υλοποιήθηκαν, ενώ πραγματοποιείται μία σύγκριση ανάμεσα στις διαφορετικές προσεγγίσεις όπου αναλύονται τα χαρακτηριστικά της κάθε μιας.

Στο Κεφάλαιο 4 προσαρμόζουμε την τεχνική της συνένωσης δύο συνόλων δεδομένων ώστε να μπορεί να εφαρμοστεί και σε περιπτώσεις που το δευτερεύον σύνολο δεν χωράει στην κύρια μνήμη μίας διεργασίας. Αρχικά, συγκρίνουμε τη βασική υλοποίηση που προσφέρει το Hive με μία προσέγγιση που διαχωρίζει στατικά τα δεδομένα βάσει των διευθύνσεων IP που περιέχουν και παρουσιάζεται ο τρόπος υλοποίησης και τα αποτελέσματα που δίνουν αυτές οι δύο μέθοδοι. Στη συνέχεια, προτείνονται και αναλύονται οι επιδόσεις δύο βελτιστοποιήσεων. Η πρώτη αφορά τη διαδικασία που γίνεται ένα scan query στην HBase, ενώ η δεύτερη αφορά τον τρόπο που διαχωρίζονται τα δεδομένα και γίνεται δυναμικά με τη βοήθεια μίας δομής δεδομένων που ονομάζεται K-d tree.

Στο Κεφάλαιο 5 κάνουμε μια αποτίμηση των επιδόσεων των διάφορων υλοποιήσεων. Παρουσιάζουμε πειραματικά αποτελέσματα που αφορούν την επίδραση που έχουν διαφορετικές παραμέτροι του συστήματος. Αρχικά, συγκρίνουμε τις επιδόσεις δύο διαφορετικών engines για την εκτέλεση εργασιών Map Reduce, του Hadoop σε σχέση με το Spark και συνοψίζουμε τα πλεονεκτήματα του κάθε συστήματος. Στη συνέχεια, εξετάζουμε την κλιμακωσιμότητα των υλοποιήσεών μας όσον αφορά τους κόμβους του cluster που είναι διαθέσιμοι καθώς επίσης και το συνολικό μέγεθος των δεδομένων προς επεξεργασία. Τέλος, παρουσιάζουμε την επίδραση του αριθμού των διαστάσεων που χρησιμοποιούνται στο K-d tree στην εκτέλεση των ερωτημάτων.

Στο Κεφάλαιο 6 γίνεται μία ανασκόπηση της υλοποίησης του συστήματος και συνοψίζονται τα τελικά συμπεράσματα και οι συνεισφορές της εργασίας αυτής. Ακόμα, παρουσιάζονται τυχόν επεκτάσεις της παρούσας δουλειάς σε διαφορετικές κατευθύνσεις.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 MapReduce

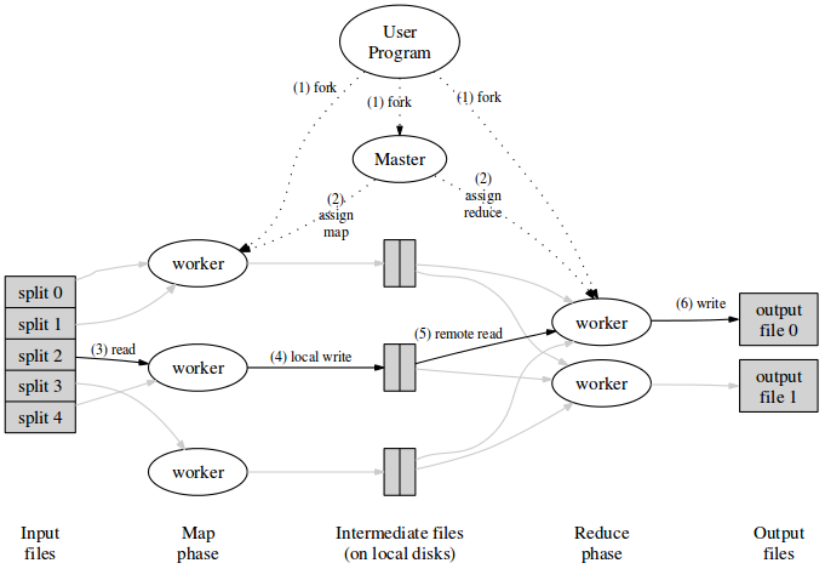
Το MapReduce είναι ένα προγραμματιστικό μοντέλο που παρουσιάστηκε από τη Google [13] και διευκολύνει την κατανεμημένη επεξεργασία τεράστιων ποσοτήτων δεδομένων, χρησιμοποιώντας ένα μεγάλο αριθμό από υπολογιστές. Οι υπολογιστές αυτοί είναι συνδεδεμένοι μεταξύ τους μέσω δικτύου (π.χ. Ethernet) και αποτελούν το λεγόμενο cluster υπολογιστών. Η ιδέα αυτή βασίζεται σε δύο κεντρικές συναρτήσεις, τις Map και Reduce , οι οποίες αποτελούν ένα γενικό μοντέλο προγραμματισμού με το οποίο μπορούμε να εκφράσουμε πολλές από τις σημερινές πραγματικές εφαρμογές. Ο προγραμματιστής πρέπει απλά να ορίσει αυτές τις δύο συναρτήσεις και το σύστημα αναλαμβάνει αυτόματα την παραλληλοποίηση και την εκτέλεση της εφαρμογής σε ένα οσοδήποτε μεγάλο cluster από υπολογιστές.

- **Συνάρτηση Map:** Η συνάρτηση αυτή παίρνει ως είσοδο ένα ζευγάρι key/value και παράγει ως έξοδο ένα σύνολο από ζευγάρια key/value. Η βιβλιοθήκη του MapReduce αναλαμβάνει να μαζέψει τις ενδιάμεσες τιμές που έχουν το ίδιο κλειδί και να τις προωθήσει ως είσοδο στην κατάλληλη συνάρτηση Reduce.

- **Συνάρτηση Reduce:** Η συνάρτηση Reduce δέχεται ένα από τα ενδιάμεσα κλειδιά (key) και μία λίστα με όλες τις τιμές (values) που αντιστοιχούν σε αυτό το κλειδί. Στη συνάρτηση αυτή ο προγραμματιστής επεξεργάζεται τις τιμές εισόδου και παράγει ένα ή περισσότερα ζευγάρια key/value.

Το πλεονέκτημα του MapReduce είναι το γεγονός ότι οι συναρτήσεις Map και Reduce είναι πλήρως παραλληλοποιήσιμες. Κάθε τέτοια συνάρτηση έχει τα δικά της δεδομένα εισόδου και μπορεί να παράγει τα αποτελέσματά της χωρίς να χρειάζεται να επικοινωνήσει με καμία άλλη. Κατά συνέπεια οι διεργασίες δεν έχουν κάποια εξάρτηση η μία από την άλλη και μπορούν να εκτελεστούν όλες ταυτόχρονα. Αυτό έχει ως αποτελέσμα να αποφεύγονται τα γνωστά προβλήματα επικοινωνίας που εμφανίζονται

σε άλλα είδη παράλληλου προγραμματισμού όταν αυξάνεται ο αριθμός των παράλληλων διεργασιών(συγχρονισμός, ανταλλαγή δεδομένων κ.λπ.). Στο βαθμό που υπάρχουν αρκετά δεδομένα εισόδου μπορούμε να μειώσουμε θεωρητικά το χρόνο εκτέλεσης σχεδόν γραμμικά σε σχέση με τον αριθμό των διαθέσιμων υπολογιστών.



Σχήμα 2.1: MapReduce Framework

Η βασική ροή εκτέλεσης ενός MapReduce job ακολουθεί τα παρακάτω βήματα:

1. Η βιβλιοθήκη εκτέλεσης αναλαμβάνει να χωρίσει το σύνολο των δεδομένων σε M κομμάτια παρόμοιου μεγέθους, το οποίο μπορεί να οριστεί από το χρήστη μέσω μίας παραμέτρου. Ο προγραμματιστής έχει τη δυνατότητα να ορίσει μία δική του συνάρτηση η οποία να χωρίζει τα δεδομένα εισόδου με τον τρόπο που απαιτείται από την εκάστοτε εφαρμογή. Στη συνέχεια δημιουργούνται πολλά αντίγραφα του προγράμματος τα οποία τρέχουν σε όλα τα μηχανήματα του cluster.

2. Ένα από αντίγραφα του προγράμματος είναι ξεχωριστό και ονομάζεται master. Τα υπόλοιπα αντίγραφα αποτελούν τους workers και ο master αναθέτει στον κάθε worker την κατάλληλη διεργασία. Υπάρχουν συνολικά M map και R reduce και ο σκοπός του master είναι να εντοπίζει ανενεργούς workers και να τους αναθέτει μία διεργασία για εκτέλεση, όπως φαίνεται και στο σχήμα 2.1.

3. Όταν ένας από τους workers αναλάβει μια διεργασία map διαβάζει το αντίστοιχο κομμάτι δεδομένων (input split) και χρησιμοποιεί είτε μία από τις διαθέσιμες

συναρτήσεις ανάγνωσης δεδομένων για να παράγει τα κατάλληλα ζευγάρια key/value είτε μία που έχει ορίσει ο προγραμματιστής. Η συνάρτηση map επεξεργάζεται τα ζευγάρια key/value με τον τρόπο που έχει ορίσει ο προγραμματιστής και παράγει νέα ζευγάρια key/value τα οποία γράφονται σε buffers.

4. Ανά τακτά χρονικά διαστήματα τα ενδιάμεσα ζευγάρια key/value που έχουν γραφτεί στους buffers γράφονται στον τοπικό σκληρό δίσκο του μηχανήματος. Τα ζευγάρια αυτά χωρίζονται σε R ξεχωριστές περιοχές σύμφωνα με μια συνάρτηση διαχωρισμού(partitioning function). Αυτή η συνάρτηση αναλαμβάνει να χωρίσει τα ενδιάμεσα ζευγάρια key/value ώστε να προετοιμάσει την είσοδο για τις διεργασίες reduce. Ο διαχωρισμός των ενδιάμεσων ζευγαριών key/value γίνεται βάσει του κλειδιού του κάθε ζευγαριού. Μπορούμε να χρησιμοποιήσουμε την προκαθορισμένη συνάρτηση διαχωρισμού $hash(key)\ mod\ R$ ή να ορίσουμε την δικιά μας συνάρτηση. Σε κάθε περίπτωση πρέπει να προσέξουμε να τηρείται ο περιορισμός τα ζευγάρια με το ίδιο κλειδί να τοποθετούνται στην ίδια περιοχή. Αφού αποθηκευθούν τα ζευγάρια στο δίσκο του μηχανήματος ο worker στέλνει τις θέσεις τους στο master ο οποίος είναι υπεύθυνος να προωθήσει τις θέσεις αυτές στους reducers.

5. Ο reducer ενημερώνεται από το master για τις θέσεις στις οποίες βρίσκονται τα ζευγάρια key/value που πρέπει να επεξεργαστεί και χρησιμοποιεί Remote Procedure Calls (RPC) για να διαβάσει τα δεδομένα από τους δίσκους των μηχανημάτων που βρίσκονται. Όταν διαβάσει όλα τα ζευγάρια εισόδου ο reducer τα ταξινομεί βάσει του κλειδιού του κάθε ζευγαριού. Το αποτέλεσμα αυτής της διαδικασίας είναι ότι τα ζευγάρια με το ίδιο κλειδί ομαδοποιούνται, ώστε για κάθε κλειδί ο reducer να επεξεργάζεται όλες τις τιμές ταυτοχρόνως.

6. Ο reducer επεξεργάζεται με τη σειρά όλα τα ενδιάμεσα ζευαρια key/valu σύμφωνα με τη συνάρτηση reduce που έχει οριστεί από τον προγραμματιστή. Τα ζευγάρια εξόδου των διεργασιών reduce γράφονται στο τελικό αρχείο εξόδου που αντιστοιχεί σε αυτή την περιοχή (partition).

7. Όταν όλες οι διεργασίες map και reduce έχουν εκτελεστεί ο master ενημερώνει το πρόγραμμα του χρήστη και συνεχίζεται η εκτέλεσή του. Μετά την επιτυχημένη ολοκλήρωση της εργασίας MapReduce η έξοδος βρίσκεται στα R αρχεία εξόδου.

Ένας πιο αυστηρός ορισμός των συναρτήσεων map και reduce με βάση την είσοδο και την έξοδό τους είναι ο ακόλουθος:

$$Map(k1, v1) \rightarrow list(k2, v2)$$
$$Reduce(k2, list(v2)) \rightarrow list(v3)$$

Από τους παραπάνω τύπους συμπεραίνουμε ότι η συνάρτηση map εφαρμόζεται σε ένα ζευγάρι key/value συγκεκριμένου τύπου και παράγει μια λίστα από ζευγάρια

διαφορετικού τύπου. Για κάθε κλειδί έχουμε μια διεργασία reduce η οποία παίρνει σαν είσοδο το κλειδί αυτό και μια λίστα που περιέχει όλες τις τιμές που αντιστοιχούν σε αυτό. Η reduce συνάρτηση παράγει και αυτή μια λίστα με τιμές οι οποίες δυνητικά είναι διαφορετικού τύπου από τις τιμές εισόδου.

2.2 Hadoop

2.2.1 Εισαγωγή

Το Hadoop είναι ένα προγραμματιστικό πλαίσιο που αναπτύσσεται από την εταιρία Apache Software Foundation και υποστηρίζει κατανεμημένη επεξεργασία μεγάλου όγκου δεδομένων. Η ανάπτυξή του ξεκίνησε το 2005 ως μια εναλλακτική επιλογή ανοιχτού κώδικα του Google Map Reduce framework και το γεγονός αυτό δίνει μεγάλη ελευθερία παραμετροποίησης στον προγραμματιστή. Ένας από τους σημαντικότερους λόγους επιτυχίας του συστήματος αυτού είναι ότι σχεδιάστηκε για να μπορεί να τρέχει σε συστάδες από πολλούς απλούς υπολογιστές (cluster) χαμηλού κόστους και αναλαμβάνει όλες τις απαραίτητες λειτουργίες για την εκτέλεση των εργασιών MapReduce . Τα προγράμματα που γράφουν οι χρήστες μπορούν να τρέξουν σε οποιοδήποτε cluster μηχανημάτων παρέχοντας ταυτόχρονα ανοχή στα σφάλματα υλικού, δηλαδή ακόμα και αν κάποιος υπολογιστής παρουσιάσει πρόβλημα (π.χ. αστοχία δίσκου, πρόβλημα στο δίκτυο) δεν θα διακοπεί η εκτέλεση του προγράμματος αλλά η δουλειά που εκτελούσε θα μοιραστεί σε άλλους διαθέσιμους κόμβους.

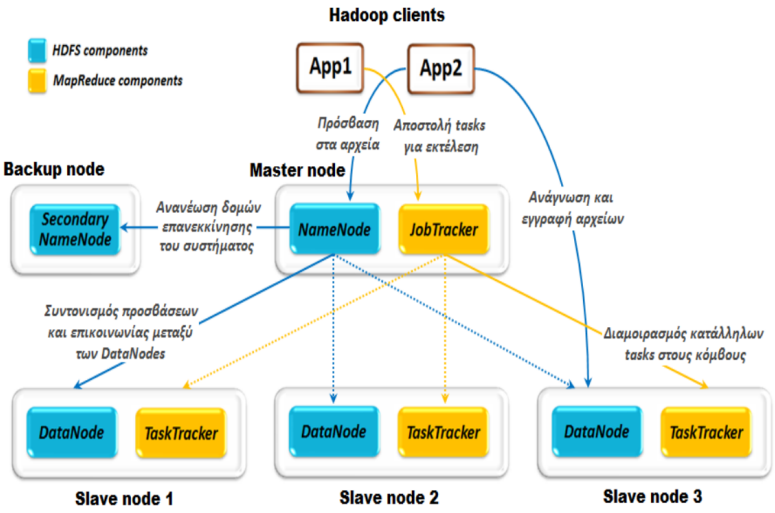
2.2.2 Ανάλυση του συστήματος

Ένα Hadoop cluster αποτελείται από έναν master και πολλούς slaves. Επομένως, πρέπει να εγκατασταθούν διάφορες διεργασίες στον master και στους slave κόμβους του cluster , ώστε να γίνεται σωστά η επικοινωνία μεταξύ των κόμβων, καθώς και η εκτέλεση των εφαρμογών MapReduce .

Στο σχήμα 2.2 φαίνεται η αρχιτεκτονική που χρησιμοποιεί το Hadoop. Σύμφωνα με αυτό στον master κόμβο εγκαθιστούμε τις παρακάτω διεργασίες:

- **JobTracker:** Ο JobTracker είναι η διεργασία του Hadoop η οποία είναι υπεύθυνη ώστε να αναθέτει MapReduce tasks σε συγκεκριμένους κόμβους του cluster. Η διεργασία αυτή προσπαθεί να αναθέσει τις εργασίες έτσι ώστε ο κάθε κόμβος να έχει τα δεδομένα εισόδου διαθέσιμα τοπικά ή τουλάχιστον να βρίσκονται σε κάποιον γειτονικό του κόμβο. Ο JobTracker είναι κρίσιμο σημείο για τη λειτουργία του MapReduce αφού αν σταματήσει να λειτουργεί θα σταματήσει και η εκτέλεση όλων των εργασιών MapReduce.

- **NameNode:** Ο NameNode είναι η κεντρική εφαρμογή που ελέγχει το HDFS το κατανεμημένο σύστημα αρχείων του Hadoop . Η διεργασία αυτή διαθέτει



Σχήμα 2.2: Hadoop Architecture

το δέντρο του συστήματος αρχείων, το οποίο περιέχει όλα τα αρχεία του συστήματος και επίσης γνωρίζει που βρίσκεται αποθηκευμένο το κάθε αρχείο στο cluster. Ο NameNode δέχεται αιτήσεις από τον κώδικα πελάτη κάθε φορά που θέλουμε να εντοπίσουμε, δημιουργήσουμε, αντιγράψουμε, μετακινήσουμε ή διαγράψουμε ένα αρχείο και επιστρέφει μια λίστα με τους Datanode servers στους οποίους βρίσκονται τα αντίστοιχα δεδομένα. Ο NameNode είναι και αυτός κρίσιμο σημείο για τη λειτουργία του συστήματος, αφού αν σταματήσει να λειτουργεί το filesystem θα πάψει να ανταποκρίνεται. Υπάρχει μια δευτερεύουσα διεργασία ο SecondaryNameNode που δημιουργεί checkpoint σε τακτά χρονικά διαστήματα και μπορεί να επαναφέρει το σύστημα αρχείων μετά από διακοπή λειτουργίας του NameNode. Η διεργασία του NameNode είναι από τις σημαντικότερες ενός Hadoop cluster και είναι αρκετά απαιτητική σε μνήμη, γι΄ αυτό πρέπει να τοποθετείται αν είναι δυνατόν μόνη της σε μηχάνημα που έχει αρκετή διαθέσιμη μνήμη RAM.

Στους slave κόμβους εγκαθιστούμε τις παρακάτω διεργασίες:

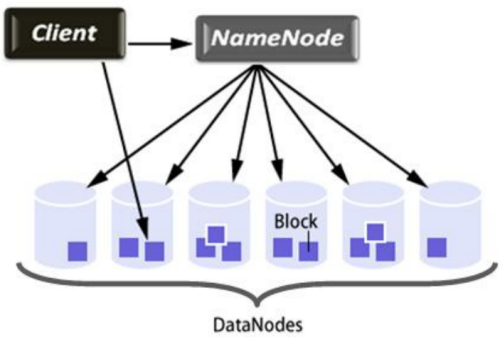
- **TaskTracker:** Η διεργασία αυτή αναλαμβάνει να εκτελέσει τις εργασίες που στέλνει ο JobTracker. Κάθε TaskTracker μπορεί να εκτελεί ταυτόχρονα έναν προκαθορισμένο αριθμό από εργασίες MapReduce , ο οποίος ορίζεται από στις παραμέτρους του συστήματος και εξαρτάται από την υπολογιστική δύναμη που έχει κάθε κόμβος στο cluster .

- **DataNode:** Αυτή είναι η διεργασία η οποία αναλαμβάνει την αποθήκευση δεδομένων στο κατανεμημένο σύστημα αρχείων HDFS . Ο DataNode επεξερ-

γάζεται τα αιτήματα που του στέλνει ο NameNode και τα οποία αφορούν την εκτέλεση λειτουργιών του συστήματος αρχείων.

2.2.3 Hadoop Distributed File System (HDFS)

Το προγραμματιστικό πακέτο του Hadoop περιέχει και μια υλοποίηση ενός κατανεμημένου συστήματος αρχείων το οποίο ονομάζεται Hadoop Distributed File System (HDFS). Το HDFS είναι ένα κατανεμημένο, κλιμακώσιμο και ανεξάρτητο από το υλικό σύστημα αρχείων που είναι γραμμένο σε Java . Ένα HDFS cluster είναι ένα σύνολο από DataNodes , οι οποίοι εξυπηρετούν την αποθήκευση και επεξεργασία των αρχείων σε κάθε κόμβο του cluster . Το HDFS χρησιμοποιεί το πρωτόκολλο TCP/IP για την επικοινωνία μεταξύ των DataNodes και τη μεταφορά των δεδομένων.



Σχήμα 2.3: HDFS Architecture

Τα βασικά πλεονεκτήματα της χρήσης του HDFS είναι:

- Το γεγονός ότι έχει σχεδιαστεί να συνεργάζεται με το Hadoop για την εκτέλεση εργασιών MapReduce (μπορεί με αποδοτικό τρόπο να προγραμματιστούν οι εργασίες γνωρίζοντας εκ των προτέρων τη θέση των δεδομένων στο cluster).

- Έχει μεγάλη ανοχή στα σφάλματα hardware και software . Πετυχαίνει αυτή την αξιοπιστία με το να κρατάει πολλά αντίγραφα των αρχείων σε διαφορετικούς κόμβους του cluster (redundancy δεδομένων).

- Μπορούμε να αποθηκεύουμε μεγάλα σε όγκο αρχεία δεδομένων, γεγονός που γίνεται εφικτό επειδή τα δεδομένα αποθηκεύονται κατά block . Συνέπεια αυτού είναι ένα μεγάλο αρχείο θα σπάσει σε πολλά μικρά blocks που θα κατανεμηθούν σε όλους τους υπολογιστές του cluster .

- Είναι κλιμακώσιμο και μπορεί να εγκατασταθεί εύκολα σε μηχανήματα χωρίς ιδιαίτερες απαιτήσεις σε υλικό.

2.3 HBase

Η HBase είναι ένα πακέτο ανοιχτού κώδικα που υλοποιεί μια μη σχεσιακή (NoSQL), κατανεμημένη βάση δεδομένων που έχει σχεδιαστεί σύμφωνα με το άρθρο της Google BigTable [11]. Το πακέτο αυτό έχει δημιουργηθεί και αυτό από την εταιρία Apache ώστε να λειτουργεί συνεργατικά με το Hadoop και να του παρέχει επιπλέον δυνατότητες αποθήκευσης. Οι εργασίες MapReduce μπορούν να χρησιμοποιήσουν την HBase για είσοδο και έξοδο δεδομένων ή απλά να έχουν πρόσβαση στα δεδομένα της κατά τη διάρκεια εκτέλεσής τους.

Η HBase διαφέρει ριζικά από τις γνωστές μας σχεσιακές βάσεις δεδομένων καθώς δεν χρησιμοποιεί την διαπροσωπεία της SQL, ούτε το entity-relationship μοντέλο αποθήκευσης δεδομένων. Το μοντέλο που ακολουθεί η HBase για την αποθήκευση των δεδομένων είναι αυτό του key/value . Κάθε γραμμή έχει ένα key που χρησιμεύει ως αναγνωριστικό και επίσης χρησιμοποιείται ώστε τα ζευγάρια που αποθηκεύονται να ταξινομούνται σε αλφαβητική σειρά με βάση αυτό το κλειδί. Αυτό δίνει μεγάλη ευχέρεια στον προγραμματιστή, αφού του επιτρέπει να επιλέξει ακριβώς πώς θα γίνουν οι προσβάσεις στη μνήμη ώστε να επιτύχει τη μεγαλύτερη δυνατή τοπικότητα δεδομένων. Συγκεκριμένα, υποστηρίζεται η λειτουργία του scan μιας περιοχής, στο οποίο μπορούμε να δώσουμε ως όρισμα ένα start key και ένα stop key και αυτό να μας επιστρέψει όλες τις εγγραφές στο διάστημα αυτό, πολύ ταχύτερα από το να εκτελούσαμε ένα get για κάθε τιμή που μας ενδιαφέρει. Τα δεδομένα του κάθε πίνακα χωρίζονται σε περιοχές που ονομάζονται regions, κάθε μία από τις οποίες περιέχει όλες τις γραμμές και στήλες μεταξύ μίας αρχικής και τελικής τιμής και οι οποίες κατανέμονται εξίσου στους slave κόμβους του cluster.

Row Key	TimeStamp	Name Family		Address Family	
		First	Last	Apt#	Street
Row1	T1	Ranjit	Das		
	T2		Dash	44	FS Road
Row2	T3	Dominik	Williams		
	T9			20	Park Street
	T12			10	Park Street

Πίνακας 2.1: Παράδειγμα πίνακα δεδομένων στην HBase

Στον πίνακα 2.1 φαίνεται η δομή ενός πιθανού πίνακα στην HBase.Όπως παρατηρούμε υπάρχει η δυνατότητα για το ίδιο κλειδί να έχουμε περισσότερες από μία εγγραφές, οι οποίες διαχωρίζονται με το πεδίο timestamp.Αυτό εξυπηρετεί στις περιπτώσεις που μας ενδιαφέρει να έχουμε διαφορετικά αντίγραφα των ίδιων δεδομένων και επιλέγουμε κάθε φορά το κατάλληλο που ταιριάζει σε μια χρονική περίοδο. Επίσης, ένα άλλο χαρακτηριστικό είναι ότι μπορούμε να ορίσουμε διαφορετικά column families

το καθένα από τα οποία μπορεί επίσης να περιέχει πολλά διαφορετικά qualifiers.

2.4 Hive

Το Hive είναι μία αποθήκη δεδομένων (data warehouse) που διευκολύνει την επεξεργασία και ανάλυσή τους όταν αυτά βρίσκονται αποθηκευμένα σε κατανεμημένα συστήματα [16]. Είναι φτιαγμένο έτσι ώστε να τρέχει πάνω από το Hadoop και να μπορεί να αποθηκεύσει και να επεξεργαστεί τα δεδομένα που απαιτούνται στο HDFS.Το Hive προσφέρει ένα μηχανισμό κατάλληλο για να μπορεί να επιλέξει ο χρήστης τη δομή των δεδομένων και μία SQL-like γλώσσα, την HiveQL , για να μπορεί ο χρήστης να επεξεργαστεί τα δεδομένα με τον τρόπο που επιθυμεί. Με αυτή τη γλώσσα ο χρήστης μπορεί να φορτώσει δεδομένα από εξωτερικές πηγές (π.χ. HDFS) και να τα επεξεργαστεί αρκετά εύκολα. Δεν υποστηρίζονται όμως οι διαδικασίες ενημέρωσης και διαγραφής γραμμών σε ήδη υπάρχοντες πίνακες. Παρ΄ όλα αυτά, η HiveQL είναι αρκετά επεκτάσιμη, καθώς ο χρήστης μπορεί να ορίσει δικές του συναρτήσεις column transformation (UDF) υλοποιημένες σε Java.Στην περίπτωση που η επεξεργασία των δεδομένων είναι αρκετά πολύπλοκη για να γίνει με τη χρήση αυτής της γλώσσας ο χρήστης μπορεί να ορίσει τις δικές του συναρτήσεις map και reduce.

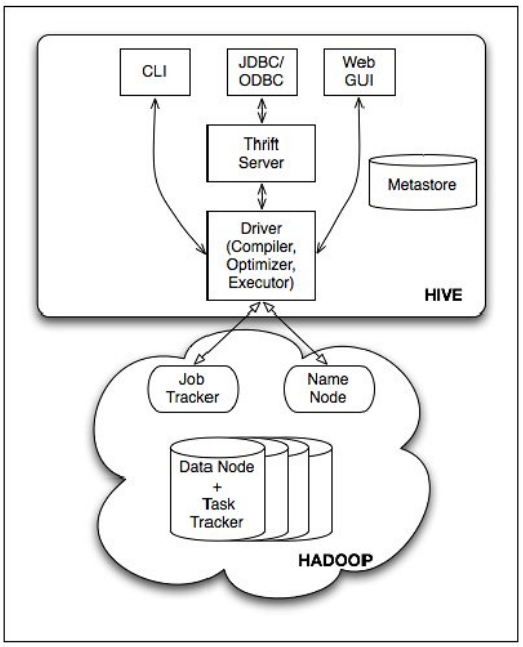
Τα δεδομένα στο Hive οργανώνονται σε:

- Tables: Οι πίνακες αυτοί είναι ανάλογοι με τους πίνακες που γνωρίζουμε από τις σχεσιακές βάσεις δεδομένων. Κάθε πίνακας έχει ένα αντίστοιχο directory στο HDFS όπου είναι αποθηκευμένα τα δεδομένα. Τα δεδομένα σειριοποιούνται και αποθηκεύονται σε διάφορα αρχεία μέσα σε αυτό το directory.Το Hive προσφέρει διάφορες μορφές σειριοποίησης που διευκολύνουν τη συμπίεση των δεδομένων, ενώ ο προγραμματιστής έχει τη δυνατότητα να ορίσει ακόμα και τις δικές του μεθόδους σειριοποίησης.

- Partitions: Κάθε πίνακας μπορεί να έχει ένα ή περισσότερα partitions που προσδιορίζουν την περαιτέρω κατανομή των δεδομένων σε υπο-φακέλους. Το πλεονέκτημα είναι ότι όλα τα δεδομένα που έχουν την ίδια τιμή στη στήλη που έχει χρησιμοποιηθεί για το partition καταλήγουν στο ίδιο sub-directory και υπάρχει η δυνατότητα να εκτελέσουμε κάποιο ερώτημα σε αυτά τα δεδομένα χωρίς να χρειαστεί να επεξεργαστούμε το σύνολο των δεδομένων.

- Buckets: Σε κάθε partition τα δεδομένα μπορούν να διαχωριστούν περαιτέρω σε buckets χρησιμοποιώντας την τιμή hash μιας στήλης του πίνακα. Κάθε bucket αποθηκεύεται ως ένα αρχείο στο directory του partition.

Στη συνέχεια αναλύουμε δύο βασικά στοιχεία της αρχιτεκτονικής του Hive, το metastore και τον compiler.



Σχήμα 2.4: Hive Architecture

Όλη η πληροφορία για τους πίνακες οργανώνεται σε ένα κατάλογο συστήματος που ονομάζεται metastore.Στο metastore αποθηκεύονται όλα τα meta-data που αφορούν τον κάθε πίνακα και τα οποία προσδιορίζονται μία φορά κατά τη δημιουργία του πίνακα και κατόπιν χρησιμοποιούνται κάθε φορά που ο πίνακας αναφέρεται σε ένα ερώτημα που εκτελείται. Οι πληροφορίες που αποθηκεύονται αφορούν στο namespace που ανήκει ο πίνακας, το οποίο καθορίζεται από τη βάση δεδομένων στην οποία ανήκει, μία λίστα με τις στήλες που περιέχει και τον τύπο δεδομένων της καθεμίας, καθώς και τυχόν partitions που έχουν οριστεί από το χρήστη.

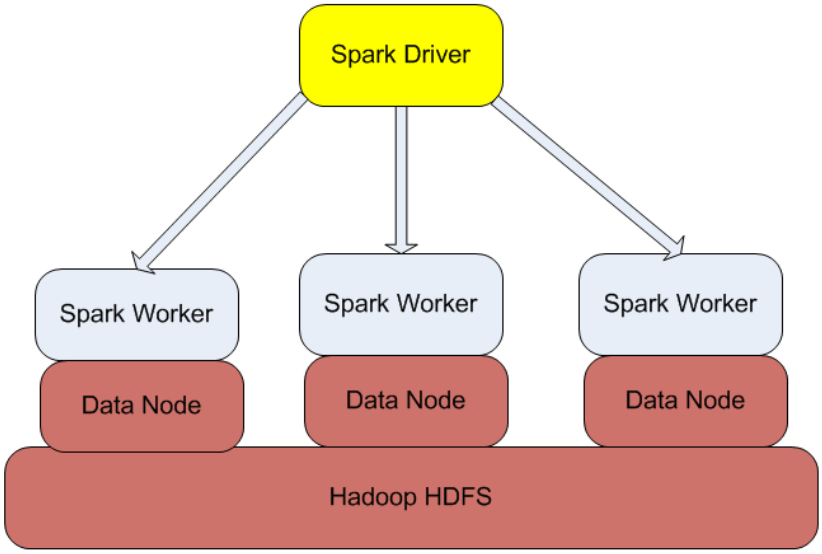
Ο compiler αναλαμβάνει να μετατρέψει το HiveQL string σε ένα πλάνο εκτέλεσης. Αρχικά, ο parser μετατρέπει το ερώτημα σε μία αναπαράσταση συντακτικού δέντρου. Ο semantic analyzer αναλαμβάνει να ανακτήσει πληροφορίες που αφορούν τα ονόματα των στηλών και να κάνει τυχόν μετατροπές τύπων δεδομένων. Ο logical plan generator μετατρέπει την εσωτερική αναπαράσταση του ερωτήματος σε ένα λογικό πλάνο εκτέλεσης αποτελούμενο από λογικούς τελεστές. Στη συνέχεια, ο optimizer βελτιστοποιεί το παραγόμενο λογικό πλάνο εφαρμόζοντας πολλαπλά περάσματα σε αυτό. Μερικές από τις λειτουργίες που επιτελεί είναι ο συνδυασμός πολλαπλών joins με κοινό κλειδί σε ένα multi-way join και συνεπώς σε μία εργασία map-reduce, η προώθηση των select όσο νωρίτερα γίνεται για να μειωθούν τα δεδομένα που πρέπει να μεταφερθούν, η επιλογή των partitions που δεν είναι απαραίτητα για το ερώτημα και ο αποκλεισμός τους σε περίπτωση που ο πίνακας είναι χωρισμένος με βάση την τιμή κάποιας στήλης του κ.λπ. Τέλος, ο physical plan generator μετατρέπει το λογικό

πλάνο σε ένα φυσικό πλάνο εκτέλεσης, το οποίο αποτελείται από ένα κατευθυνόμενο ακυκλικό γράφο (DAG) εργασιών map-reduce.

2.5 Spark και Shark

2.5.1 Εισαγωγικές έννοιες

Το μοντέλο MapReduce και όλα τα συστήματα που στηρίζονται σε αυτό, χρησιμοποιούν μία ακυκλική ροή εκτέλεσης, η οποία όμως δεν ταιριάζει σε αρκετές σύγχρονες εφαρμογές. Σε αυτές συμπεριλαμβάνονται εφαρμογές machine learning που αποτελούνται από επαναλαμβανόμενες ενέργειες, καθώς επίσης και διαδραστικά εργαλεία ανάλυσης δεδομένων (όπως το Hive). Για αυτό το λόγο αναπτύχθηκε το Spark [19] ένα εναλλακτικό σύστημα που μπορεί να υποστηρίξει τέτοιες εφαρμογές, ενώ ταυτόχρονα διατηρεί τα βασικά πλεονεκτήματα του MapReduce που είναι η δυνατότητα κλιμάκωσης και η ανοχή στα σφάλματα. Όλα αυτά επιυγχάνονται με τη βοήθεια μίας δομής που ονομάζεται Resilient Distributed Dataset (RDDs). Ένα RDD είναι μία συλλογή αντικειμένων μόνο για ανάγνωση μοιρασμένη σε ένα cluster από υπολογιστές που μπορεί να ανακτηθεί στην περίπτωση που χαθεί ένα partition.Είναι σχεδιασμένο έτσι ώστε να μπορεί να τρέξει πάνω από το Hadoop και να χρησιμοποιεί δεδομένα τα οποία βρίσκονται ήδη στο HDFS χωρίς να χρειάζεται κάποια εξειδικευμένη ρύθμιση. Η αρχιτεκτονική του συστήματος είναι αρκετά παρόμοια με αυτή του Hadoop όπως φαίνεται στο σχήμα 2.5.



Σχήμα 2.5: Spark Architecture

Ο spark driver τρέχει στο master κόμβο, ενώ οι workers τρέχουν στους slaves.Ο driver είναι υπεύθυνος να αναθέτει στους workers εργασίες και να επιτηρεί σε περίπτω-

ση που κάποιες δεν ολοκληρωθούν ή δημιουργηθεί κάποιο πρόβλημα σε αναλογία με τον JobTracker του Hadoop.

Το Shark [18] από την άλλη είναι ένα νέο εργαλείο ανάλυσης δεδομένων που τρέχει πάνω από το Spark χρησιμοποιώντας τα RDDs προκειμένου να μπορεί να τρέξει SQL ερωτήματα πολύ αποδοτικότερα από ό,τι τα ήδη υπάρχοντα συστήματα. Για να γίνει αυτό εφικτό έγιναν κάποιες τροποποιήσεις στο σύστημα εκτέλεσης του Spark για να μπορεί να υποστηρίξει την αποδοτική εκτέλεση SQL ερωτημάτων [18]. Το Shark είναι πλήρως συμβατό με το Hive, καθώς τα ίδια ακριβώς ερωτήματα μπορούν να τρέξουν χωρίς καμία αλλαγή, ενώ επίσης έχει τη δυνατότητα να συνδεθεί στο metastore και κατά συνέπεια έχει πρόσβαση σε δεδομένα που προϋπάρχουν στο Hive.

2.5.2 Resilient Distributed Datasets

Το κύριο στοιχείο του Spark είναι τα Resilient Distributed Datasets (RDDs), τα οποία είναι αμετάβλητες συλλογές αντικειμένων που είναι μοιρασμένες σε όλους τους κόμβους ενός cluster. Βασική λειτουργία των RDDs είναι ότι μπορεί ο προγραμματιστής να εφαρμόσει διάφορους μετασχηματισμούς και από ένα RDD να παράγει ένα άλλο. Στη μνήμη τελικά αποθηκεύεται μόνο η ακολουθία των μετασχηματισμών που έχουν εφαρμοστεί σε ένα RDD και όχι τα ίδια τα δεδομένα που προκύπτουν. Η αποτίμηση των μετασχηματισμών γίνεται μόνο κατά την εκτέλεση μίας παράλληλης λειτουργίας (π.χ. flatMap), ενώ υπάρχει η δυνατότητα να οριστεί ότι το dataset είναι χρήσιμο να αποθηκευτεί στην cache μετά τον πρώτο υπολογισμό του για να μπορεί να χρησιμοποιηθεί με μεγαλύτερη ταχύτητα. Με αυτόν τον τρόπο είναι αρκετά εύκολο αν ένας κόμβος παρουσιάσει κάποιο πρόβλημα και σταματήσει να λειτουργεί να αποκαταστήσουμε τα χαμένα δεδομένα χρησιμοποιώντας τη σειρά μετασχηματισμών που είναι αποθηκευμένη για αυτό το RDD.

2.5.3 Εκτέλεση ερωτημάτων SQL over RDDs

Τα συστήματα όπως το Hive και το Shark χρησιμοποιούνται συχνά για να επεξεργαστούν νέα δεδομένα που δεν έχουν ακόμα φορτωθεί στο σύστημα αποθήκευσης. Αυτό αποκλείει τεχνικές στατικής βελτιστοποίησης ερωτημάτων που στηρίζονται σε a priori στατιστικά όπως ευρετήρια. Η έλλειψη στατιστικών για τα νέα δεδομένα αλλά και η διαδεδομένη χρήση UDFs για την εκτέλεση ερωτημάτων απαιτούν μια δυναμική τεχνική βελτιστοποίησης των ερωτημάτων. Για αυτό το σκοπό το Spark επεκτάθηκε ώστε να υποστηρίζει partial DAG execution , μία τεχνική που επιτρέπει δυναμική τροποποίηση του πλάνου εκτέλεσης του ερωτήματος βάσει στατιστικών που συλλέγονται κατά το run-time.

Ένα άλλο χαρακτηριστικό του Shark είναι η εκτέλεση in-memory υπολογισμών για να πετύχει μικρή καθυστέρηση στην απάντηση ερωτημάτων. Όμως χρησιμοποιώντας με απλοϊκό τρόπο την υπάρχουσα μνήμη του Spark μπορεί να μην έχει το επιθυμητό αποτέλεσμα. Γι΄ αυτό το Shark υλοποιεί μία τεχνική που λέγεται columnar memory

store. Ουσιαστικά, η αποθήκευση των δεδομένων γίνεται με βάση τις στήλες τις οποίες περιέχουν οι πίνακες. Σε αντίθεση με το Spark που αποθηκεύει τα partitions των δεδομένων ως συλλογές από JVM αντικείμενα, το Shark αποθηκεύει όλες τις στήλες των πρωταρχικών τύπων ως JVM arrays, το οποίο οδηγεί σε καλύτερες επιδόσεις [18].

Είναι συχνό φαινόμενο τα δεδομένα να αποθηκεύονται χρησιμοποιώντας κάποιο λογικό διαχωρισμό σε μία ή περισσότερες στήλες. Το Shark έχει τη δυνατότητα να εκμεταλλευτεί αυτό το διαχωρισμό, καθώς μπορεί να αποκλείσει τις περιοχές που δεν βρίσκονται στο εύρος τιμών που έχει προσδιορίσει το ερώτημα. Έτσι, εξοικονομείται πολύτιμος χρόνος, αφού δεν χρειάζεται να διαβάσει καθόλου τα δεδομένα που δεν είναι σχετικά με το ερώτημα. Η διαδικασία αυτή ονομάζεται map pruning.

2.6 Περιγραφή του Dataset

Τα δεδομένα που χρησιμοποιήθηκαν ως ένα use case σε αυτή τη διπλωματική προέρχονται από τον ελληνικό κόμβο διασύνδεσης GR-IX. Το GR-IX είναι ένα Internet Exchange Point, δηλαδή ένας κόμβος στον οποίο συνδέονται όλοι οι πάροχοι που δρουν στην ελληνική επικράτεια για να μπορούν να ανταλλάσουν μεταξύ τους δικτυακή κίνηση χωρίς να είναι απαραίτητο να την δρομολογούν μέσω τρίτων δικτύων. Ο κόμβος αυτός διαχειρίζεται από το ΕΔΕΤ το οποίο είναι υπεύθυνο για τη συντήρηση και την παροχή των υπηρεσιών του.

Όπως είναι φυσικό από τον κόμβο αυτό περνάει ένα πολύ μεγάλο ποσοστό της δικτυακής κίνησης και τα δεδομένα που προκύπτουν είναι της τάξης των πολλών terra bytes. Κάθε μέρα με τη βοήθεια του εργαλείου sFlow γίνεται ένα sampling των πακέτων IP που διέρχονται από αυτό τον κόμβο ανά τακτά χρονικά διαστήματα. Το εργαλείο αυτό αποθηκεύει τα πακέτα που συλλαμβάνει σε μία συγκεκριμένη μορφή, κρατώντας αρκετές πληροφορίες σχετικές με τον αποστολέα και τον παραλήπτη του πακέτου, αλλά και με το ίδιο το πακέτο (μέγεθος πακέτου, timestamp). Τα ανωνυμοποιημένα δεδομένα που χρησιμοποιήσαμε καλύπτουν μία χρονική περίοδο από τα τέλη Ιουλίου του 2013 έως τα μέσα Φεβρουαρίου του 2014. Το συνολικό μέγεθος των δεδομένων που ήταν διαθέσιμα ήταν περίπου 200GB σε συμπιεσμένη μορφή. Για τους σκοπούς της εργασίας αυτής, επειδή δεν μας ήταν απαραίτητα όλα τα πεδία που προσφέρει το sFlow, πραγματοποιήσαμε μία προεπεξεργασία των δεδομένων και τελικώς κρατήσαμε ορισμένα από αυτά που μας ήταν χρήσιμα. Αυτά συνοψίζονται ακολούθως:

$$ipFrom, intIPFrom, ipTo, intIPTo, protocol, srcPort, dstPort, ipSize, date$$

Το τελικό μέγεθος των δεδομένων μετά από την προεπεξεργασία αυτή είναι περίπου 50GB σε συμπιεσμένη μορφή. Σκοπός της ανάλυσής μας είναι να συνδυάσουμε την πληροφορία αυτή με επιπλέον σύνολα δεδομένων που περιέχουν meta-πληροφορίες όπως την κατάταξη της κάθε IP στο αυτόνομο σύστημα (AS) που ανήκει ή την αντιστοίχισή της με τη χώρα προέλευσης. Η πληροφορία αυτή συλλέχτηκε από τη βάση

δεδομένων GeoLite [2], μία geolocation βάση δεδομένων που χρησιμεύει για την αντιστοίχιση των διευθύνσεων IP στο αυτόνομο σύστημα που ανήκουν και στη χώρα προέλευσης τους. Τα μεγέθη αυτών των αρχείων είναι 12MB για το αρχείο που περιέχει τα αυτόνομα συστήματα και 7MB για το αρχείο με τις χώρες. Τα δεδομένα αυτά ανανεώνονται στις αρχές κάθε μήνα, οπότε μπορούμε να χρησιμοποιούμε διαφορετικά meta-data αναλόγως της ημερομηνίας που προέρχονται τα αρχικά δεδομένα μας. Τα αρχεία αυτά περιέχουν τα δεδομένα στη μορφή:

$$ipStart,\ ipStop,\ ASName/\{Country, CountryCode\}$$

Στον παραπάνω τύπο οι τιμές ipStart και ipStop έχουν αντιστοιχηθεί σε ένα μοναδικό ακέραιο αριθμό μέσω μιας συνάρτησης μετατροπής όπως περιγράφεται εδώ [2].

Εκτός από αυτή τη βάση δεδομένων, χρησιμοποιήθηκε και μία άλλη πηγή αποθετηρίου δεδομένων που συλλέγονται για ερευνητικούς σκοπούς ύστερα από προσπέλαση του δημόσιου διαδικτύου, το scans.io [4]. Αυτό χρησιμοποιήθηκε ως πληροφορία για την αντιστοίχιση της κάθε IP με το DNS όνομα του υπολογιστή στο οποίο βρίσκεται το interface. Για αυτό το λόγο, το μέγεθος αυτού του αρχείου είναι σχετικά μεγάλο, περίπου 57GB. Παρόμοια με τα άλλα σετ δεδομένων, έτσι και αυτό ανανεώνεται μία φορά το μήνα. Η μορφή των δεδομένων σε αυτό το αρχείο είναι:

$$IP,\ DNSName$$

Σε αυτή την περίπτωση οι IP που περιέχονται στο αρχείο βρίσκονται στην κλασική μορφή τους (dot-decimal notation).

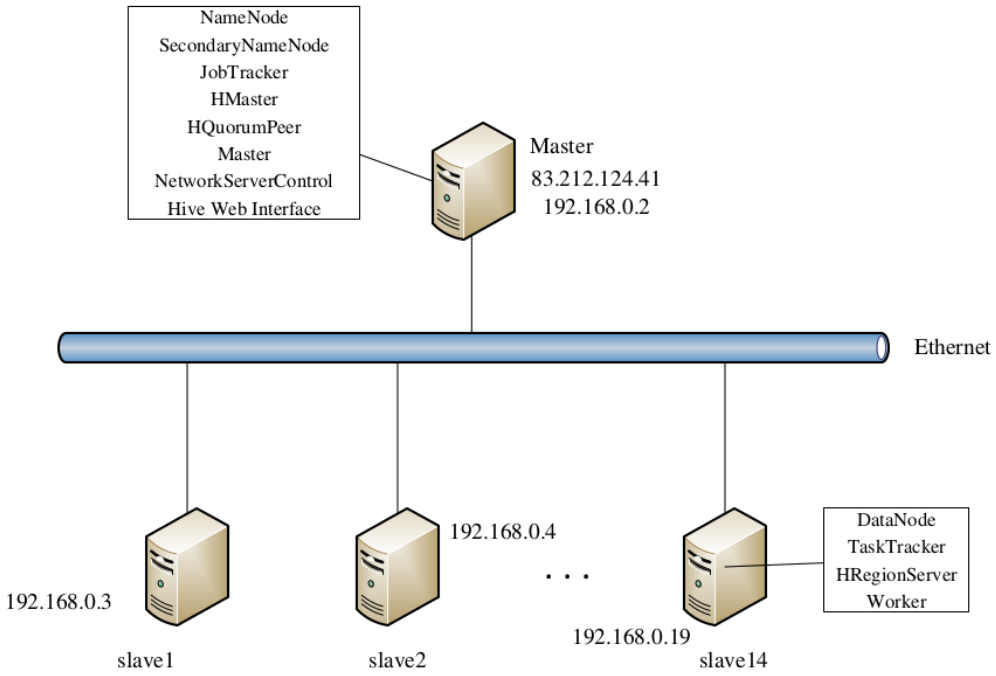
2.7 Περιγραφή του cluster

Για την εκτέλεση των πειραμάτων δημιουργήσαμε ένα cluster από υπολογιστές χρησιμοποιώντας την υποδομή που προσφέρει ο ~okeanos [17], ένα έργο που παρέχει υπηρεσίες Infrastructure as a Service (IaaS) στην Ελληνική ερευνητική και ακαδημαϊκή κοινότητα, που δημιουργήθηκε και συνεχίζει να αναπτύσσεται από το Ελληνικό Δίκτυο Έρευνας και Τεχνολογίας (ΕΔΕΤ). Αυτή η πλατφόρμα παρέχει στο χρήστη τη δυνατότητα για υπολογιστική ισχύ (μέσω εικονικών μηχανών), δικτύωσης και αποθήκευσης δεδομένων. Το cluster που δημιουργήσαμε αποτελείται από 1 master κόμβο και 14 slaves με τις εξής δυνατότητες:

Node	CPU	RAM	Disk
master	4 cores 2.1GHz	4GB	10GB
slave	4 cores 2.1GHz	8GB	60GB

Πίνακας 2.2: Χαρακτηριστικά κόμβων του cluster

Στο master κόμβο τρέχουν οι master διεργασίες όλων των tools και frameworks που χρησιμοποιούμε, δηλαδή ο JobTracker και ο NameNode του Hadoop και οι master της HBase και του Spark. Επίσης, σε αυτό τον κόμβο είναι εγκατεστημένο το Hive και το Shark. Αντίστοιχα, στους slaves τρέχουν οι υπόλοιπες διεργασίες που αφορούν τα εργαλεία αυτά, όπως ο Datanode, ο TaskTracker, ο RegionServer και ο Worker. Αυτά φαίνονται εποπτικά στο σχήμα 2.6.



Σχήμα 2.6: Διάγραμμα παράταξης cluster

Κεφάλαιο 3

Υλοποίηση map join με μικρό meta-dataset

3.1 Ερωτήματα που χρησιμοποιούν ένα μικρό meta-dataset

Ξεκινάμε την ανάλυση μας σε αυτό το κεφάλαιο από μία ομάδα ερωτημάτων που στηρίζονται σε ένα "μικρό" meta-dataset για να επιτύχουν τον επιθυμητό συνδυασμό πληροφορίας. Με την έννοια μικρό αρχείο, υπονοούμε αρχεία που δεν ξεπερνούν τα μερικά δεκάδες MB (τυπικά 10-20). Σε αυτή την κατηγορία ερωτημάτων εμπίπτουν ερωτήματα που αφορούν τη συσχέτιση των IP διευθύνσεων με τα αυτόνομα συστήματα που ανήκουν ή τις χώρες προέλευσής τους. Ενδεικτικά αναφέρουμε ότι θα ασχοληθούμε με ερωτήματα κατάταξης (αλλιώς top-K) εισερχόμενης κίνησης σε ένα αυτόνομο σύστημα, εξερχόμενης κίνησης, καθώς και κίνησης μεταξύ δύο διαφορετικών αυτόνομων συστημάτων. Αντίστοιχα ερωτήματα μπορούν να εκτελεστούν και για την περίπτωση των χωρών προέλευσης για κάθε IP. Η κίνηση μετριέται βάσει του αριθμού πακέτων με τα οποία σχετίζεται (δηλαδή έχει στείλει ή/και έχει λάβει) μία διεύθυνση IP. Για παράδειγμα, ένα από τα ερωτήματα που εξετάζουμε και έχοντας υπόψη το μοντέλο δεδομένων που συζητήθηκε στην ενότητα 1.2 είναι το παρακάτω:

```
SELECT t2.asName, t3.asName, COUNT(*) as total
FROM SFlows t1, ASNames t2, ASNames t3
WHERE t1.ipFrom = t2.ip and t1.ipTo = t3.ip
GROUP BY t2.asName, t3.asName
ORDER BY total DESC;
```

3.2 Υλοποίηση ερωτημάτων με το built-in Shuffle Join του Hive

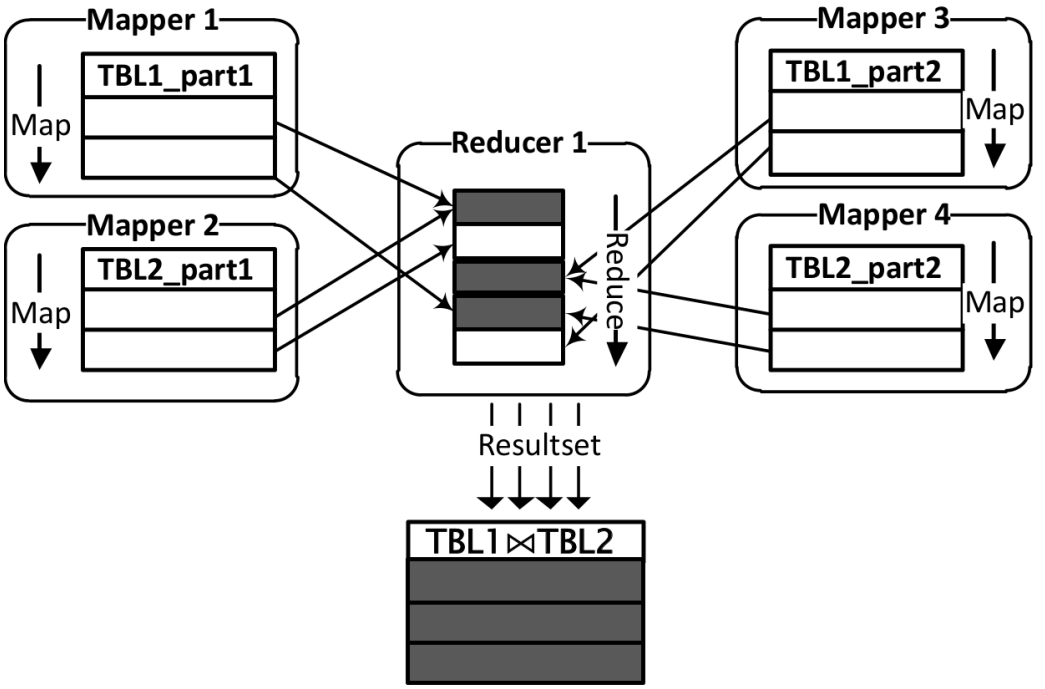
3.2.1 Επισκόπηση του Shuffle Join

Το shuffle join [15] είναι ένας αλγόριθμος που εφαρμόζει την πράξη της ένωσης δύο πινάκων στις σχεσιακές βάσεις δεδομένων. Σκοπός του αλγορίθμου είναι να βρει, για κάθε διαφορετική τιμή του χαρακτηριστικού βάσει του οποίου γίνεται η ένωση, το σύνολο των εγγραφών σε κάθε σχέση που έχει αυτή την τιμή. Το Hive εφαρμόζει μία εργασία Map Reduce για να υλοποιήσει αυτό τον αλγόριθμο, η λειτουργία της οποίας είναι η εξής. Η κάθε διεργασία map αναλαμβάνει να επεξεργαστεί ένα τμήμα ενός εκ των δύο πινάκων και στη συνέχεια φροντίζει να παράγει ως έξοδο για κάθε εγγραφή ένα ζεύγος key-value μαζί με ένα αναγνωριστικό που δείχνει τον πίνακα από τον οποίο προέρχεται η κάθε εγγραφή. Στη συνέχεια ο reducer αναλαμβάνει για κάθε κλειδί εισόδου να ελέγξει αν υπάρχουν τα αναγνωριστικά και των δύο πινάκων, οπότε στην περίπτωση αυτή η εγγραφή που εξετάζουμε είναι στις ζητούμενες που πρέπει να παραχθούν. Τα παραπάνω φαίνονται εποπτικά στο ψευδοκώδικα 1.

Ψευδοκώδικας 1 Υλοποίηση shuffle join του Hive

```
1:  function Map(k1, v1)
2:      for each table i do
3:          for every record in table i do
4:              emit(key, value) with tag i
5:  end function
6:
7:  function Reduce(key, list(values))
8:      for each key do
9:          if tag1 and tag2 are in list(values) then
10:             emit(key, list(values).toString())
11: end function
```

Ο τρόπος αυτός εφαρμογής της ένωσης των πινάκων δεν είναι αρκετά αποδοτικός, αφού χρειάζεται να συνδυάσει όλες τις δυνατές τιμές των δύο πινάκων στο συγκεκριμένο χαρακτηριστικό που εφαρμόζεται η ένωση και πρέπει να γίνει αντιγραφή όλων των δεδομένων εισόδου στη συνάρτηση Reduce, η οποία εφαρμόζει στη συνέχεια το join. Ακόμα χειρότερα, απαιτείται η μεταφορά μέσω του δικτύου και των δύο πινάκων ανάμεσα στους κόμβους που αναλαμβάνουν να τρέξουν την εργασία Map Reduce, γεγονός που μας υποβάλλει σε περιορισμούς που εξαρτώνται από το throughput του δικτύου με συνέπεια να επηρεάζεται αρνητικά η απόδοση της εφαρμογής. Το σχήμα 3.1 παρουσιάζει τη διαδικασία του Shuffle Join.



Σχήμα 3.1: Shuffle Join

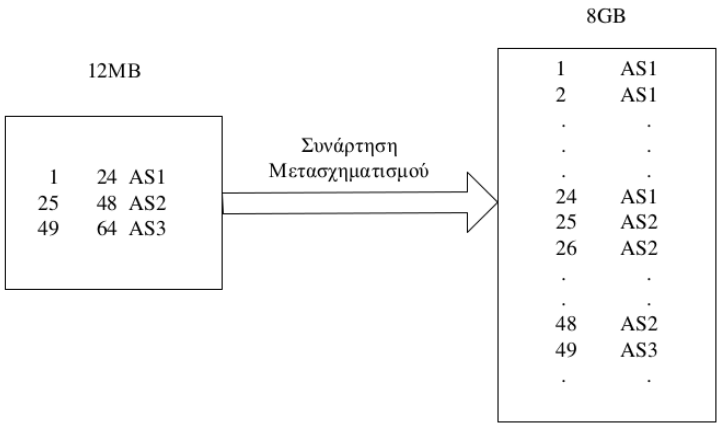
3.2.2 Εφαρμογή Shuffle Join

Η υλοποίηση του shuffle join είναι αρκετά ξεκάθαρη στην εφαρμογή της στα ερωτήματα που μας ενδιαφέρουν. Το σημείο στο οποίο πρέπει να δώσουμε προσοχή είναι ότι το αρχείο με τα meta-data έχει μόνο το εύρος των IP που ανήκουν σε ένα αυτόνομο σύστημα. Για να μπορέσουμε να εφαρμόσουμε το join στις στήλες με τις IP πηγής και προορισμού θα πρέπει να εκτελέσουμε μια προεπεξεργασία του αρχείου και για κάθε μία IP που υπάρχει στο εύρος τιμών που δίνεται να κατασκευάσουμε μία εγγραφή με το συγκεκριμένο όνομα του αυτόνομου συστήματος. Αυτό παρουσιάζεται στο σχήμα 3.2.

Η συνάρτηση μετασχηματισμού είναι αρκετά απλή στην υλοποίησή της και παρουσιάζεται στον ψευδοκώδικα 2.

Ψευδοκώδικας 2 Blow up του αρχείου με την αντιστοίχιση IP και AS η!

```
1: for every line in file do
2:     parts = line.split(",")
3:     for i = int(parts[0]); i < int(parts[1]); i++ do
4:         write(i, parts[2])
```



Σχήμα 3.2: Μετασχηματισμός του αρχείου IP-AS

Η επιλογή αυτή έχει ένα σημαντικό κόστος, την υπερβολική αύξηση του όγκου των δεδομένων μετά την επεξεργασία αυτή. Συγκεκριμένα, αρχικά το αρχείο έχει μέγεθος περίπου 12MB, ενώ μετά το μέγεθος αυξάνει σε περίπου 8GB σε συμπιεσμένη μορφή. Αφού έχει γίνει αυτή η επεξεργασία στη συνέχεια μπορούμε να εφαρμόσουμε ένα απλό HiveQL ερώτημα που επιτυγχάνει το ζητούμενο join.

```
SELECT first_join.asName, as2.asName, COUNT(*) as total
FROM (SELECT intipto, asName
      FROM sflows JOIN ases ON sflows.intipfrom = ases.ip)
       first_join
      JOIN ases as2 ON first_join.intipto = as2.ip
GROUP BY first_join.asName, as2.asName
ORDER BY total DESC;
```

Στην προκειμένη περίπτωση δίνουμε ένα παράδειγμα με τον κώδικα που χρησιμοποιήσαμε για την εκτέλεση του ερωτήματος που αφορά τα ζευγάρια αυτόνομων συστημάτων με την υψηλότερη μεταξύ τους κίνηση.

3.2.3 Πειραματικά αποτελέσματα της τεχνικής του Shuffle Join

Σε αυτή την ενότητα παρουσιάζουμε τα πειραματικά αποτελέσματα που εξάγαγαμε από την εκτέλεση του απλού shuffle join που επιτελεί το Hive στο ερώτημα που παρουσιάσαμε στην ακριβώς προηγούμενη ενότητα. Στον πίνακα 3.1 παρουσιάζονται συνοπτικά διάφορες παράμετροι από την εκτέλεσή του.

Job Phase	#Tasks	Bytes Read		Bytes Written		Elapsed Time	
		File	HDFS	File	HDFS	Total	Per Task
Map	5000	90GB	69GB	128GB	0	2h	1.5min
Reduce	161	57GB	0	57GB	4.6GB	50min	15min
Total	5161	147GB	69GB	185GB	4.6GB	2h 50min	

Πίνακας 3.1: Συγκεντρωτικά αποτελέσματα από την εκτέλεση του shuffle join στα δεδομένα IP-AS

Αξίζει να σημειώσουμε ότι το συγκεκριμένο ερώτημα απαιτεί 4 εργασίες Map Reduce για να ολοκληρωθεί και παραπάνω παρουσιάζονται αθροιστικά οι τιμές για όλες μαζί. Όπως παρατηρούμε το ερώτημα για να εκτελεστεί απαιτεί συνολικά περίπου 3 ώρες για να ολοκληρωθεί όταν χρησιμοποιεί πλήρως το cluster που έχει περιγραφεί στην ενότητα 2.7. Αρχικά, αναφέρουμε ότι το Hive γενικά δρομολογεί ανάλογο αριθμό διεργασιών map σχετικά με το μέγεθος του πίνακα που είναι αποθηκευμένος. Στη δική μας περίπτωση, επειδή τα αρχεία που αποτελούν τον πίνακα είναι συμπιεσμένα, κάθε mapper αναλαμβάνει ένα ολόκληρο αρχείο ώστε να μπορεί από την επικεφαλίδα να αναγνωρίσει τον αλγόριθμο συμπίεσης και το format των δεδομένων. Τα αρχεία αυτά είναι μικρότερα από το chunk size που χρησιμοποιεί το framework, οπότε ο περιορισμός αυτός δεν παραβιάζεται. Ο συνολικός αριθμός των διεργασιών map που δρομολογούνται ισούται με το άθροισμα του αριθμού των αρχείων που αποτελούνται οι δύο πίνακες που συμμετέχουν στο ερώτημα και είναι αρκετά μεγάλος όπως φαίνεται. Ακόμα, οι διεργασίες map και reduce αναλώνονται αρκετά σε διαδικασίες εισόδου-εξόδου δεδομένων (I/O procedures), καθώς διαβάζουν και γράφουν πολλά δεδομένα, γεγονός που έχει σαφή αντίκτυπο στην απόδοση του ερωτήματος. Τέλος, η κάθε διεργασία map διαρκεί κατά μέσο όρο περίπου 10 φορές λιγότερο από ό,τι η αντίστοιχη reduce γεγονός που μας επιβεβαιώνει ότι σε αυτή την περίπτωση το join πραγματοποιείται κατά τη φάση reduce και εκεί συνεπώς συσσωρεύεται ο περισσότερος φόρτος εργασίας.

3.3 Υλοποίηση ερωτημάτων με την τεχνική του Map Join

3.3.1 Επισκόπηση του Map Join

Μία αποδοτικότερη προσέγγιση για την πραγματοποίηση του join των δύο πινάκων στις περιπτώσεις που ο ένας είναι πολύ μικρότερος από τον άλλο είναι να χρησιμοποιήσουμε μια τεχνική που ονομάζεται map join[10]. Η τεχνική αυτή χρησιμοποιείται στη γενική περίπτωση εφαρμογών log processing (όπως αυτή που εξετάζουμε και εμείς), όπου ο ένας πίνακας που περιέχει τα log αρχεία είναι πολύ μεγαλύτερος από τον πίνακα αναφοράς. Στη συνέχεια αναλύουμε τον τρόπο λειτουργίας του map join και θα χρησιμοποιούμε το σύμβολο L για το μεγάλο πίνακα, καθώς και το σύμβολο S για

το μικρό.

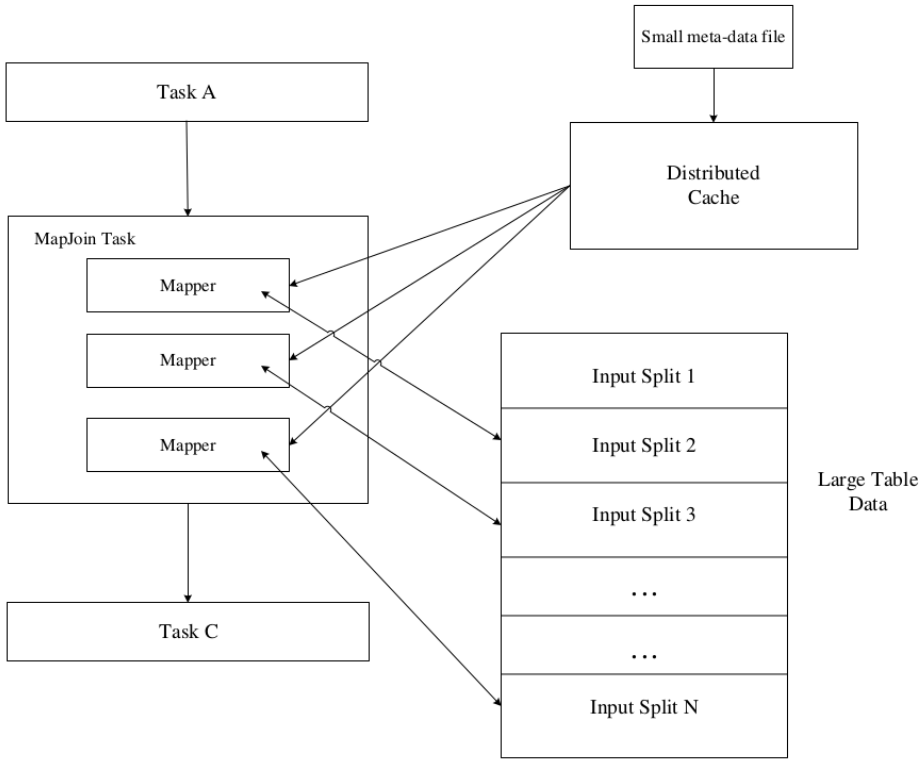
Το κίνητρο για τη μέθοδο αυτή είναι το γεγονός πως αν χρησιμοποιήσουμε την κύρια μνήμη για να αποθηκεύσουμε τον πίνακα S σε κάθε διεργασία mapper που θα ενεργοποιήσει το σύστημα, οι υπολογισμοί που απαιτούνται για την παραγωγή του αποτελέσματος θα μειωθούν σημαντικά και η εκτέλεση θα επιταχυνθεί σε μεγάλο βαθμό. Με αυτό τον τρόπο αποφεύγουμε τη μεταφορά και των δύο πινάκων μέσω του δικτύου, αφού ο S θα φορτώνεται σε κάθε διεργασία map και ταυτόχρονα όλη η επεξεργασία μπορεί να γίνεται, καθώς διαβάζεται κάθε εγγραφή του L. Συγκεκριμένα, η κάθε διεργασία map αναλαμβάνει ένα κομμάτι από τα δεδομένα του L (input split), ενώ παράλληλα φορτώνεται στη μνήμη της κατά την αρχικοποίηση ο πίνακας S [15]. Επίσης, χρησιμοποιείται μία δομή που ευνοεί την αναζήτηση βάσει ενός μοναδικού κλειδιού, όπως για παράδειγμα ένα HashTable. Το HashTable χρησιμοποιείται για να αποθηκεύσει τα δεδομένα του S , ενώ διαβάζοντας μία μία τις εγγραφές από το input split του L που ανέλαβε η διεργασία, ψάχνουμε στο HashTable για να βρούμε την αντίστοιχη τιμή. Η διαδικασία αυτή είναι αρκετά αποδοτική αν σκεφτούμε ότι το κόστος αναζήτησης σε ένα HashTable είναι κατά κανόνα σταθερό, δηλαδή $O(1)$. Αν υπάρχει η αντίστοιχη τιμή στη βοηθητική δομή, αυτή η εγγραφή παράγεται στο αποτέλεσμα, ειδάλλως αγνοείται. Το σχήμα 3.3 παρουσιάζει εποπτικά τη διαδικασία εκτέλεσης του map join.

3.3.2 Χρήση UDF στο Hive

Το Hive παρέχει ένα πολύ βολικό plugin που επιτρέπει στον προγραμματιστή να ορίσει τις δικές του συναρτήσεις πολύ εύκολα, τις οποίες μπορεί στη συνέχεια να συμπεριλάβει στις προκαθορισμένες συναρτήσεις που παρέχει το εργαλείο και να τις ενσωματώσει σε ερωτήματα που επεξεργάζονται δεδομένα. Η ελευθερία που προσφέρεται είναι εξαιρετικά μεγάλη, αφού σε αυτού του είδους τις συναρτήσεις μπορούμε να συμπεριλάβουμε οποιοδήποτε κώδικα, από απλές ενέργειες στη Java μέχρι διάβασμα αρχείων από το HDFS και σύνδεση με την HBase και επεξεργασία των προερχόμενων από εκεί δεδομένων.

Η διαδικασία δημιουργίας μίας συνάρτησης UDF είναι πολύ εύκολη. Ουσιαστικά, πρέπει να ορίσουμε μια κλάση που να επεκτείνει τη βασική κλάση UDF και να ορίσουμε τη μέθοδο evaluate ώστε να εκτελεί τις απαιτούμενες ενέργειες. Αυτή είναι και η μόνη απαίτηση που πρέπει να ικανοποιήσουμε. Κατά τα άλλα ο κώδικάς μας μπορεί να περιέχει ή/και να χρησιμοποιεί οποιαδήποτε άλλη βιβλιοθήκη κώδικα Java. Στη συνέχεια παραθέτουμε ένα απλό παράδειγμα ορισμού ενός UDF.

```
public final class Lower extends UDF {
  public Text evaluate(final Text s) {
    if (s == null) { return null; }
    return new Text(s.toString().toLowerCase());
  }
```



Σχήμα 3.3: Διάγραμμα Map Join

}

Αφού έχουμε ορίσει τη συνάρτηση που επιθυμούμε, στη συνέχεια πρέπει να την ενσωματώσουμε στις συναρτήσεις που αναγνωρίζει και μπορεί να χρησιμοποιήσει το Hive. Αυτό γίνεται πολύ εύκολα σε τρία βήματα:

1. Δημιουργούμε ένα jar αρχείο με την κλάση που φτιάξαμε
2. Προσθέτουμε το jar αρχείο στο classpath του Hive με την εντολή:

   add jar $/path/to/jar/file$

3. Ορίζουμε τη συνάρτηση που θέλουμε με την ακόλουθη εντολή:

   create temporary function $functionName$**as**$'nameOfTheClassInJarFile'$

Αξίζει να σημειωθεί ότι η μέθοδος evaluate δεν ορίζεται από κάποια διαπροσωπεία στη Java, καθώς μπορεί να δεχτεί ένα αυθαίρετο αριθμό ορισμάτων και μπορεί να επιστρέψει επίσης έναν αυθαίρετο τύπο αποτελέσματος. Όπως είναι αναμενόμενο, είναι

ακόμη δυνατό να ορίσουμε πολλές παραλλαγές της μεθόδου evaluate με διαφορετικό αριθμό ή/και τύπο ορισμάτων (method overloading). Το σύστημα εκτέλεσης είναι σε θέση να επιλέξει την κατάλληλη μέθοδο evaluate εξετάζοντας τον αριθμό και τον τύπο των ορισμάτων της Hive συνάρτησης που κλήθηκε. Η συνάρτηση αυτή μπορεί να κληθεί σε οποιοδήποτε σημείο του ερωτήματος, ουσιαστικά μπορεί να χρησιμοποιηθεί ακριβώς όπως μία κανονική στήλη του πίνακα. Για παράδειγμα, είναι δυνατό να τη χρησιμοποιήσουμε ως μέρος του αποτελέσματος που θα εξάγουμε ή να χρησιμοποιήσουμε το αποτέλεσμα της συνάρτησης για να εφαρμόσουμε μία ενέργεια Group by. Τέλος, δίνουμε ένα παράδειγμα χρήσης μίας συνάρτησης UDF.

```
SELECT my_lower(title), sum(freq)
FROM titles group by my_lower(title)
```

3.3.3 Εφαρμογή Map Join

Προχωράμε τώρα στην περιγραφή της υλοποίησης του map join στην περίπτωση που έχουμε να χρησιμοποιήσουμε ένα μικρό dataset για να συνδυάσουμε με τα δεδομένα από τον κόμβο διασύνδεσης. Σκοπός μας είναι να σχεδιάσουμε μια συνάρτηση η οποία θα δέχεται ως όρισμα το αρχείο με τη meta-πληροφορία και μία διεύθυνση IP και σαν αποτέλεσμα θα επιστρέφει το αυτόνομο σύστημα στο οποίο ανήκει ή τη χώρα προέλευσης. Στην παρακάτω περιγραφή θεωρούμε ότι η πληροφορία που θέλουμε να εξάγουμε είναι το αυτόνομο σύστημα που ανήκει η διεύθυνση.

Ο σχεδιασμός της συνάρτησης είναι σχετικά απλός, καθώς θα ακολουθήσουμε ακριβώς τη λογική που περιγράφηκε παραπάνω για την εκτέλεση map join στη γενική περίπτωση. Πρώτα, κατά την αρχικοποίηση του mapper που αναλαμβάνει ένα κομμάτι εισόδου να επεξεργαστεί φροντίζουμε να φορτώσουμε στην κύρια μνήμη το αρχείο με την αντιστοίχιση των διευθύνσεων IP και αυτόνομων συστημάτων. Υπενθυμίζουμε ότι το αρχείο αυτό έχει τη μορφή:

$$ipStart,\ ipStop,\ ASName$$

Σημειώνουμε ότι με τη συγκεκριμένη μέθοδο δεν είναι απαραίτητο να επαυξήσουμε την πληροφορία που περιέχεται στο αρχείο. Τοποθετούμε, λοιπόν, σε μία δομή που ονομάζεται TreeMap [6] δύο εγγραφές για κάθε αυτόνομο σύστημα, μία για την πρώτη διεύθυνση που ανήκει σε αυτό καθώς και μία για την τελευταία διεύθυνση. Η δομή αυτή αποτελεί μια υλοποίηση βασισμένη στα Red-Black trees [8] που είναι μια παραλλαγή ενός ισοσταθμισμένου δυαδικού δέντρου αναζήτησης. Για κάθε διεύθυνση IP που θέλουμε να επεξεργαστούμε μπορούμε σε χρόνο $O(logn)$ να αναζητήσουμε την τιμή της στο δέντρο αυτό και να επιστρέψουμε το ζητούμενο αποτέλεσμα. Συγκεκριμένα, βρίσκουμε την πρώτη διεύθυνση που είναι μεγαλύτερη ή ίση από τη ζητούμενη και αν αυτή αντιστοιχεί στην τελική διεύθυνση από το εύρος διευθύνσεων ενός αυτόνομου συστήματος, τότε αυτή ανήκει σε αυτό το αυτόνομο σύστημα, αλλιώς αν η

Ψευδοκώδικας 3 Συνάρτηση GetAS

```
 1: TreeMap<Long, String> ASMap = NULL
 2: function EVALUATE(String ip, String mapFile)
 3:     if ASMap == NULL then
 4:         while (line = mapFile.readLine()) != NULL do
 5:             fields = line.split(",")
 6:             ASMap.put(fields[0], fields[2] + "_start")
 7:             ASMap.put(fields[1], fields[2] + "_stop")
 8:     ipParts = ip.split("\\.")
 9:     if ipParts.length > 1 then
10:         intIP = ipParts[0] * 16777216 + ipParts[1] * 65536
11:                 + ipParts[2] * 256 + ipParts[3]
12:         key = ASMap.ceilingKey(intIP)
13:         if key != NULL then
14:             if key == intIP then
15:                 return ASMap.get(key).split("_")[0]
16:             else
17:                 part = ASMap.get(key).split("_")
18:                 if part[1].equals("stop") then
19:                     return part[0]
20:     return NULL
21: end function
```

διεύθυνση που μας επιστρέφεται είναι η αρχική ενός αυτόνομου συστήματος σημαίνει ότι για τη ζητούμε IP δεν υπάρχει η καταγραφή της σε κάποιο αυτόνομο σύστημα. Ο ψευδοκώδικας 3 σκιαγραφεί την υλοποίηση που προτείνουμε.

Αφού έχουμε δημιουργήσει αυτή τη συνάρτηση και την έχουμε προσθέσει σε αυτές που μπορεί να κατανοήσει το Hive , όπως περιγράφηκε στην ενότητα 3.3.2, στη συνέχεια είναι αρκετά εύκολο να τη χρησιμοποιήσουμε για να εκτελέσουμε το ίδιο ερώτημα με αυτό της ενότητας 3.2.2. Παραθέτουμε ακολούθως τον κώδικα για αυτή την περίπτωση:

```
INSERT OVERWRITE LOCAL DIRECTORY 'outputs/topASPairs_full'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
SELECT getas(ipfrom, './GeoIPASNum2.csv'),
       getas(ipto, './GeoIPASNum2.csv'), COUNT(*) AS pairs
FROM sflows
GROUP BY getas(ipfrom, './GeoIPASNum2.csv'),
         getas(ipto, './GeoIPASNum2.csv')
ORDER BY pairs DESC;
```

3.3.4 Πειραματικά αποτελέσματα της τεχνικής του Map Join

Στην ενότητα αυτή παρουσιάζουμε τα πειραματικά αποτελέσματα από την εκτέλεση του map join όπως εφαρμόστηκε στην περίπτωση για την καταγραφή της κίνησης μεταξύ των αυτόνομων συστημάτων. Ο πίνακας 3.2 παρουσιάζει συνοπτικά ό,τι προέκυψε:

Job Phase	# Tasks	Bytes Read		Bytes Written		Elapsed Time	
		File	HDFS	File	HDFS	Total	Per Task
Map	4701	0GB	50GB	1GB	0	42min	40sec
Reduce	99	16MB	0	30MB	1MB	10min	4min
Total	4800	16MB	50GB	1.03GB	1GB	52min	

Πίνακας 3.2: Συγκεντρωτικά αποτελέσματα από την εκτέλεση του map join

Από τα παραπάνω φαίνεται καθαρά γιατί αυτός ο τρόπος υπερτερεί από το απλό shuffle join που εκτελεί το Hive. Αρχικά, δεν χρειάζεται να κάνουμε blow-up του αρχείου με την αντιστοίχιση των IP σε αυτόνομα συστήματα, γεγονός που μας εξοικονομεί τόσο χώρο (το αρχείο είναι 12MB αλλά αν το επαυξήσουμε γίνεται περίπου 65GB) όσο και αριθμό διεργασιών map που πρέπει να εκτελεστούν, αφού πλέον χρειάζεται να έχουμε φορτώσει μόνο τον πίνακα με τα δεδομένα από το GR-IX. Η μεγάλη διαφορά φαίνεται στον όγκο δεδομένων εισόδου-εξόδου που αναλώνονται από τις map και reduce διεργασίες. Τα νούμερα στην περίπτωση αυτή είναι αρκετά μικρότερα από την προηγούμενη, σε κάποιες περιπτώσεις δε έχουν έως και τάξεις μεγέθους διαφορά. Συνέπεια αυτού είναι ο συνολικός χρόνος εκτέλεσης να μειώνεται δραστικά σε περίπου 52 λεπτά. Όπως παρατηρούμε η κάθε διεργασία map ολοκληρώνεται στον μισό χρόνο σε σχέση με πριν, ενώ μεγάλη διαφορά έχουμε και στις reduce διεργασίες που ολοκληρώνονται 4 φορές περίπου γρηγορότερα. Πλέον, δηλαδή το join πραγματοποιείται στη map φάση της εργασίας και επειδή οι πράξεις γίνονται in-memory κερδίζουμε σημαντικά σε απόδοση. Ένα τελευταίο στοιχείο που αξίζει να σημειωθεί είναι ότι πλέον απαιτούνται μόνο 2 Map-Reduce εργασίες για να παραχθεί το ζητούμενο αποτέλεσμα, γεγονός που μας επισημαίνει τη χρησιμότητα των UDFs για να μπορέσουμε να ενισχύσουμε τη λειτουργικότητα και αποτελεσματικότητα των ερωτημάτων που έχουμε τη δυνατότητα να εκτελέσουμε.

Κεφάλαιο 4

Υλοποίηση και τεχνικές βελτιστοποίσης map join με μεγάλο meta-dataset

Σε αυτό το κεφάλαιο στρέφουμε την προσοχή μας στην επεξεργασία ερωτημάτων που απαιτούν τη χρήση ενός μεγάλου meta-dataset για να καταφέρουμε να συνδυάσουμε τα δύο είδη πληροφοριών που θα μας επιτρέψουν να παρουσιάσουμε τα τελικά αποτελέσματα. Ένα τέτοιο σύνολο δεδομένων είναι η πληροφορία για την αντιστοίχιση των διευθύνσεων IP στα DNS ονόματα των υπολογιστών που ανήκουν. Το αρχείο αυτό είναι αρκετά μεγάλο (περίπου 57GB) συγκρινόμενο σε τάξη μεγέθους με το ίδιο το σύνολο δεδομένων από το GR-IX που είναι περίπου 50GBσε συμπιεσμένη μορφή. Σε αυτή την περίπτωση η εφαρμογή της τεχνικής του map join δεν είναι άμεσα εφαρμόσιμη, καθώς απαιτείται μία αρχική προεπεξεργασία των δεδομένων που θα μας επιτρέψει την κατάτμηση των δεδομένων με κατάλληλο τρόπο, ώστε η τελική επεξεργασία να γίνεται αποδοτικά. Τα ερωτήματα που εξετάζουμε σε αυτή την ενότητα είναι αντίστοιχα σε μορφή με αυτά του κεφαλαίου 3, με μόνη διαφορά ότι στηρίζονται σε ένα μεγάλο δευτερεύον σύνολο δεδομένων. Το ερώτημα που ακολουθεί ξεκαθαρίζει τα παραπάνω:

```
SELECT t2.dnsName, t3.dnsName, COUNT(*) as total
FROM SFlows t1, DNSNames t2, DNSNames t3
WHERE t1.ipFrom = t2.ip and t1.ipTo = t3.ip
GROUP BY t2.dnsName, t3.dnsName
ORDER BY total DESC;
```

4.1 Αρχική υλοποίηση ερωτημάτων με το built-in Shuffle Join

Η υλοποίηση του shuffle join είναι και σε αυτή την περίπτωση αρκετά εύκολη και σχετικά προφανής. Το πρώτο που πρέπει να φροντίσουμε είναι το αρχείο με τα δεδομένα να αποθηκευτεί σε ένα πίνακα στο Hive που θα έχει δύο στήλες δεδομένων, μία με τη διεύθυνση IP και μία με το DNS όνομα που αντιστοιχεί αυτή η διεύθυνση. Αξίζει να σημειώσουμε ότι το αρχείο αυτό περιέχει τα δεδομένα που έχουν προκύψει από ενεργές σαρώσεις του δημόσιου διαδικτύου, συνεπώς δεν περιέχουν όλες τις πιθανές διευθύνσεις με τα ονόματά τους, αλλά όσες συμμετείχαν σε ανταλλαγή δεδομένων. Ο κώδικας που χρησιμοποιήθηκε για την εκτέλεση αυτού του ερωτήματος είναι αρκετά όμοιος με τον αντίστοιχο της ενότητας 3.2.2.

```
SELECT first_join.dnsname, dns2.dnsname, COUNT(*) AS total
FROM (SELECT intipto, dnsname
      FROM sflows JOIN dnsnames ON sflows.intipfrom = dnsnames.ip)
       first_join
     JOIN dnsnames dns2 ON first_join.intipto = dns2.ip
GROUP BY first_join.dnsname, dns2.dnsname
ORDER BY total DESC;
```

Στον πίνακα 4.1 παρουσιάζουμε τα αποτελέσματα από την εκτέλεση αυτού του ερωτήματος στο cluster που έχουμε στήσει:

Job Phase	# Tasks	Bytes Read		Bytes Written		Elapsed Time	
		File	HDFS	File	HDFS	Total	Per Task
Map	5014	86GB	73GB	131GB	0	1h 49min	1.3min
Reduce	171	44GB	0	55GB	13GB	1h	20.4min
Total	5185	130GB	73GB	186GB	13GB	2h 15min	

Πίνακας 4.1: Συγκεντρωτικά αποτελέσματα από την εκτέλεση του shuffle join στα δεδομένα IP-DNS

Το συγκεκριμένο ερώτημα απαιτεί και αυτό 4 εργασίες Map Reduce για να εκτελεστεί σε πλήρη αντιστοιχία με το άλλο ερώτημα που αφορά τα αυτόνομα συστήματα. Αυτό που πρέπει να σημειώσουμε είναι ότι επειδή ο συνολικός αριθμός των διαφορετικών ζευγαριών με ονόματα υπολογιστών που προκύπτει είναι υπερβολικά μεγάλος (σε σχέση με τα διαφορετικά δυνατά ζευγάρια αυτόνομων συστημάτων), στις παραπάνω μετρήσεις έχουμε παραλείψει την τελευταία εργασία που εκτελεί την ταξινόμηση, αφού σε αυτή υπάρχει μία μόνο διεργασία reduce, η οποία αναλαμβάνει να επεξεργαστεί όλο το αποτέλεσμα με συνέπεια να υπάρχει μεγάλη καθυστέρηση που δεν είναι αντιπροσωπευτική του συνολικού χρόνου εκτέλεσης. Το ίδιο εφαρμόζουμε και στις επόμενες μεθόδους με τις οποίες πειραματιζόμαστε στην εκτέλεση αυτού του ερωτήματος. Στη συγκεκριμένη περίπτωση παρατηρούμε ότι ο όγκος δεδομένων που ανταλλάσσεται είναι

πάλι αρκετά μεγάλος και στη φάση του map αλλά και στη φάση του reduce γεγονός που επηρεάζει ανασταλτικά το συνολικό χρόνο εκτέλεσης που φτάνει τις 2 ώρες και 15 λεπτά.

4.2 Εφαρμογή της τεχνικής Map Join

Όπως έχουμε ήδη εξηγήσει, η τεχνική του map join στηρίζεται στο γεγονός ότι το ένα από τα δύο σύνολα δεδομένων που θέλουμε να συνδυάσουμε είναι αρκετά μικρότερο από το άλλο, οπότε μπορούμε να εκμεταλλευτούμε αυτό το γεγονός φορτώνοντας στη μνήμη στη διαδικασία αρχικοποίησης των διεργασιών το μικρό σύνολο δεδομένων και στη συνέχεια κατά την επεξεργασία της κάθε εγγραφής του μεγάλου πίνακα να επιτελείται η διαδικασία της ένωσης των πινάκων. Στην περίπτωση που εξετάζουμε τώρα αυτό δεν είναι άμεσα εφαρμόσιμο, καθώς και τα δύο σύνολα δεδομένων είναι αρκετά μεγάλα για να μπορούν να χωρέσουν στην κύρια μνήμη των διεργασιών.

Το πρόβλημα αυτό μπορεί να αντιμετωπιστεί αν αναλογιστούμε τα εξής. Για αρχή, είναι δεδομένο ότι το κάθε αρχείο που έχουμε συλλέξει με το εργαλείο sFlow περιέχει ένα περιορισμένο αριθμό από διευθύνσεις IP, συνεπώς αντίστοιχα απαιτείται ένας περιορισμένος αριθμός από εγγραφές από το αρχείο με τα ονόματα των υπολογιστών για να γίνει ο συνδυασμός των δεδομένων. Αυτό μας δίνει την ιδέα ότι αν μπορούμε να γνωρίζουμε ποιες διευθύνσεις IP υπάρχουν σε κάθε αρχείο, τότε μπορούμε να φορτώσουμε στη μνήμη μόνο το τμήμα του άλλου αρχείου που είναι σχετικό με αυτές και να εκτελεστεί έτσι το map join. Αυτό που πρέπει να εξασφαλίσουμε συνπεπώς είναι ότι το εύρος των πιθανών IP που υπάρχουν σε κάθε αρχείο θα είναι σχετικά περιορισμένο, ώστε τελικώς το κομμάτι που θα χρειαστεί να φορτώσουμε από το αρχείο με τα ονόματα DNS να μπορεί να χωρέσει στην κύρια μνήμη. Η αρχική κατάτμηση των αρχείων δεν μας εξασφαλίζει ότι αυτή η συνθήκη ικανοποιείται, οπότε θα πρέπει να εφαρμόσουμε μία προεπεξεργασία των δεδομένων, η οποία θα κατανέμει τις εγγραφές σε καινούρια αρχεία που αφενός θα έχουν παρόμοιο συνολικό αριθμό εγγραφών (ώστε όλες οι διεργασίες να αναλαμβάνουν παρόμοιο αριθμό δεδομένων) και αφετέρου θα περιέχουν ένα περιορισμένο εύρος διευθύνσεων, ώστε να μπορέσει να εφαρμοστεί και σε αυτή την περίπτωση η τεχνική του map join.

Αυτή η κατάτμηση των δεδομένων μπορεί να εφαρμοστεί με διάφορους τρόπους καθένας από τους οποίους έχει κάποια πλεονεκτήματα και μειονεκτήματα. Στη συνέχεια του κεφαλαίου εστιάζουμε σε μερικά από αυτά τα πιθανά εναλλακτικά σενάρια εφαρμογής της κατάτμησης των δεδομένων και αναλύουμε τις επιδόσεις και τα τεχνικά χαρακτηριστικά της κάθε μεθόδου.

4.3 Στατικός διαχωρισμός των δεδομένων βάσει των διευθύνσεων IP

4.3.1 Εισαγωγή

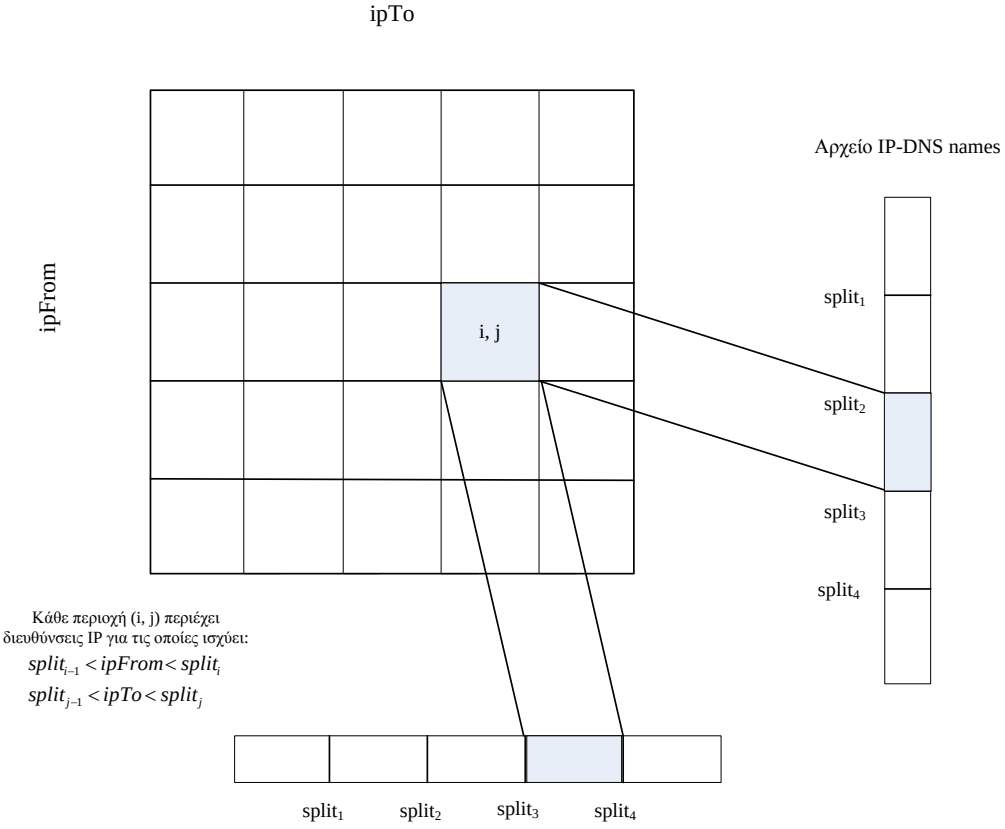
Η πρώτη προσέγγιση που θέσαμε σε εφαρμογή είναι να διαχωρίσουμε στατικά τα δεδομένα βάσει των διευθύνσεων IP που περιέχονται σε αυτά και στο αρχείο με τα ονόματα των υπολογιστών. Αυτό που πρέπει να επισημάνουμε είναι ότι τα αρχεία που προέρχονται από το GR-IX περιέχουν δύο διευθύνσεις IP, τη διεύθυνση προέλευσης και τη διεύθυνση προορισμού. Σε αντίθεση το αρχείο με τα ονόματα υπολογιστών περιέχει μόνο μία IP. Αυτό σημαίνει ότι θα πρέπει να βρούμε ένα διαχωρισμό των διευθύνσεων από το αρχείο με τα ονόματα των υπολογιστών και στη συνέχεια βάσει αυτού του διαχωρισμού να εφαρμόσουμε μία κατάτμηση των δεδομένων σε δύο διαστάσεις, όπου στη μία διάσταση θα βρίσκεται η τιμή της διεύθυνσης προέλευσης και στην άλλη η διεύθυνση προορισμού.

Αυτό που πρέπει να προσέξουμε είναι ότι τα κομμάτια του αρχείου με τα DNS ονόματα που θα προκύψουν θα πρέπει να μπορούν να χωρέσουν στην κύρια μνήμη της διεργασίας που θα αναλάβει την εκτέλεση ενός συγκεκριμένου αρχείου (συγκεκριμένα πρέπει να μπορούν να χωράνε δύο αρχεία, ένα για τις διευθύνσεις πηγής και ένα για τις διευθύνσεις προορισμού). Για να εξασφαλιστεί αυτό και δεδομένου του μεγέθους του αρχείου με τα ονόματα των υπολογιστών, βρίσκουμε ότι πρέπει να χωριστεί σε 148 κομμάτια. Χρησιμοποιώντας αυτά τα κομμάτια ως σημείο αναφοράς χωρίζουμε αντίστοιχα και τα sFlow δεδομένα σε αρχεία που γνωρίζουμε το εύρος διευθύνσεων IP που περιέχουν, τόσο τις διευθύνσεις προέλευσης όσο και τις αντίστοιχες προορισμού. Τα παραπάνω φαίνονται παραστατικά και στο σχήμα 4.1.

Για την αποθήκευση του meta-dataset χρησιμοποιούμε την HBase που προσφέρει τη δυνατότητα αποθήκευσης των δεδομένων με ταξινομημένο τρόπο ολικά ως προς τις διευθύνσεις IP. Αυτό μας επιτρέπει να εφαρμόσουμε αποδοτικά τη μεταφορά των meta-δεδομένων που απαιτούνται όταν γνωρίζουμε το εύρος των διευθύνσεων που μας ενδιαφέρουν. Αν το αρχείο ήταν αποθηκευμένο απλώς στο HDFS τότε για να φέρουμε στη μνήμη ένα συγκεκριμένο τμήμα του, θα έπρεπε να κάνουμε σειριακή ανάγνωση μέχρι το σημείο εκείνο και στη συνέχεια να φέρουμε το επιθυμητό κομμάτι δεδομένων, γεγονός που θα είχε σοβαρό αντίκτυπο στην απόδοση των ερωτημάτων.

4.3.2 Υλοποίηση της μεθόδου του στατικού διαχωρισμού

Το πρώτο βήμα για την υλοποίηση αυτής της μεθόδου είναι η φόρτωση του αρχείου με τα ονόματα των υπολογιστών στην HBase. Αφού έχουμε βρει τις τιμές των IP που θα χρησιμοποιηθούν ως σημεία διαχωρισμού, στη συνέχεια χρησιμοποιώντας τη μέθοδο του bulk loading [3] που προσφέρει η HBase φορτώνουμε τα δεδομένα σε κα-



Σχήμα 4.1: Στατικός διαχωρισμός των αρχείων βάσει των διευθύνσεων IP που περιέχουν

τάλληλο αριθμό από regions που έχουμε δημιουργήσει εκ των προτέρων. Το σύστημα εκτέλεσης της HBase αναλαμβάνει να φορτώσει αποδοτικά τα δεδομένα στον πίνακα διατηρώντας την αύξουσα σειρά στις διευθύνσεις IP.

Ακολούθως, πρέπει να ορίσουμε την εργασία Map Reduce που θα διαχωρίσει τα sFlow δεδομένα στα κατάλληλα αρχεία. Επίσης, η εργασία αυτή θα φροντίσει να δημιουργήσει και κάποια επιπλέον αρχεία που θα περιέχουν τη βοηθητική πληροφορία των μοναδικών IP που υπάρχουν σε κάθε αρχείο. Αυτό χρειάζεται, καθώς στο εύρος των διευθύνσεων που υπάρχουν στο αρχείο δεν είναι απαραίτητο να βρίσκονται στα πραγματικά δεδομένα όλες οι μοναδικές IP διευθύνσεις ανάμεσα στην αρχική και την

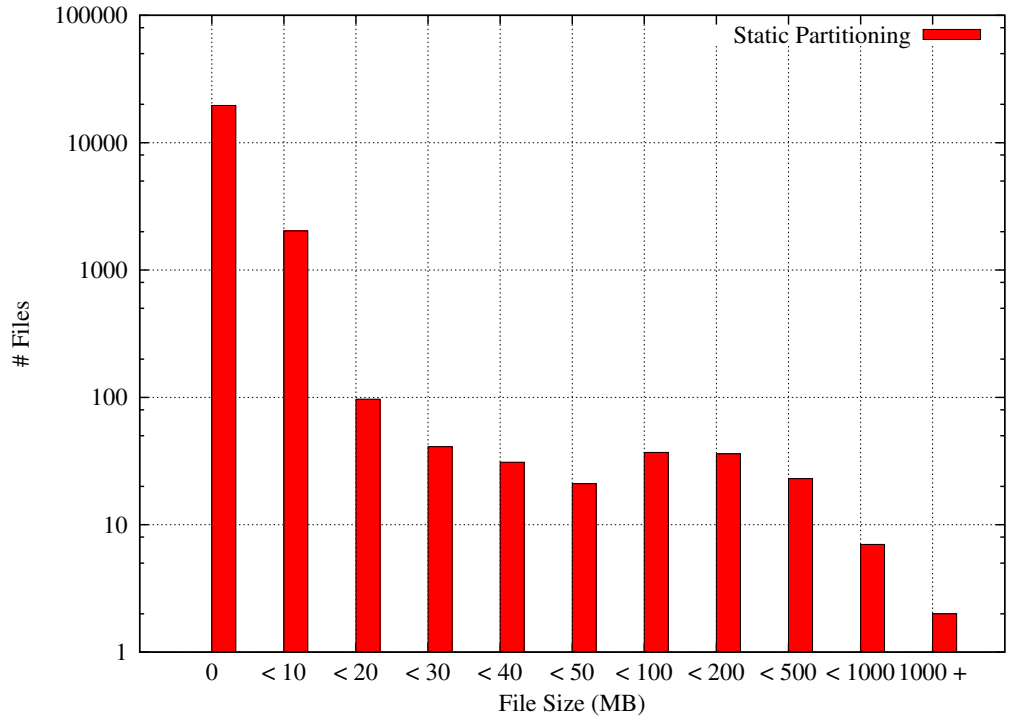
Ψευδοκώδικας 4 Στατικός διαχωρισμός των δεδομένων

```
1: function Map
2:     function setup
3:         read PartitionFile in TreeMap IPMap
4:     end function
5:     for every record r in file do
6:         parts = r.split(" ")
7:         ipFrom = parts[0]
8:         ipTo = parts[2]
9:         min = min(ipFrom, ipTo)
10:        max = max(ipFrom, ipTo)
11:        keyFrom = findIdFrom(IPMap)
12:        keyTo = findIdFrom(IPMap)
13:        partNum = keyFrom * partition_no + keyTo
14:        write(partNum, r + " " + partNum)
15: end function
16: function CustomPartitioner(key, value, numReduceTasks)
17:     return key
18: end function
19: function Reduce(key, list(values))
20:     HashSet<String> set = NULL
21:     function cleanup
22:         part_id = getTaskId()
23:         set.sort()
24:         write each value of set
25:         in file named "uniqueIPs/part-r-" + part_id
26:     end function
27:     for each value in values do
28:         set.add(ipFrom)
29:         set.add(ipTo)
30:         write(key, value)
31: end function
```

τελική τιμή. Χρησιμοποιώντας αυτή την πληροφορία μπορούμε να κρατήσουμε στο hashtable που θα φτιάξουμε μόνο τις διευθύνσεις που όντως εμφανίζονται στα αρχεία. Η εργασία αυτή, λοιπόν, αναλαμβάνει στη map φάση να διαβάσει το αρχείο με τα σημεία διαχωρισμού και στη συνέχεια φροντίζει ανάλογα με τις διευθύνσεις IP με τις οποίες σχετίζεται η κάθε εγγραφή να βγάζει ως έξοδο ένα μοναδικό ακέραιο που συμβολίζει το αναγνωριστικό του τμήματος (partition) που θα πρέπει να τοποθετηθεί αυτή η εγγραφή. Στη συνέχεια χρησιμοποιώντας ένα custom partitioner στην εργασία Map Reduce στέλνουμε τις εγγραφές για επεξεργασία στη διεργασία reduce με το αντίστοιχο αναγνωριστικό. Η διεργασία reduce έχει δύο ευθύνες:

1. Για κάθε εγγραφή που αντιστοιχεί στο κλειδί εισόδου εμφανίζει το αποτέλεσμα στην έξοδο.

2. Έχει μία δομή στην οποία κρατάει όλες τις μοναδικές IP διευθύνσεις που εμφανίζονται στο αρχείο και πριν την ολοκλήρωσή της γράφει το αποτέλεσμα σε ένα αρχείο το όνομα του οποίου περιέχει το αναγνωριστικό της διεργασίας.



Σχήμα 4.2: Κατανομή του μεγέθους των αρχείων με το στατικό διαχωρισμό

Μετά από αυτό το διαχωρισμό δημιουργούνται συνολικά $148 * 148 = 21904$ αρχεία στα οποία βρίσκονται πλέον διαχωρισμένες οι διευθύνσεις IP με συγκεκριμένο τρόπο και μπορούμε να εκτελέσουμε ερωτήματα στηριζόμενοι στην πληροφορία από το αρχείο με τα σημεία διαχωρισμού ώστε να γνωρίζουμε κάθε φορά ποιες διευθύνσεις περιέχονται. Ένα μειονέκτημα αυτού του τρόπου είναι ότι ο διαχωρισμός γίνεται με κριτήριο να χωράνε τα δεδομένα από τα κομμάτια του αρχείου με τα ονόματα υπολογιστών στην κύρια μνήμη και επομένως δε λαμβάνεται υπόψη η κατανομή των διευθύνσεων στο κύριο σύνολο δεδομένων. Αυτό έχει ως συνέπεια να δημιουργούνται πολλά αρχεία που περιέχουν πολύ λίγες έως και καθόλου εγγραφές, ενώ από την άλλη υπάρχουν και αρχεία που έχουν πολλές εγγραφές, δημιουργείται δηλαδή μια ανισοκατανομή στο μέγεθος των τελικών αρχείων. Το σχήμα 4.2 παρουσιάζει αυτό το πρόβλημα.

Ψευδοκώδικας 5 Συνάρτηση GetDns

```
1: ArrayList<String> splitPointsList = NULL
2: HashMap<String, String> DnsMap = NULL
3: partition_no = 148
4: function EVALUATE(partNum, ip, splitPointsFile)
5:     if DnsMap == NULL then
6:         read PartitionFile in splitPontsList
7:         table = new HTable(conf, "table_name")
8:         scan.setStartRow(splitPointsList.get(partNum / partition_no - 1))
9:         scan.setStopRow(splitPointsList.get(partNum / partiton_no))
10:        br = new BufferedReader("uniqueIPs/part-r-" + partNum)
11:        line = br.readLine()
12:        s = table.getScanner(scan)
13:        for result = s.next(); result != null; result=s.next() do
14:            if line == NULL then
15:                break
16:            while result.getRowKey().compareTo(line) > 0 do
17:                line = br.readLine()
18:                if line == NULL then
19:                    break
20:            if result.getRowKey().equals(line) then
21:                DnsMap.put(result.getRowKey(),
22:                result.getValue(family, qualifier))
23:                line = br.readLine()
24:                if line == NULL then
25:                    break
26:        scan.setStartRow(splitPointsList.get(partNum%partition_no-1))
27:        scan.setStopRow(splitPointsList.get(partNum % partiton_no))
28:        s = table.getScanner(scan)
29:        for result = s.next(); result != null; result=s.next() do
30:            if line == NULL then
31:                break
32:            while result.getRowKey().compareTo(line) > 0 do
33:                line = br.readLine()
34:                if line == NULL then
35:                    break
36:            if result.getRowKey().equals(line) then
37:                DnsMap.put(result.getRowKey(),
38:                result.getValue(family, qualifier))
39:                line = br.readLine()
40:                if line == NULL then
41:                    break
42:    return DnsMap.get(ip)
43: end function
```

Όπως είναι εμφανές η συντριπτική πλειοψηφία των αρχείων δεν περιέχουν εγγραφές, γιατί όπως εξηγήσαμε αντιστοιχούν σε ζευγάρια περιοχών που δεν περιέχουν διευθύνσεις IP οι οποίες να εμφανίζουν μεταξύ τους κίνηση. Αυτά τα αρχεία είναι σκόπιμο να μην τα συμπεριλάβουμε στον τελικό πίνακα, αφού δεν θα συνεισφέρουν κάτι στο τελικό αποτέλεσμα. Παρατηρούμε ακόμα ότι υπάρχουν αρκετά αρχεία με μεσαίο μέγεθος αλλά όμως υπάρχουν μερικά που είναι σχετικά μεγάλα, γεγονός που δεν είναι πολύ καλό γιατί οι διεργασίες map που θα αναλάβουν να τα επεξεργαστούν ενδεχομένως να χρειαστούν περισσότερο χρόνο.

Έχοντας εξασφαλίσει ότι για κάθε αρχείο εισόδου είμαστε σε θέση να γνωρίζουμε ποιο τμήμα από τα δεδομένα που υπάρχουν στην HBase μας είναι χρήσιμο κάθε φορά, μπορούμε να προχωρήσουμε στην υλοποίηση μιας συνάρτησης που θα έχει παρόμοια συμπεριφορά με την αντίστοιχη που δημιουργήσαμε για τα αυτόνομα συστήματα. Οι βασικές αρχές είναι πάλι ότι έχουμε ένα hashtable που χρησιμεύει για την αποθήκευση της συμπληρωματικής πληροφορίας των ονομάτων των υπολογιστών και στη συνέχεια καθώς γίνεται η ανάγνωση της κάθε εγγραφής αυτό χρησιμεύει για να αντιστοίχιζεται η κάθε IP σε ένα DNS όνομα. Σημειώνουμε ότι χρησιμοποιείται η πληροφορία των μοναδικών IP που έχουμε συλλέξει ώστε να αποθηκεύονται στο hashtable μόνο οι απαραίτητες (δηλαδή όσες υπάρχουν στο αρχείο) διευθύνσεις. Το αρχείο των μοναδικών IP είναι ταξινομημένο και καθώς επίσης τα δεδομένα από την HBase είναι επίσης ταξινομημένα σε σχέση με τις διευθύνσεις, μπορούμε να εφαρμόσουμε ένα είδους sort merge join για να συνδυάσουμε τη ζητούμενη πληροφορία. Ο ψευδοκώδικας 5 παρουσιάζει συνοπτικά αυτές τις λειτουργίες.

Με τον τρόπο που έχουμε ήδη περιγράψει προσθέτουμε τη συνάρτηση αυτή στο Hive και στη συνέχεια χρησιμοποιούμε ένα παρόμοιο Hive script με αυτό που χρησιμοποιήσαμε για τα αυτόνομα συστήματα προκειμένου να εκτελέσουμε το επιθυμητό ερώτημα.

4.3.3 Πειραματικά αποτελέσματα της μεθόδου του στατικού διαχωρισμού

Στην ενότητα αυτή παρουσιάζουμε τα αποτελέσματα της μεθόδου του στατικού διαχωρισμού, τόσο κατά τη φάση της κατάτμησης των δεδομένων όσο και κατά τη φάση εκτέλεσης του ερωτήματος. Στον πίνακα 4.2 συνοψίζονται τα αποτελέσματα αυτά συγκεντρωτικά.

Έχουμε να παρατηρήσουμε ότι τόσο η εργασία που πραγματοποιεί την κατάτμηση του συνόλου δεδομένων, όσο και το ίδιο το ερώτημα είναι αρκετά αργά στην εκτέλεσή τους. Παρόλο που η διαδικασία του partitioning γίνεται μία φορά πριν την εκτέλεση οποιωνδήποτε ερωτημάτων, ο χρόνος είναι εξαιρετικά μεγάλος και οφείλεται στο γεγονός ότι δημιουργούνται πάρα πολλά αρχεία, τα οποία τελικώς παραμένουν άδεια, αυξάνουν όμως το χρόνο εκτέλεσης αφού το σύστημα αρχικοποιεί όλες τις reduce

Job	Job Phase	# Tasks	Bytes Read		Bytes Written		Elapsed Time	
			File	HDFS	File	HDFS	Total	Per Task
Partition	Map	4701	860GB	50GB	320GB	0	58min	1min
	Reduce	21904	170GB	0	175GB	25GB	14h	2min
	Total	26605	1030GB	50GB	495GB	25GB	14h 30min	
Query	Map	2316	560MB	27GB	1.5GB	0	7h 20min	20min
	Reduce	99	260MB	0	276MB	40MB	5min	2min
	Total	2415	822MB	27GB	1.8GB	40MB	7h 30min	

Πίνακας 4.2: Συγκεντρωτικά αποτελέσματα από το static partitioning

διεργασίες ασχέτως αν υπάρχουν δεδομένα να υποβληθούν σε επεξεργασία (κυρίως όπως βλέπουμε ο χρόνος εξαρτάται από τη φάση reduce). Όσον αφορά το χρόνο εκτέλεσης του ερωτήματος παρατηρούμε ένα σχετικά μεγάλο μέσο όρο εκτέλεσης της κάθε διεργασίας map γεγονός που οφείλεται στη μεταφορά των δεδομένων από την HBase στη μνήμη της διεργασίας. Από τα log files μπορούμε να συμπεράνουμε ότι η διαδικασία μεταφοράς των δεδομένων αποτελεί περίπου το 90% του χρόνου εκτέλεσης, ενώ η επεξεργασία μόνο το 10% αντίστοιχα. Οι χρόνοι αυτοί μας οδηγούν στο να προσπαθήσουμε να βελτιστοποιήσουμε τη διαδικασία εκτέλεσης, στοχεύοντας τόσο σε καλύτερο αλγόριθμο κατάτμησης των δεδομένων, όσο και σε βελτιστοποίηση του χρόνου που χρειάζεται για να φέρουμε τα δεδομένα από την HBase. Συγκεκριμένα, στοχεύουμε σε δύο κύριες μεθόδους που αναλύονται στις επόμενες ενότητες. Η πρώτη είναι η βελτιστοποίηση του τρόπου που συλλέγουμε τα δεδομένα από την HBase με τη χρήση του scan, ενώ η δεύτερη αφορά την κατάτμηση των δεδομένων χρησιμοποιώντας ένα γενικότερο τρόπο που θα επιτρέπει το χωρισμό των δεδομένων πιο αποδοτικά, δηλαδή σε αρχεία περίπου ίσου μεγέθους και σε σημαντικά λιγότερο χρόνο.

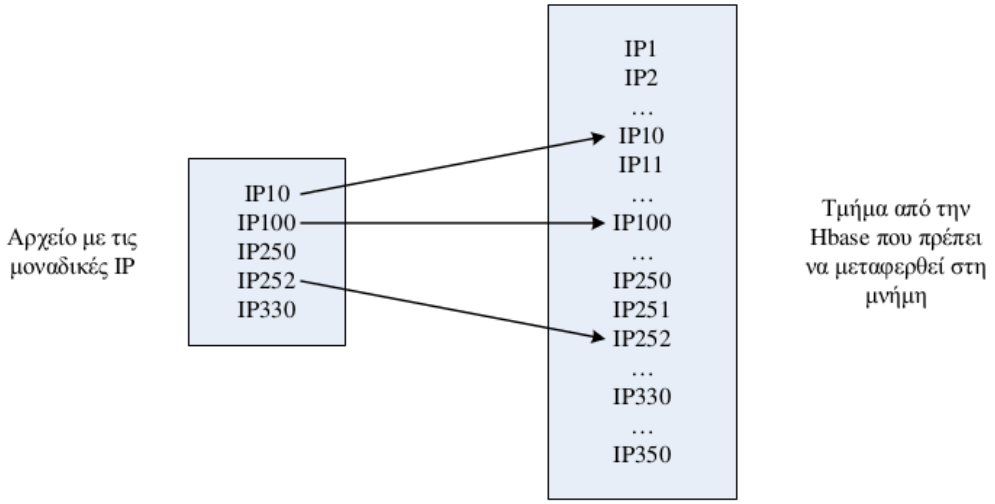
4.4 Βελτιστοποίηση της μεθόδου scan από την HBase

4.4.1 Εντοπισμός του προβλήματος

Στην ενότητα αυτή θα αναφερθούμε στη βελτιστοποίηση που έγινε στον τρόπο εκτέλεσης του scan από την HBase. Η λειτουργία αυτή όπως δίνεται από το API της HBase επιτρέπει στον προγραμματιστή να φέρει από την HBase μία ομάδα εγγραφών που αντιστοιχούν σε ένα εύρος κλειδιών που ορίζεται από το χρήστη. Το API προσφέρει μόνο τη δυνατότητα σειριακής προσπέλασης των δεδομένων με τη χρήση μίας μεθόδου $next()$ πράγμα που σημαίνει ότι θα πρέπει μία προς μία όλες οι πιθανές εγγραφές στο εύρος των κλειδιών που ορίσαμε να μεταφερθούν μέσω δικτύου από τον αντίστοιχο RegionServer στον οποίο βρίσκονται, στη μνήμη της διεργασίας.

Το πρόβλημα που ανακύπτει στην δική μας περίπτωση είναι ότι ο αριθμός των εγγραφών που μας είναι χρήσιμες είναι πολύ μικρότερος από αυτόν των συνολικών που μεταφέρουμε από την HBase (καθορίζεται από το αρχείο με τις μοναδικές IP).

Το μειονέκτημα εντοπίζεται στο ότι η μία χρήσιμη διεύθυνση μπορεί να απέχει πολλές θέσεις από την επόμενη (σχ. 4.3) και αναγκαστικά θα μεταφερθούν όλες οι υπόλοιπες μέσω δικτύου χωρίς να μας ενδιαφέρουν. Προφανώς, λοιπόν το throughput του δικτύου που έχουμε διαθέσιμο είναι ένας ανασταλτικός παράγοντας στη διαδικασία μεταφοράς των δεδομένων, που με αυτό τον τρόπο κοστίζει σε χρόνο, ο οποίος μπορεί να εξοικονομηθεί αν χρησιμοποιήσουμε μία καλύτερη προσέγγιση. Συγκεκριμένα, προσθέτουμε μία επιπλέον λειτουργικότητα στο scan που εκτελείται από την HBase δημιουργώντας μία καινούρια μέθοδο που δεν είναι απαραίτητο να διασχίζει σειριακά τις εγγραφές, αλλά μπορεί να κάνει αναζήτηση σε μια εγγραφή αν της δώσουμε σαν όρισμα το κλειδί-στόχο. Η λειτουργία αυτής της μεδόδου στηρίζεται σε μία ευριστική συνάρτηση που αποφασίζει με τη βοήθεια μίας παραμέτρου αν είναι αποδοτικότερο να προσπελάσει σειριακά τις εγγραφές μέχρι τη ζητούμενη ή αν πρέπει να μεταβεί απευθείας σε μια επόμενη. Η παράμετρος αυτή είναι ουσιαστικά μία εκτίμηση των ενδιάμεσων εγγραφών που πρέπει να αγνοηθούν. Όταν οι ενδιάμεσες εγγραφές είναι λίγες, συμφέρει να τις προσπελάσουμε σειριακά μέχρι τη ζητούμενη με χρήση της μεθόδου next(), αλλιώς αν είναι πολλές συμφέρει να αγνοήσουμε τις επόμενες και να μεταβούμε απευθείας στη ζητούμενη.



Σχήμα 4.3: Οι μοναδικές IP που απαιτούνται για μεταφορά

Ψευδοκώδικας 6 Συνάρτηση GetDns με βελτιστοποιημένο scan

```
1: ArrayList<String> splitPointsList = NULL
2: HashMap<String, String> DnsMap = NULL
3: partition_no = 148
4: function EVALUATE(partNum, splitPointsFile, ip)
5:     if DnsMap == NULL then
6:         read PartitionFile in splitPontsList
7:         table = new HTable(conf, "table_name")
8:         scan.setStartRow(splitPointsList.get(partNum / partition_no - 1))
9:         scan.setStopRow(splitPointsList.get(partNum / partiton_no))
10:        br = new BufferedReader("uniqueIPs/part-r-" + partNum)
11:        line = br.readLine()
12:        s = table.getScanner(scan)
13:        result = s.next()
14:        while line != NULL && result != NULL do
15:            intUniqueIP = ip_to_int(line)
16:            intScanIP = ip_to_int(result.getRowKey())
17:            if intUniqueIP > intScanIP then
18:                result = s.seekTo(line, intUniqueIP-intScanIP)
19:            else
20:                line = br.seekTo(result.getRowKey())
21:            if line == NULL || result == NULL then
22:                break
23:            if result.getRowKey().equals(line) then
24:                DnsMap.put(result.getRowKey(), result.getValue())
25:        scan.setStartRow(splitPointsList.get(partNum%partition_no-1))
26:        scan.setStopRow(splitPointsList.get(partNum % partiton_no))
27:        s = table.getScanner(scan)
28:        result = s.next()
29:        while line != NULL && result != NULL do
30:            intUniqueIP = ip_to_int(line)
31:            intScanIP = ip_to_int(result.getRowKey())
32:            if intUniqueIP > intScanIP then
33:                result = s.seekTo(line, intUniqueIP-intScanIP)
34:            else
35:                line = br.seekTo(result.getRowKey())
36:            if line == NULL || result == NULL then
37:                break
38:            if result.getRowKey().equals(line) then
39:                DnsMap.put(result.getRowKey(), result.getValue())
40:    return DnsMap.get(ip)
41: end function
```

4.4.2 Υλοποίηση και αποτελέσματα της βελτιστοποίησης του scan

Η μέθοδος που προσθέσαμε προσφέρει τη λειτουργία που περιγράφηκε προηγουμένως και επιπροσθέτως μας επιτρέπει να υλοποιήσουμε μία πιο συμπαγή μορφή της συνάρτησης για τη μετατροπή των διευθύνσεων IP σε DNS ονόματα. Ο ψευδοκώδικας 6 παρουσιάζει τη μορφή αυτής της συνάρτησης.

Εκτελώντας τώρα το ίδιο ερώτημα σε σχέση με την προηγούμενη ενότητα παρατηρούμε θεαματική βελτίωση της απόδοσης στο χρόνο εκτέλεσης, όπως φαίνεται στον πίνακα 4.3.

Job Phase	# Tasks	Bytes Read		Bytes Written		Elapsed Time	
		File	HDFS	File	HDFS	Total	Per Task
Map	2316	456MB	27GB	1.5GB	0	38min	2min
Reduce	99	250MB	0	276MB	40MB	5min	2min
Total	2415	700GB	27GB	1.8GB	40MB	40min	

Πίνακας 4.3: Αποτελέσματα εκτέλεσης ερωτήματος IP-Dns με βελτιστοποιημένο scan

Συγκρίνοντας τα δεδομένα που φαίνονται παραπάνω με αυτά της προηγούμενης ενότητας παρατηρούμε ότι τα I/O δεδομένα που ανταλλάσσονται από τις διεργασίες map και reduce είναι ακριβώς τα ίδια, ενώ η σημαντική διαφορά έγκειται στο χρόνο ολοκλήρωσης της κάθε διεργασίας map (από 20 λεπτά σε 2 λεπτά) και κατά συνέπεια στο συνολικό χρόνο εκτέλεσης της map φάσης. Αυτή η θεαματική βελτίωση οφείλεται αποκλειστικά στον τρόπο που μεταφέρονται τα δεδομένα από την HBase που πλέον γίνεται πολύ αποδοτικά. Στη reduce φάση δεν έχουμε κάποια διαφορά στο χρόνο εκτέλεσης, πράγμα πολύ λογικό αφού οι reducers έχουν να επεξεργαστούν ακριβώς τα ίδια δεδομένα με πριν.

4.5 Διαχωρισμός των δεδομένων με τη χρήση K-dimensional tree

Στην ενότητα αυτή παρουσιάζουμε μία ακόμα βελτιστοποίηση που εφαρμόσαμε στην περίπτωση των ερωτημάτων με τα ονόματα υπολογιστών. Με τη συγκεκριμένη μέθοδο εστιάζουμε στον τρόπο διαχωρισμού των δεδομένων και στην προσπάθεια να γίνει αυτή η διαδικασία πιο ομοιόμορφα σε σχέση με το στατικό διαχωρισμό, που δε λαμβάνει υπόψη την κατανομή των διεύθυνσεων IP και το γεγονός ότι υπάρχουν κάποιες περιοχές που περιέχουν πολλές μαζεμένες, ενώ άλλες περιοχές είναι αρκετά αραιές. Χρειαζόμαστε λοιπόν μία δυναμική δομή δεδομένων που διευκολύνει το

χωρικό διαχωρισμό των δεδομένων σε περιοχές που περιέχουν παρόμοιο αριθμό δεδομένων. Μία τέτοια δομή δεδομένων είναι το K-dimensional tree που περιγράφουμε στη συνέχεια.

4.5.1 K-dimensional (K-d) tree

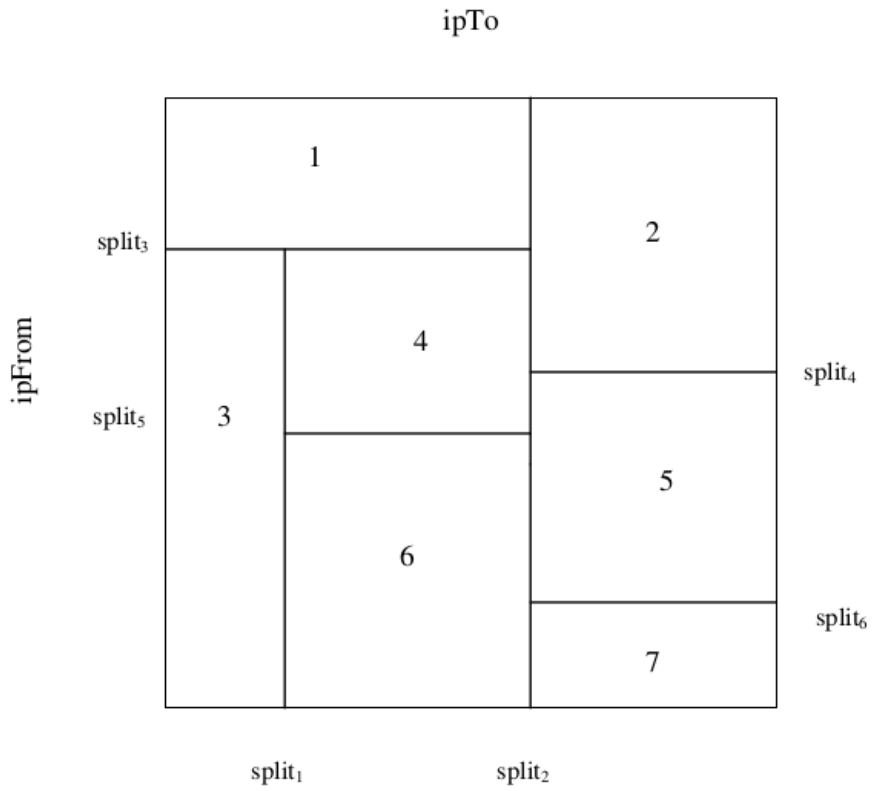
Γενικά για το K-dimensional tree

Το K-dimensional tree [9] (K-d tree) είναι μια δομή δεδομένων που χρησιμοποιείται για χωρική κατάτμηση και οργάνωση των σημείων σε ένα K-διάστατο χώρο δεδομένων, που βρίσκει χρησιμότητα σε πολλές εφαρμογές, όπως σε μία αναζήτηση κλειδιού πολλαπλών διαστάσεων (αναζητήσεις εύρους και κοντινότερου γείτονα). Τα K-d δέντρα είναι μια ειδική περίπτωση ενός δυαδικού δέντρου χωρικής κατάτμησης.

Κάθε κόμβος είναι ένα σημείο στον k-διάστατο χώρο. Κάθε κόμβος που δεν είναι φύλλο του δέντρου μπορεί να θεωρηθεί ως ένα υπερεπίπεδο που χωρίζει το συνολικό χώρο σε δύο μέρη, που ονομάζονται ημι-χώροι. Αυτός ο διαχωρισμός αφορά μία συγκεκριμένη διάσταση του χώρου που επιλέγεται με ένα κατάλληλο αλγόριθμο κάθε φορά. Τα σημεία που βρίσκονται στα αριστερά αυτού του υπερεπιπέδου αντιπροσωπεύονται από το αριστερό υποδέντρο, ενώ αντίστοιχα τα σημεία που βρίσκονται δεξιά του υπερεπιπέδου αντιπροσωπεύονται από το δεξιό υποδέντρο. Στη συνέχεια αναλύουμε κάποιες βασικές λειτουργίες του k-d δέντρου, οι οποίες μας είναι χρήσιμες για την υλοποίηση των συναρτήσεων που θα παρουσιάσουμε στην επόμενη ενότητα.

Εισαγωγή ενός στοιχείου στο K-d tree

Η εισαγωγή ενός στοιχείου γίνεται αντίστοιχα με οποιοδήποτε άλλο δυαδικό δέντρο αναζήτησης. Ξεκινώντας από τον κόμβο-ρίζα αναζητούμε το φύλλο που πρέπει να τοποθετηθεί το καινούριο σημείο. Αν είναι μεγαλύτερο από την τιμή του κόμβου ψάχνουμε στο δεξιό υποδέντρο, αλλιώς ψάχνουμε στο αριστερό. Η ιδιαιτερότητα παρουσιάζεται όταν ο κόμβος πρέπει να χωριστεί σε δύο κόμβους, γεγονός που καθορίζεται από μία παράμετρο που ρυθμίζει το μέγιστο αριθμό σημείων (bucketCapaxity) που μπορεί να περιέχει ένας κόμβος-φύλλο. Όταν λοιπόν το καινούριο σημείο που θα εισαχθεί θα αυξήσει τον αριθμό των σημείων του συγκεκριμένου κόμβου πάνω από αυτή τη μέγιστη τιμή, τότε ο κόμβος πρέπει να διαιρεθεί σε δύο. Για να γίνει αυτό επιλέγουμε τη διάσταση με το μεγαλύτερο εύρος τιμών και σε αυτή τη διάσταση βρίσκουμε την ενδιάμεση τιμή (median value) από τα ήδη υπάρχοντα σημεία και χωρίζουμε τον κόμβο σε δύο καινούριους, τοποθετώντας την καινούρια τιμή στον κατάλληλο. Αυτή η διαδικασία μπορεί να δημιουργήσει ανομοιόμορφες περιοχές (σχ. 4.4), σε αντίθεση με το στατικό τρόπο διαχωρισμού, αλλά μας εξασφαλίζει ότι όλα τα αρχεία που θέλουμε να δημιουργήσουμε θα περιέχουν ισοκατανεμημένο αριθμό εγγραφών.



Σχήμα 4.4: Παράδειγμα κατάτμησης ενός χώρου με τη χρήση K-d tree

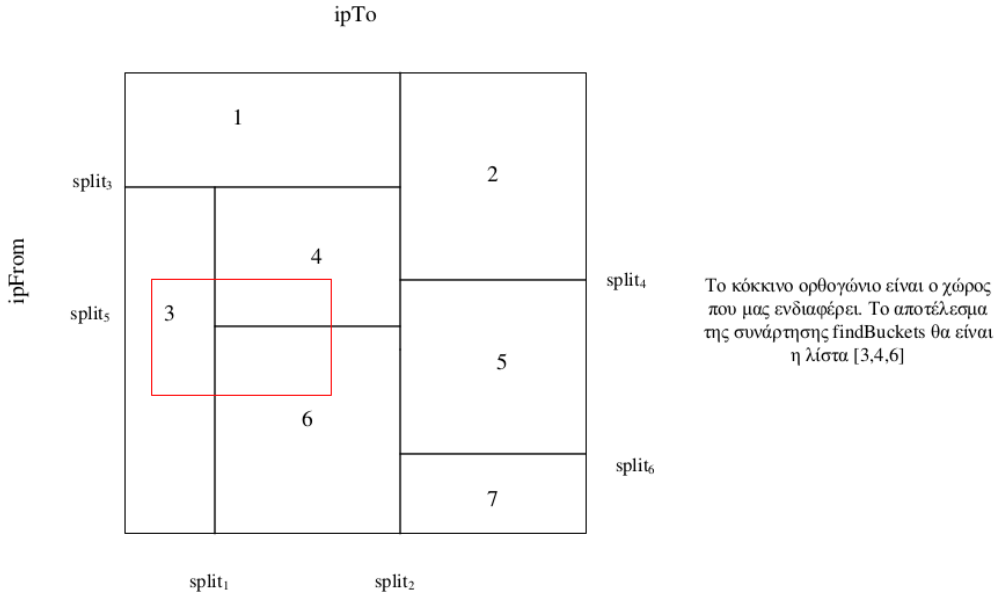
Αναζήτηση ενός σημείου

Η αναζήτηση στοιχείου είναι ακριβώς αντίστοιχη όπως σε ένα δυαδικό δέντρο αναζήτησης. Ο αλγόριθμος δέχεται ως είσοδο ένα σημείο του k-διάστατου χώρου και αναζητεί τον κόμβο-φύλλο στον οποίο ανήκει σε χρόνο $O(logn)$. Ως αποτέλεσμα επιστρέφεται το αναγνωριστικό του φύλλου στο οποίο ανήκει το ζητούμενο σημείο.

Αναζήτηση φύλλων που περιέχουν μία περιοχή σημείων

Εκτός από την αναζήτηση ενός συγκεκριμένου σημείου, στο k-d δέντρο έχουμε τη δυνατότητα να αναζητήσουμε μία λίστα με τα αναγνωριστικά των φύλλων που περιέχουν ένα υποσύνολο του συνολικού χώρου. Συγκεκριμένα, ο χρήστης δίνει τις ελάχιστες και μέγιστες τιμές που τον ενδιαφέρουν από κάθε διάσταση, οι οποίες σχηματίζουν ένα υπερκύβο k διαστάσεων και σαν αποτέλεσμα επιστρέφεται μία λίστα

με τα αναγνωριστικά των φύλλων που έχουν επικάλυψη με αυτό χώρο και περιέχουν κάποιες από τις τιμές του. Ένα παράδειγμα αυτής της λειτουργίας φαίνεται στο σχήμα 4.5.



Σχήμα 4.5: Παράδειγμα εφαρμογής της αναζήτησης φύλλων που περιέχονται σε μία περιοχή τιμών

4.5.2 Υλοποίηση διαχωρισμού των δεδομένων με χρήση του K-d tree

Στην ενότητα αυτή θα περιγράψουμε τον τρόπο υλοποίησης του k-d partitioning όπως τον εφαρμόσαμε στην περίπτωση των ερωτημάτων που αφορούν την αντιστοίχιση των IP σε DNS ονόματα. Η υλοποίηση αποτελείται από τρεις φάσεις, μία δειγματοληψία των δεδομένων για να δημιουργηθεί το k-d tree, τη διαδικασία κατάτμησης των δεδομένων και την υλοποίηση της συνάρτησης UDF στο Hive.

Δειγματοληψία δεδομένων

Το πρώτο βήμα για την υλοποίηση του διαχωρισμού με τη μέθοδο του k-d tree είναι η ίδια η δημιουργία του δέντρου. Για το σκοπό αυτό κάνουμε μία δειγματοληψία των δεδομένων για να δημιουργήσουμε το δέντρο, τα φύλλα του οποίου θα

περιγράφουν τα όρια των περιοχών των τιμών που θα περιέχουν τα τελικά αρχεία. Η δειγματοληψία γίνεται με μια εργασία Map Reduce που διαβάζει ένα ποσοστό των συνολικών δεδομένων (π.χ. 1%) και στη συνέχεια η συνάρτηση reduce διαβάζει όλες αυτές τις εγγραφές, απομονώνει τα πεδία που αντιστοιχούν στις διαστάσεις που μας ενδιαφέρουν και εισάγει τα σημεία που προκύπτουν στο δέντρο. Τέλος, γράφει το δέντρο αυτό σε ένα αρχείο, από το οποίο θα το διαβάσει μετά η εργασία που θα επιτελέσει το partitioning.

Ψευδοκώδικας 7 Δειγματοληψία των δεδομένων για τη δημιουργία του K-d δέντρου

```
1: function MAP(sampleRate)
2:     function RUN(context) setup(context)
3:         while context.nextKeyValue() do
4:             if getRandomNumber < sampleRate then
5:                 map(context.getCurrentKey(), context.getCurrentValue(),
6:                         context)
7:         cleanup(context)
8:     end function
9: end function
10: function REDUCE(key, list(values), dimensions)
11:     KdTree kd = new KdTree(dimensions.length, bucketSize)
12:     for each val in values do
13:         parts = val.split(" ")
14:         point = new double[dimensions.length]
15:         for i = 0; i < dimensions.length; i++ do
16:             point[i] = make_to_double(part[dimensions[i]])
17:         kd.add(point, dummy_int)
18:     kd.printTree("tree_partition")
19: end function
```

Κατάτμηση των δεδομένων

Αφού έχει δημιουργηθεί το δέντρο που περιέχει την απαραίτητη πληροφορία για να γίνει η κατάτμηση των δεδομένων, το επόμενο βήμα είναι να πραγματοποιηθεί ο διαχωρισμός των δεδομένων βάσει αυτής της πληροφορίας. Η εργασία Map Reduce που υλοποιήσαμε για αυτό το σκοπό έχει ομοιότητες με την αντίστοιχη που επιτελεί το στατικό διαχωρισμό. Και σε αυτή την περίπτωση, στη φάση map χρησιμοποιείται η πληροφορία για τον τρόπο που διαχωρίζονται τα δεδομένα ώστε να επαυξηθούν με ένα αναγνωριστικό που συμβολίζει το partition στο οποίο ανήκουν. Ένας custom partitioner στέλνει τα ενδιάμεσα δεδομένα στην κατάλληλη διεργασία reduce η οποία φροντίζει να παράγει στην έξοδο τις εγγραφές και το αρχείο με τις μοναδικές IP που

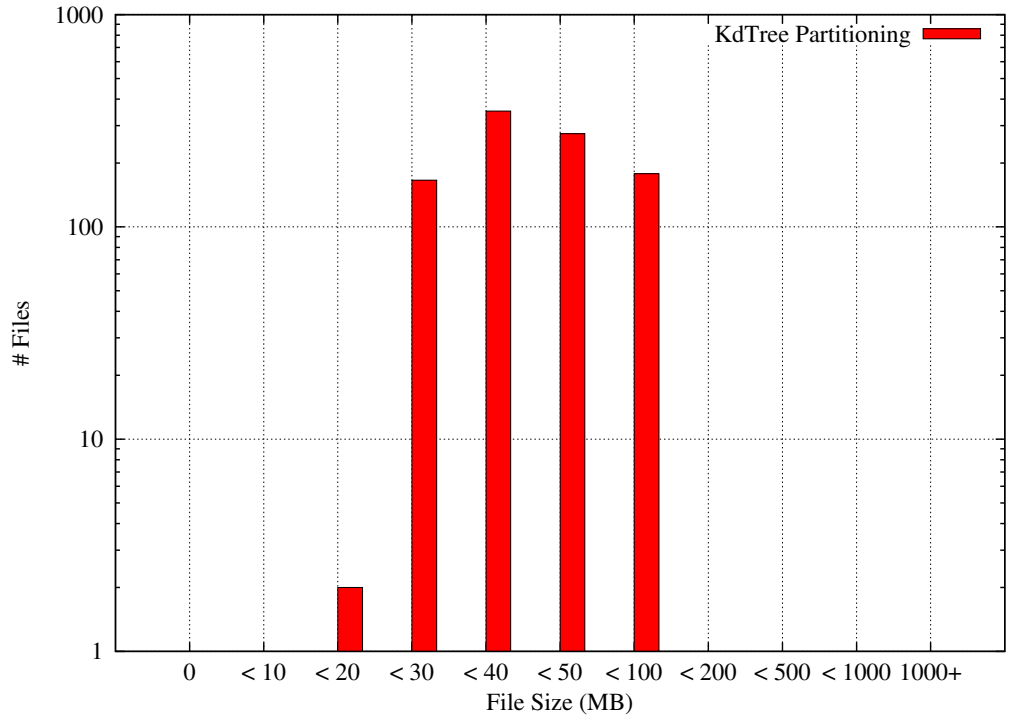
περιέχει το κάθε αρχείο. Ο ψευδοκώδικας 8 συνοψίζει αυτές τις λειτουργίες.

Ψευδοκώδικας 8 Δυναμικός διαχωρισμός των δεδομένων

```
1: function Map
2:     function setup(dimensions)
3:         min, max = new double[dimensions.length]
4:         read treePartitionFile in KdTree kd
5:         for each m in min do
6:             m = 0
7:         for each m in max do
8:             m = maxDouble
9:         kd.findBuckets(min, max, l)
10:        l.sort()
11:    end function
12:    for every record r in file do
13:        parts = r.split(" ")
14:        point = new double[dimensions.length]
15:        for i = 0; i < dimensions.length; i++ do
16:            point[i] = make_to_double(part[dimensions[i]])
17:        partNum = kd.find(point)
18:        write(partNum, r + " " + partNum)
19: end function
20: function CustomPartitioner(key, value, numReduceTasks, l)
21:    return l.indexOf(key)
22: end function
23: function Reduce(key, list(values))
24:    HashSet<String> set = NULL
25:    function cleanup
26:        part_id = getTaskId()
27:        set.sort()
28:        write each value of set
29:        in file named "uniqueIPs/part-r-" + part_id
30:    end function
31:    for each value in values do
32:        set.add(ipFrom)
33:        set.add(ipTo)
34:        write(key, value)
35: end function
```

Μετά από αυτό το διαχωρισμό δημιουργούνται μόνο 972 αρχεία σε σύγκριση με τα περίπου 20000 της μεθόδου του στατικού διαχωρισμού. Ο λόγος είναι ότι τα αρχεία αυτά είναι περισσότερο ισορροπημένα σε σχέση με τον αριθμό των εγγραφών που

περιέχει το καθένα. Πλέον δεν υπάρχουν αρχεία που να μην περιέχουν καθόλου εγγραφές, ενώ λείπουν και τα σχετικά μεγάλα αρχεία που περιείχαν πάρα πολλές εγγραφές. Το γεγονός αυτό αποτυπώνεται στο σχήμα 4.6.



Σχήμα 4.6: Κατανομή μεγέθους αρχείων με δυναμικό διαχωρισμό

Είναι εμφανές λοιπόν ότι αυτός ο τρόπος υπερτερεί διότι τα αρχεία που προκύπτουν είναι πιο ισορροπημένα όσον αφορά το μέγεθός τους, ενώ τα δεδομένα πλέον είναι χωρισμένα ως προς διάφορες παραμέτρους γεγονός που θα μας επιτρέψει να εκτελέσουμε διάφορα ερωτήματα χρησιμοποιώντας κάποιο φίλτρο ως προς μία ή περισσότερες από τις διαστάσεις που έχουμε χρησιμοποιήσει για το διαχωρισμό.

Υλοποίηση της συνάρτησης GetDns

Έχοντας έτοιμα τα δεδομένα και το δέντρο μπορούμε να προχωρήσουμε στην υλοποίηση της συνάρτησης μετατροπής μιας διεύθυνσης IP στο αντίστοιχο DNS όνομα. Η υλοποίηση αυτή είναι παρόμοια με την περίπτωση του στατικού διαχωρισμού. Ουσιαστικά, αυτό που αλλάζει είναι το αρχικό κομμάτι που αφορά τον τρόπο που βρίσκουμε σε ποιο partition ανήκει το αρχείο που διαβάζεται. Ο ψευδοκώδικας 9 παρουσιάζει την υλοποίηση αυτής της συνάρτησης.

Ψευδοκώδικας 9 Συνάρτηση GetDns με χρήση K-d tree

```
1: ArrayList<String> splitPointsList = NULL
2: HashMap<String, String> DnsMap = NULL
3: partition_no = 148
4: function EVALUATE(partNum, treePartition, uniqueIP, dimensions, ip)
5:     if DnsMap == NULL then
6:         min, max = new double[dimensions.length]
7:         read treePartitionFile in KdTree kd
8:         for each m in min do
9:             m = 0
10:        for each m in max do
11:            m = maxDouble
12:        kd.findBuckets(min, max, l)
13:        l.sort()
14:        partNum = l.indexOf(partNum)
15:        table = new HTable(conf, "table_name")
16:        br = new BufferedReader(uniqueIP + "/part-r-" + partNum)
17:        line = br.readLine()
18:        scan.setStartRow(line)
19:        s = table.getScanner(scan)
20:        result = s.next()
21:        while line != NULL && result != NULL do
22:            intUniqueIP = ip_to_int(line)
23:            intScanIP = ip_to_int(result.getRowKey())
24:            if intUniqueIP > intScanIP then
25:                result = s.seekTo(line, intUniqueIP-intScanIP)
26:            else
27:                line = br.seekTo(result.getRowKey())
28:            if line == NULL || result == NULL then
29:                break
30:            if result.getRowKey().equals(line) then
31:                DnsMap.put(result.getRowKey(), result.getValue())
32:    return DnsMap.get(ip)
33: end function
```

4.5.3 Πειραματικά αποτελέσματα

Στην ενότητα αυτή παρουσιάζουμε τα αποτελέσματα της μεθόδου του δυναμικού διαχωρισμού με τη χρήση K-d tree, τόσο κατά τη φάση της κατάτμησης των δεδομένων όσο και κατά τη φάση εκτέλεσης του ερωτήματος. Επίσης, παρουσιάζουμε τα αποτελέσματα από την εκτέλεση ενός ερωτήματος που περιορίζεται στα δεδομένα μίας συγκεκριμένης εβδομάδας. Στον πίνακα 4.4 συνοψίζονται τα αποτελέσματα αυτά.

Job	Job Phase	# Tasks	Bytes Read		Bytes Written		Elapsed Time	
			File	HDFS	File	HDFS	Total	Per Task
Sampling	Map	4701	0GB	50GB	1GB	0	14min	14sec
	Reduce	1	823MB	0	823MB	207KB	15min	15min
	Total	4702	823MB	50GB	1.8GB	207KB	16min 30sec	
Partition	Map	4701	67GB	50GB	80GB	0	55min	1.5min
	Reduce	972	41GB	0	41GB	43GB	1h 30min	5min
	Total	5673	109GB	50GB	121GB	43GB	1h 50min	
Query	Map	972	94MB	43GB	1.7GB	0	1h 2min	4min
	Reduce	99	300MB	0	315MB	42MB	3min	1min
	Total	1071	394MB	43GB	2GB	42MB	1h 5min	
Filter 1 week	Map	81	4.5MB	3GB	109MB	0	8min	4min
	Reduce	15	41GB	0	44MB	22MB	6min	6min
	Total	96	45GB	3GB	153MB	22MB	9min	

Πίνακας 4.4: Συγκεντρωτικά αποτελέσματα από το δυναμικό partitioning

Παρατηρούμε και ποσοτικά και χρονικά την υπεροχή του δυναμικού τρόπου διαχωρισμού σε σχέση με τη στατική μέθοδο. Στη φάση map της εργασίας οι χρόνοι είναι παραπλήσιοι, όμως ο συνολικός χρόνος εκτέλεσης του partitioning είναι πολύ μικρότερος από την άλλη περίπτωση κυρίως εξαιτίας της καλύτερης κατάτμησης των δεδομένων που δεν απαιτεί να δημιουργηθούν πολλά περιττά αρχεία (δηλαδή αρχεία που δεν περιέχουν κάποιες εγγραφές). Πρέπει να επισημάνουμε επίσης ότι και τα δεδομένα που μεταφέρονται είναι αρκετά λιγότερα σε αυτή την περίπτωση. Όσον αφορά το ερώτημα, ο χρόνος εκτέλεσης αυξάνεται σε ένα μικρό ποσοστό, γεγονός που δικαιολογείται. Όταν χρησιμοποιούμε περισσότερες διαστάσεις για το διαχωρισμό των δεδομένων, είναι πιθανό σε κάποια partitions το εύρος των IP που περιέχονται σε αυτά να είναι σχετικά μεγάλο, πράγμα που σημαίνει ότι όταν χρειαστεί να φέρουμε από την HBase τα meta-δεδομένα μπορεί να απαιτείται περισσότερος χρόνος για να γίνει η μεταφορά τους. Αυτό το κόστος είναι σχετικά μικρό, αν αναλογιστούμε ότι αυτή η μέθοδος του δυναμικού διαχωρισμού μας επιτρέπει να εκτελέσουμε ερωτήματα σε συγκεκριμένες περιόδους (π.χ. με ένα φίλτρο στην ημερομηνία) ή που να αφορούν συγκεκριμένες διευθύνσεις IP χωρίς να χρειαστεί να επεξεργαστούμε όλα τα δεδομένα πρώτα, αλλά εκτελώντας το ερώτημα μόνο στα αρχεία που μας ενδιαφέρουν, κάνοντας χρήση του δέντρου. Το ερώτημα που παρουσιάζεται στον πίνακα 4.4 αφορά τα δεδομένα μίας εβδομάδας και εκτελείται σε μόλις 9 λεπτά, ενώ με τον άλλο τρόπο (στατικό διαχωρισμό) θα χρειαζόταν παρόμοιο χρόνο με το ερώτημα που αφορά τα συνολικά δεδομένα, αφού θα έπρεπε να γίνει επεξεργασία σε όλα τα αρχεία εισόδου. Το πλεονέκτημα επομένως αυτής της μεθόδου είναι η ευελιξία που μας προσφέρει στην εκτέλεση εξειδικευμένων ερωτημάτων σε ένα περιορισμένο εύρος των συνολικών δεδομένων.

Κεφάλαιο 5

Πειραματικό μέρος: Αξιολόγηση των παραμέτρων του συστήματος

Στο κεφάλαιο αυτό θα παρουσιάσουμε την επίδραση διάφορων παραμέτρων στην απόδοση εκτέλεσης των ερωτημάτων. Πρώτα, εξετάζουμε τη διαφορά μεταξύ δύο συστημάτων εκτέλεσης Map Reduce εργασιών, του Hive και του Spark αναλύοντας τα πλεονεκτήματα του δεύτερου σε κάποιες περιπτώσεις. Στη συνέχεια, πειραματιζόμαστε με την κλιμάκωση των ερωτημάτων όσον αφορά τον αριθμό των κόμβων του cluster και το μέγεθος των συνολικών δεδομένων. Τέλος, δείχνουμε την επίδραση του αριθμού των διαστάσεων στο χρόνο εκτέλεσης των ερωτημάτων που αφορούν τα DNS ονόματα.

5.1 Hive vs Shark

Στην ενότητα αυτή θα παρουσιάσουμε τη διαφορά στο χρόνο εκτέλεσης των ερωτημάτων ανάλογα με το σύστημα που χρησιμοποιούμε κάθε φορά. Για αρχή, παρουσιάζουμε μία σύγκριση των δύο συστημάτων και τα πλεονεκτήματα του Shark που προκύπτουν [18].

Ενδιάμεσα αποτελέσματα

Το Hive γράφει τα ενδιάμεσα αποτελέσματα στο δίσκο σε δύο περιπτώσεις, η μία είναι για να υπάρχει backup των αποτελεσμάτων μίας διεργασίας map σε περίπτωση που αστοχήσει η διεργασία reduce, ενώ η άλλη αφορά εργασίες Map Reduce που αποτελούνται από πολλαπλά στάδια και απαιτείται να αποθηκεύονται στο HDFS τα ενδιάμεσα αποτελέσματα από κάθε στάδιο.

Για την πρώτη περίπτωση το Shark χρησιμοποιεί την κύρια μνήμη για να γράφει τα αποτελέσματα και αν αυτά είναι πάρα πολλά, τότε γίνεται spill στο δίσκο. Αυτή η προσέγγιση προσφέρει σημαντική επιτάχυνση στην περίπτωση που έχουμε ερωτήματα aggregation και filtering όπου τα δεδομένα εξόδου είναι σημαντικά λιγότερα από τα δεδομένα εισόδου. Για τη δεύτερη περίπτωση το Shark επεκτείνει το μοντέλο εκτέλεσης του Map Reduce σε ένα γενικότερο ακυκλικό γράφο εκτέλεσης που έχει τη δυνατότητα να τρέξει πολλαπλά στάδια Map Reduce χωρίς να χρειάζεται να γράφει τα ενδιάμεσα αποτελέσματα στο δίσκο.

Δομή και διάταξη των δεδομένων

Πολλά συστήματα υποστηρίζουν έξυπνους τρόπους αποθήκευσης δεδομένων για να επιταχύνουν την εκτέλεση ερωτημάτων. Το Hive για παράδειγμα, υποστηρίζει τη δημιουργία partitions στους πίνακες (μία δομή αντίστοιχη με ευρετήριο, αφού γνωρίζει το εύρος κλειδιών σε κάθε αρχείο και δεν είναι απαραίτητο να διαβάσει όλο τον πίνακα για την εκτέλεση ορισμένων ερωτημάτων). Αυτό επεκτείνεται στο Shark που υποστηρίζει in-memory αναπαραστάσεις των δεδομένων. Συγκεκριμένα, το Shark αναπαριστά ένα μπλοκ από ν-άδες ως μία μοναδική εγγραφή στο Spark, δηλαδή ένα Java object. Ένα ακόμη χαρακτηριστικό που δεν υπήρχε σε προηγούμενα συστήματα είναι ο έλεγχος του τρόπου που διαχωρίζονται τα δεδομένα στους κόμβους του συστήματος.

Στρατηγικές εκτέλεσης

Το Hive αναλώνει σημαντικό χρόνο στην ταξινόμηση των δεδομένων προτού γράψει τα ενδιάμεσα στο δίσκο, γεγονός που αποτελεί περιορισμό του μοντέλου ενός περάσματος του Map Reduce. Το Spark έχει κάποια χαρακτηριστικά που βοηθούν να ξεπεραστεί αυτός ο περιορισμός. Συγκεκριμένα, το Spark χρησιμοποιώντας ένα μοντέλο εκτέλεσης που ονομάζεται Partial DAG execution (PDE) έχει τη δυνατότητα να τροποποιεί τη ροή εκτέλεσης των διεργασιών βάσει στατιστικών που συλλέγονται από τα δεδομένα. Είναι δυνατό να αλλάξει ακόμα και η στρατηγική εκτέλεσης ενός join ανάλογα με το εύρος των κλειδιών που περιέχονται στα ενδιάμεσα αποτελέσματα.

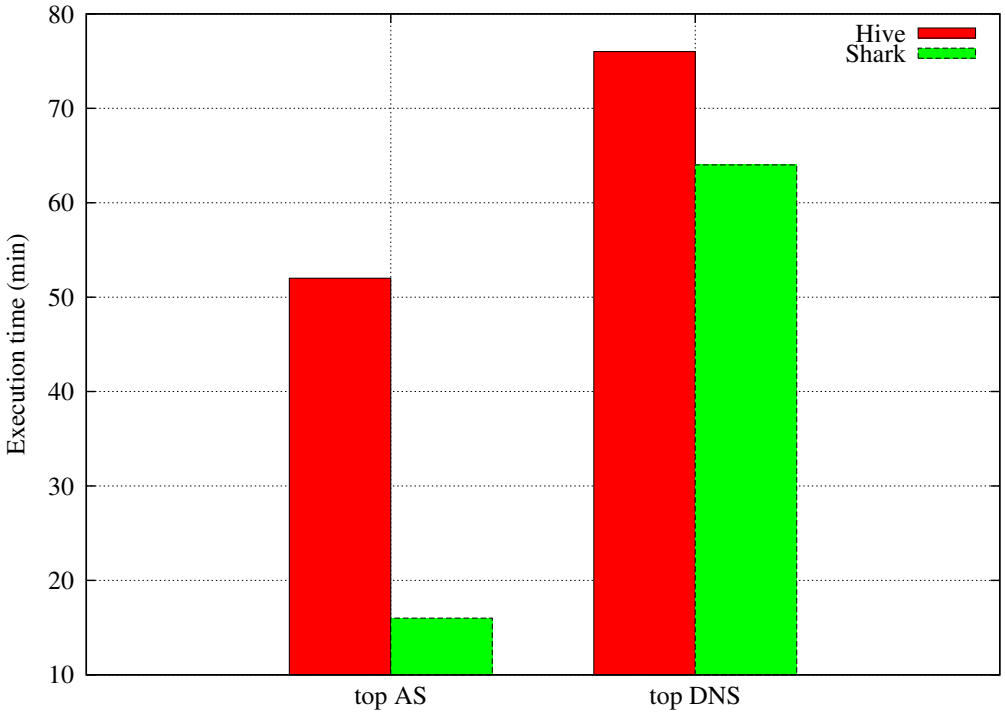
Κόστος δρομολόγησης των διεργασιών

Τα παραδοσιακά συστήματα εκτέλεσης εργασιών Map Reduce ήταν σχεδιασμένα για πολύωρες εργασίες, όπου η κάθε διεργασία διαρκούσε μερικά λεπτά. Δρομολογούσαν την κάθε διεργασία σε διαφορετική διεργασία του λειτουργικού συστήματος και σε αρκετές περιπτώσεις αυτό είχε σημαντική καθυστέρηση ακόμα και στην υποβολή μίας διεργασίας. Για παράδειγμα, το Hadoop χρησιμοποιεί περιοδικά "heart-beats" για κάθε εργάτη περίπου κάθε 3 δευτερόλεπτα για να αναθέσει διεργασίες και καταλαβαίνει συνολική καθυστέρηση στην έναρξη των διεργασιών κάθε 5-10 δευτερόλεπτα.

Αυτό προφανώς δεν είναι αποδεκτό στην περίπτωση ερωτημάτων που θέλουμε να εκτελούνται αρκετά γρήγορα. Το Spark αποφεύγει αυτό το πρόβλημα χρησιμοποιώντας μία event-driven βιβλιοθήκη RPC για να δρομολογεί διεργασίες και επίσης επαναχρησιμοποιεί τις διεργασίες που έχουν ήδη ολοκληρώσει την εκτέλεσή της προηγούμενης δουλειάς τους. Με αυτό τον τρόπο μπορεί να δρομολογήσει χιλιάδες διεργασίες το δευτερόλεπτο με μία καθυστέρηση της τάξης των msec. Ένα ακόμα πλεονέκτημα είναι το πόσο καλά μπορεί να ισορροπήσει το φόρτο εργασίας ανάμεσα στους κόμβους, ακόμα και στην περίπτωση που υπάρχουν αναπάντεχες καθυστερήσεις (π.χ. λόγω δικτύου). Με το Hadoop θα έπρεπε να είμαστε πολύ προσεκτικοί στον τρόπο που θα μοιράζαμε τα δεδομένα ανάμεσα στις διεργασίες, αφού η οποιαδήποτε ανισορροπία θα μπορούσε να οδηγήσει σε μεγάλη καθυστέρηση στην εκτέλεση των ερωτημάτων. Το Spark από την άλλη είναι σε θέση να υποστηρίξει σχετικά άνετα μερικές χιλιάδες διεργασίες.

Πειραματικά αποτελέσματα από τη σύγκριση Hive-Shark

Έχοντας αναλύσει τα χαρακτηριστικά στα οποία διαφοροποιείται το Shark από το Hive παραθέτουμε τα αποτελέσματα από την εκτέλεση των ερωτημάτων για τα δύο συστήματα στο σχήμα 5.1.

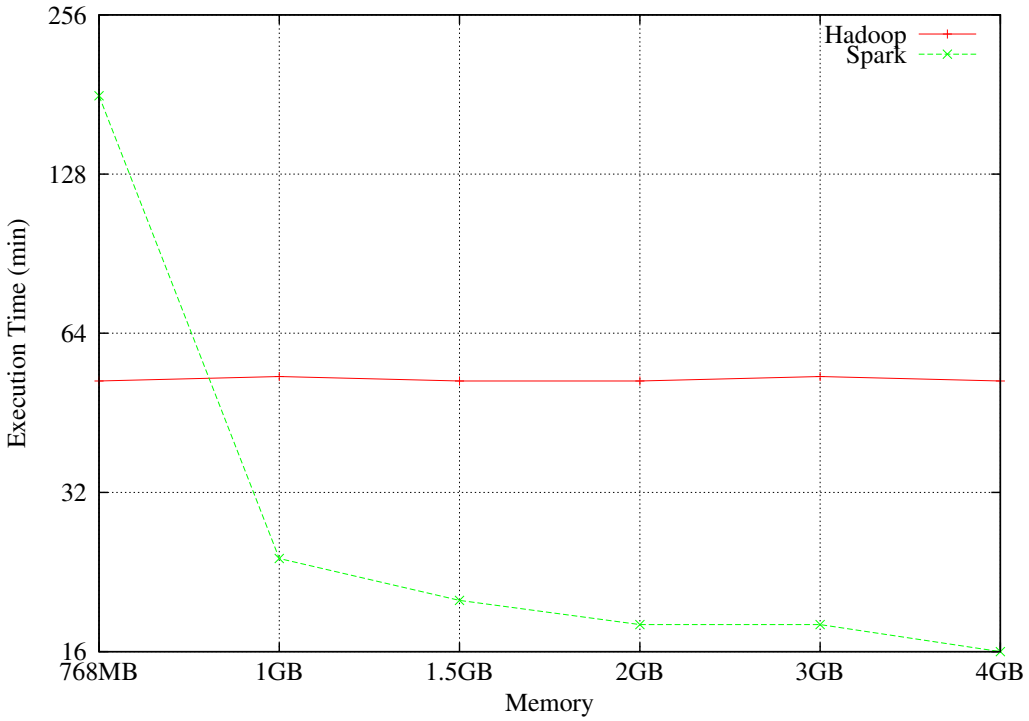


Σχήμα 5.1: Hive vs Shark

Έχουμε να παρατηρήσουμε ότι στην περίπτωση του ερωτήματος με τα αυτόνομα

συστήματα το Shark είναι σημαντικά γρηγορότερο στην εκτέλεση από ότι το Hive. Παρόλα αυτά δεν συμβαίνει το ίδιο στην περίπτωση του άλλου ερωτήματος. Αυτό μπορεί να εξηγηθεί από το γεγονός ότι στη δεύτερη περίπτωση το σημαντικότερο ποσοστό του χρόνου εκτέλεσης μίας διεργασίας (περίπου το 90%) αφιερώνεται στη μεταφορά των απαιτούμενων δεδομένων από την HBase. Έτσι, στη δεύτερη περίπτωση το Shark δεν μπορεί να εκμεταλλευτεί πλήρως τα πλεονεκτήματα που αναφέρθηκαν παραπάνω. Αντίθετα, στην πρώτη περίπτωση που όλες οι λειτουργίες που απαιτούνται γίνονται in-memory φαίνεται η υπεροχή του Spark που οφείλεται κυρίως στον τρόπο που δρομολογεί τις διεργασίες και στο γεγονός ότι αποφεύγει να γράφει όλα τα ενδιάμεσα αποτελέσματα στο δίσκο.

Ένα ακόμα ενδιαφέρον χαρακτηριστικό είναι το πώς επηρεάζει το χρόνο εκτέλεσης η μνήμη που διαθέτουμε στον κάθε worker που τρέχει διεργασίες Map Reduce στην περίπτωση του Hadoop και του Spark. Η συμπεριφορά αυτή παρουσιάζεται στο σχήμα 5.2.



Σχήμα 5.2: Επίδραση διαθέσιμης μνήμης στην εκτέλεση των ερωτημάτων

Είναι ενδιαφέρον να παρατηρήσουμε ότι στην περίπτωση του Hadoop η μνήμη που διαθέτουμε δεν παίζει σημαντικό ρόλο και ο χρόνος εκτέλεσης είναι ουσιαστικά ο ίδιος. Αυτό συμβαίνει, διότι το Hadoop γράφει όλα τα ενδιάμεσα αποτελέσματα στο δίσκο και επομένως δεν εκμεταλλεύεται την παραπάνω μνήμη. Σε αντίθεση, το Spark κάνει εκτεταμένη χρήση της μνήμης για αποθήκευση των ενδιάμεσων αποτελεσμάτων και

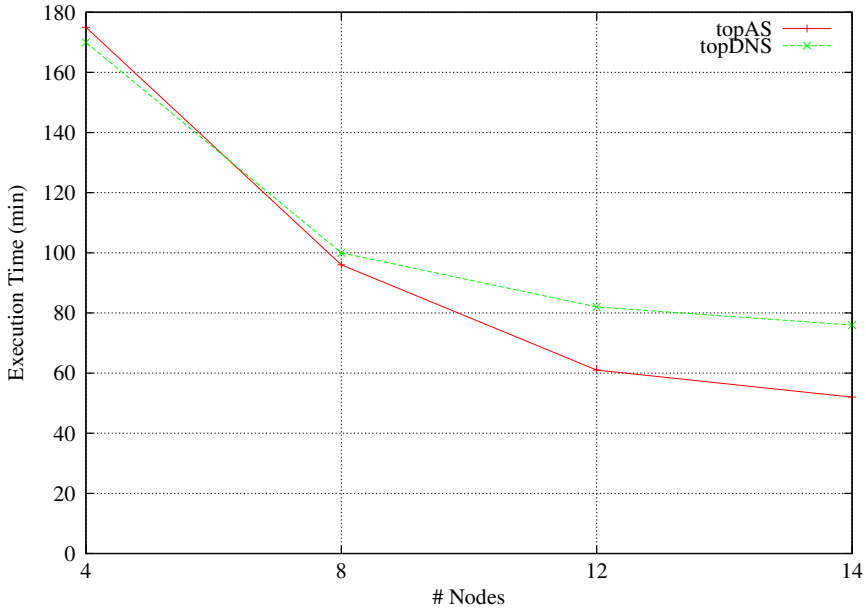
για αυτό παρατηρούμε μία σταδιακή μείωση στο χρόνο εκτέλεσης, καθώς αυξάνουμε τη διαθέσιμη μνήμη. Ειδικά στην περίπτωση που δίνουμε λιγότερο από 1GB ο χρόνος εκτέλεσης είναι πολλές φορές μεγαλύτερος από τις υπόλοιπες, γεγονός που μπορεί να εξηγηθεί αν αναλογιστούμε ότι αυτή η ποσότητα μνήμης δεν επαρκεί για να αποθηκεύονται τα ενδιάμεσα δεδομένα που απαιτούνται με αποτέλεσμα να συμβαίνουν συχνά spills στο δίσκο. Από την άλλη, όσο αυξάνεται η μνήμη τόσο περισσότερα δεδομένα χωράνε σε αυτή, τα spills στο δίσκο γίνονται πιο σπάνια και η εκτέλεση των ερωτημάτων επιταχύνεται. Τέλος, να σημειώσουμε ότι η αύξηση της μνήμης έχει νόημα μέχρι το σημείο που δεν επηρεάζονται οι υπόλοιπες διεργασίες που τρέχουν στους κόμβους (δηλαδή οι διεργασίες για το HDFS και την HBase) για αυτό η μέγιστη τιμή που εξετάσαμε ήταν τα 4GB.

5.2 Μελέτη κλιμακωσιμότητας του συστήματος

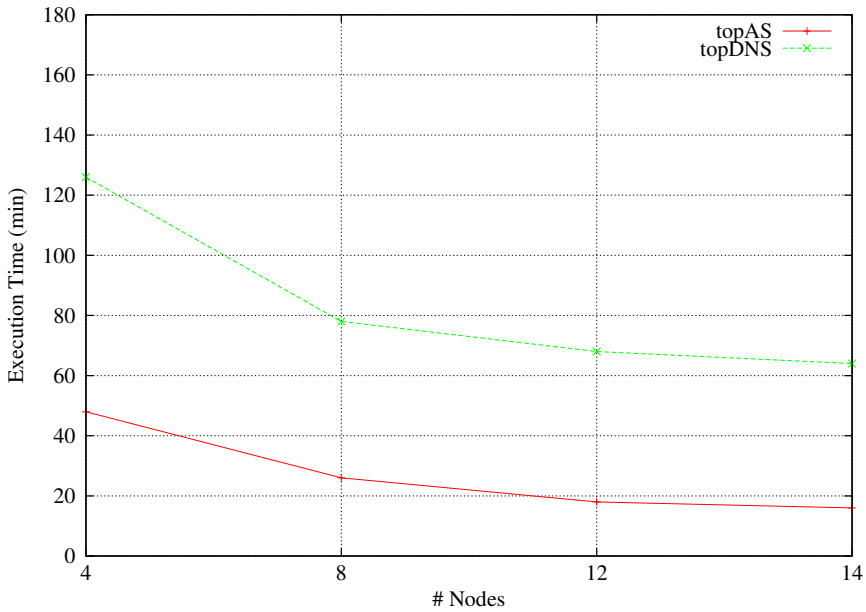
5.2.1 Κλιμακωσιμότητα ως προς τον αριθμό των κόμβων του cluster

Σε αυτή την ενότητα εξετάζουμε τη δυνατότητα κλιμάκωσης των ερωτημάτων που εκτελούμε όσον αφορά τους διαθέσιμους κόμβους του συστήματος. Στο πείραμα αυτό θεωρούμε ότι τα δεδομένα που βρίσκονται ισοκατανεμημένα στους κόμβους του συστήματος παραμένουν αμετάβλητα και αυξομειώνουμε το μέγιστο αριθμό των πιθανών διεργασιών map και reduce που μπορούν να τρέξουν σε κάθε κόμβο. Μηδενίζοντας τον αριθμό αυτό, ουσιαστικά αποκλείουμε από την εκτέλεση του ερωτήματος συγκεκριμένους κόμβους του συστήματος. Στο σχήμα 5.3 παρουσιάζονται τα αποτελέσματα που προέκυψαν.

Οι παρατηρήσεις που έχουμε να κάνουμε είναι οι εξής. Αρχικά, στην περίπτωση του Hive παρατηρούμε ότι για μικρό αριθμό κόμβων τα δύο διαφορετικά είδη ερωτημάτων χρειάζονται τον ίδιο περίπου χρόνο για να τρέξουν (σχήμα 5.3α΄). Θα ήταν αναμενόμενο, ενδεχομένως, το ερώτημα με τα DNS ονόματα που πρέπει να μεταφέρει δεδομένα από την HBase να χρειάζεται περισσότερο χρόνο από ότι το άλλο ερώτημα. Αυτό όπως φαίνεται συμβαίνει όσο μεγαλώνει ο αριθμός των διαθέσιμων κόμβων. Πρέπει να λάβουμε υπόψη μας το γεγονός ότι στην περίπτωση που λίγοι κόμβοι (π.χ. 4) προσπαθούν να συνδεθούν στην HBase και να μεταφέρουν δεδομένα, επειδή οι αιτήσεις μοιράζονται και στους 14 κόμβους που βρίσκονται τα δεδομένα το bottleneck που δημιουργείται είναι σαφώς λιγότερο από την περίπτωση που όλοι οι κόμβοι τρέχουν mapper διεργασίες που προσπαθούν να αποκτήσουν πρόσβαση στα δεδομένα. Έτσι, στην περίπτωση των λίγων κόμβων ο χρόνος δεν αυξάνεται αναλογικά όπως θα περιμέναμε. Επιπλέον, πρέπει να τονίσουμε είναι ότι το ερώτημα με τα αυτόνομα συστήματα απαιτεί να εκτελεστούν περίπου 4 φορές περισσότερες διεργασίες map. Αυτό, δεδομένου ότι το Hive δεν κάνει το ίδιο αποδοτική δρομολόγηση



(α′) Hive



(β′) Shark

Σχήμα 5.3: Κλιμακωσιμότητα ως προς τον αριθμό των κόμβων

των διεργασιών, έχει ως αποτέλεσμα να εισάγει μία σημαντική καθυστέρηση, ικανή να ισοσταθμίσει την καθυστέρηση της φόρτωσης των δεδομένων από την HBase του άλλου ερωτήματος. Όσο βέβαια μεγαλώνει ο αριθμός των διαθέσιμων κόμβων και οι

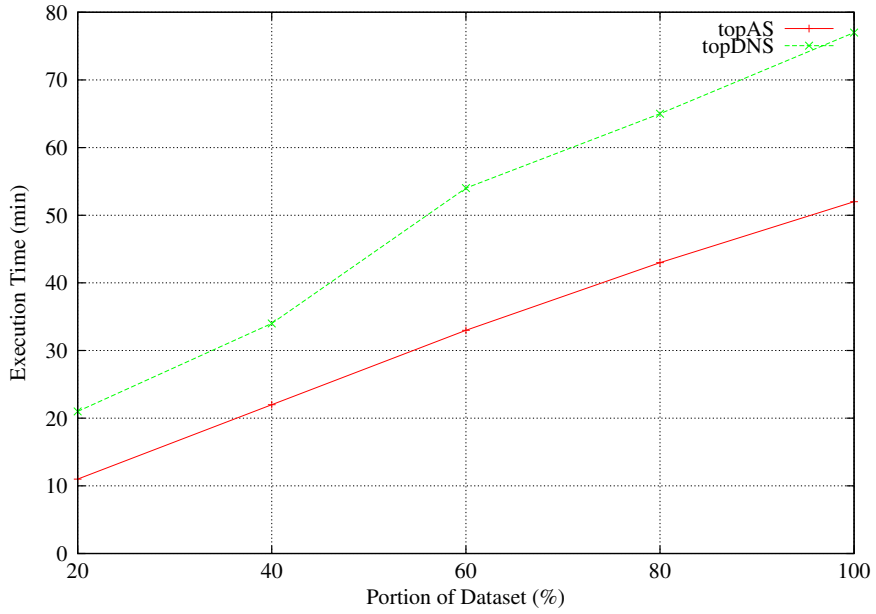
διεργασίες μπορούν να κατανεμηθούν ταυτόχρονα σε περισσότερους κόμβους, αυτή η συμπεριφορά εξασθενίζει και έτσι το ερώτημα με τα αυτόνομα συστήματα ολοκληρώνει πιο γρήγορα την εκτέλεσή του. Συνολικά, το ερώτημα με τα ονόματα DNS δεν κλιμακώνει πολύ καλά σε σχέση με το άλλο που προσεγγίζει ικανοποιητικά τη γραμμική κλιμάκωση (linear scaling) καθώς αυξάνονται οι κόμβοι. Αυτό πιθανότατα συμβαίνει γιατί κάποια από τα αρχεία περιέχουν σχετικά μεγάλο εύρος από διευθύνσεις IP και το overhead να μεταφερθούν οι εγγραφές από την HBase οδηγεί κάποιες διεργασίες να διαρκούν πολύ χρόνο πράγμα που επισκιάζει τη συνολική απόδοση. Στο σχήμα 5.3β΄ βλέπουμε τα αντίστοιχα αποτελέσματα για το Shark. Πάλι, παρατηρούμε ότι το ερώτημα topDns δεν κλιμακώνει τόσο καλά όσο το άλλο. Όμως εδώ βλέπουμε ότι το ερώτημα topAS εκτελείται σε σημαντικά λιγότερο χρόνο από το άλλο ανεξάρτητα από τον αριθμό των διαθέσιμων κόμβων. Φαίνεται, λοιπόν, πάλι η υπεροχή του Shark στη δρομολόγηση των διεργασιών που καταφέρνει να επικαλύψει το γεγονός ότι το ένα ερώτημα αποτελείται από πολλές map διεργασίες, ενώ από την άλλη, η καθυστέρηση που εισάγεται από τη φόρτωση των δεδομένων από την HBase είναι εμφανής στο άλλο ερώτημα με τα DNS ονόματα.

5.2.2 Κλιμακωσιμότητα ως προς το μέγεθος του dataset

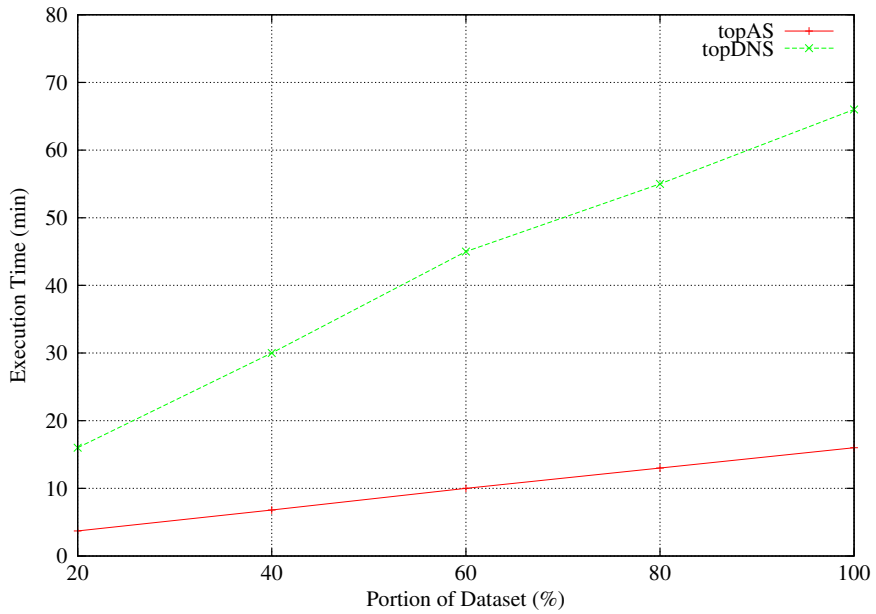
Σε αυτή την ενότητα πειραματιζόμαστε με το μέγεθος του συνολικού dataset που αναλαμβάνει να επεξεργαστεί το σύστημα. Στα παρακάτω πειράματα χρησιμοποιήσαμε όλους τους διαθέσιμους κόμβους του συστήματος, ενώ αυτή τη φορά μεταβάλαμε το μέγεθος των δεδομένων προς επεξεργασία. Στο σχήμα 5.4 παρουσιάζονται τα αποτελέσματα τόσο για το Hive όσο και για το Shark.

Τα συμπεράσματα από αυτό το πείραμα είναι παρεμφερή για τα δύο διαφορετικά συστήματα εκτέλεσης Map Reduce. Συγκεκριμένα, τα δύο διαφορετικά ερωτήματα κλιμακώνουν αρκετά καλά, προσεγγίζοντας σε μεγάλο βαθμό την ιδανική συμπεριφορά. Η τάση αυτή είναι ξανά περισσότερο εμφανής στην περίπτωση του ερωτήματος topAS, ενώ για το άλλο ερώτημα οι χρόνοι ολοκλήρωσης είναι γενικά μεγαλύτεροι ανεξαρτήτως του μεγέθους των δεδομένων, λόγω του overhead που εισάγει η μεταφορά των δεδομένων από την HBase . Στην περίπτωση του ερωτήματος topDns ενδιαφέρον έχει επίσης να παρουσιάσουμε τη συμπεριφορά του partitioning που γίνεται κατά την προεπεξεργασία των δεδομένων. Το σχήμα 5.5 δείχνει αυτή τη συμπεριφορά.

Να τονίσουμε ότι στο χρόνο του partitioning έχουμε υπολογίσει αθροιστικά τόσο το χρόνο για τη δειγματοληψία των δεδομένων όσο και το χρόνο για την κατάτμηση των δεδομένων σε διαφορετικά αρχεία που περιέχουν γνωστό εύρος τιμών. Επίσης, για τις μετρήσεις χρησιμοποιήθηκε μόνο το Hive. Όπως μπορούμε να παρατηρήσουμε, το pre-partitioning των δεδομένων κλιμακώνει και αυτό ικανοποιητικά σε σχέση με το μέγεθος των δεδομένων, πράγμα που σημαίνει ότι το σύστημά μας μπορεί να αντεπεξέλθει επιτυχώς σε διάφορες καταστάσεις με διαφορετικές κάθε φορά παρα-



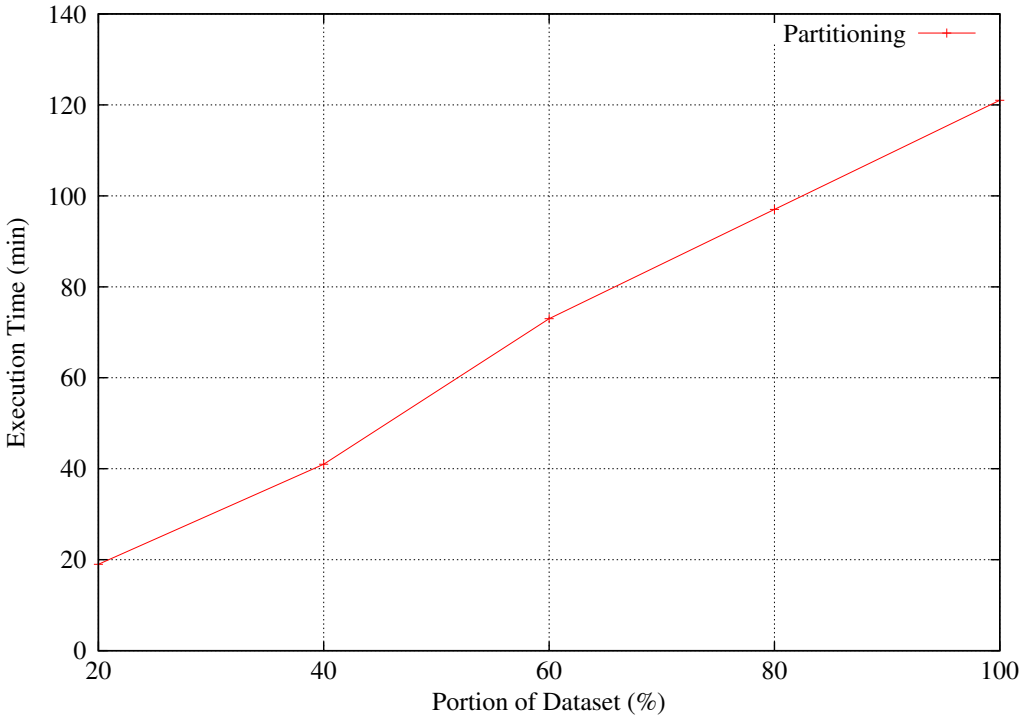
(α′) Hive



(β′) Shark

Σχήμα 5.4: Κλιμακωσιμότητα ερωτημάτων ως προς το μέγεθος του dataset

μέτρους. Για την ακρίβεια, από το 60% των δεδομένων και πάνω η απόδοση του partitioning προκύπτει λίγο χειρότερη από γραμμική, γεγονός που οφείλεται στο ότι πρέπει να δρομολογηθούν αρκετά περισσότερες διεργασίες, πράγμα που εισάγει μία



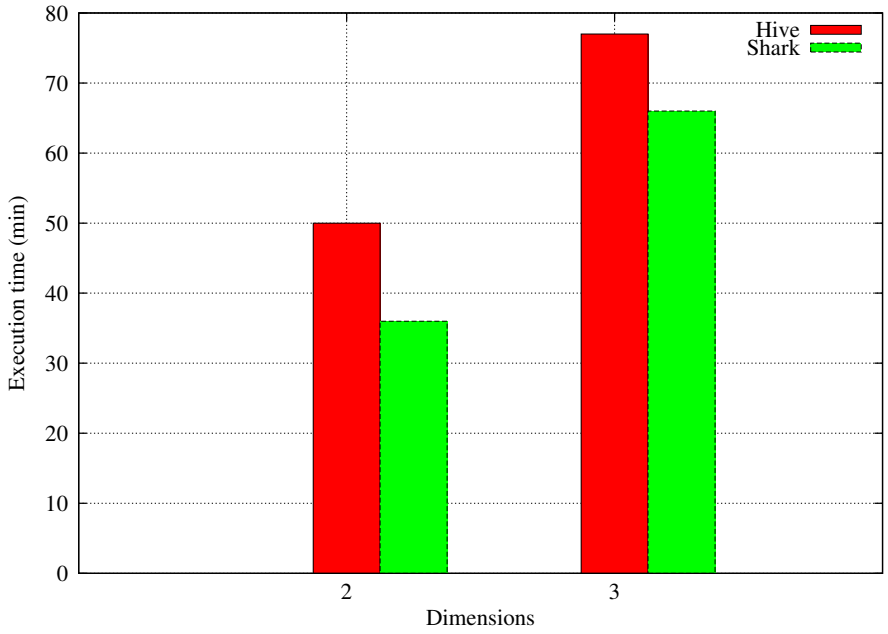
Σχήμα 5.5: Κλιμακωσιμότητα partitioning ως προς το μέγεθος του dataset

μικρή καθυστέρηση.

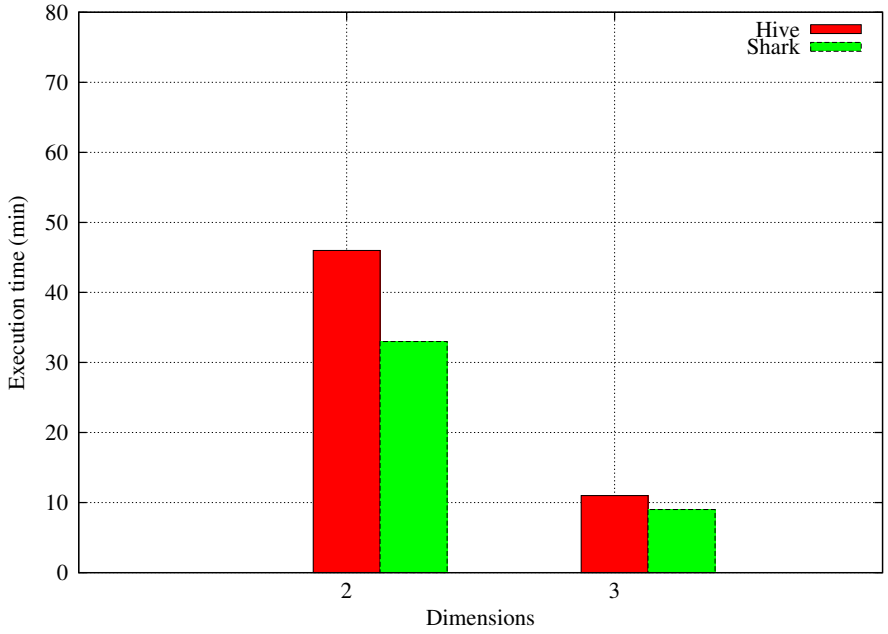
5.3 Επίδραση αριθμού διαστάσεων K-d tree

Στρέφουμε την προσοχή μας τώρα σε μία τελευταία παράμετρο του συστήματος, τον αριθμό των διαστάσεων που χρησιμοποιούμε για το partitioning στο K-d tree. Για τους σκοπούς της διπλωματικής εξετάσαμε τη συμπεριφορά του συστήματος όταν χρησιμοποιούμε μόνο δύο διαστάσεις, τη διεύθυνση προέλευσης και τη διεύθυνση προορισμού, σε σχέση με την περίπτωση που χρησιμοποιούμε τρεις διαστάσεις, τις δύο προαναφερθείσες και την ημερομηνία του πακέτου.

Στο σχήμα 5.6α΄ παρουσιάζεται ο χρόνος εκτέλεσης του ίδιου ερωτήματος στις δύο περιπτώσεις με τις 2 και 3 διαστάσεις. Αυτό που παρατηρούμε είναι ότι στην περίπτωση με τις περισσότερες διαστάσεις η εκτέλεση του ερωτήματος απαιτεί περισσότερο χρόνο κατά ένα σημαντικό ποσοστό σε σχέση με τις λιγότερες διαστάσεις, ανεξαρτήτως του συστήματος που χρησιμοποιείται. Η συμπεριφορά αυτή είναι αναμενόμενη αν σκεφτούμε ότι στην περίπτωση που χρησιμοποιούμε μία παραπάνω διάσταση αλλάζει ο τρόπος που κατανέμονται τα δεδομένα και πλέον σε ένα αρχείο οι διευθύνσεις IP που περιέχονται μπορεί να ανήκουν σε ένα μεγαλύτερο εύρος, πράγμα που συνεπάγεται ότι



(α′) Εκτέλεση ερωτήματος topDns

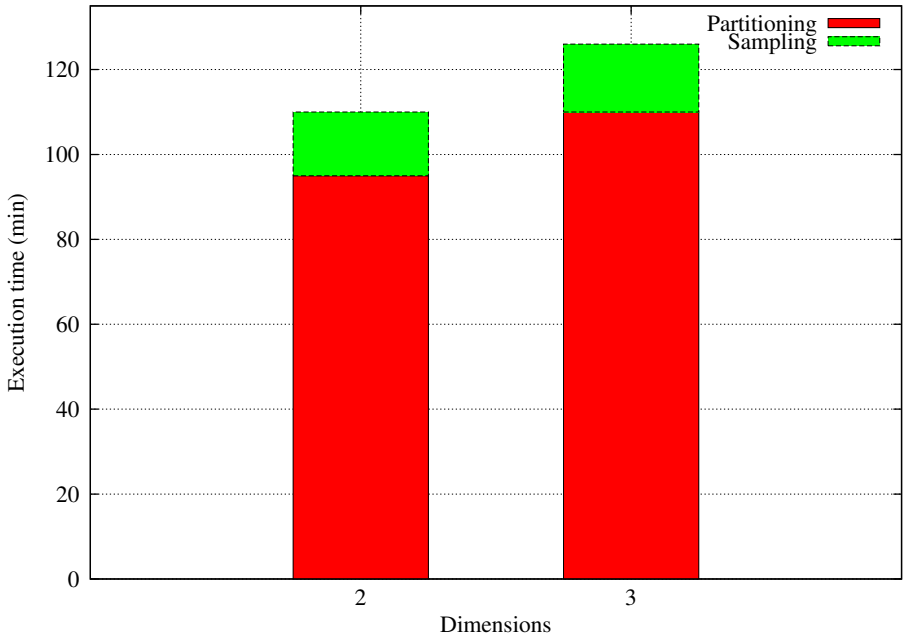


(β′) Filtering σε μια εβδομάδα δεδομένων

Σχήμα 5.6: Επίδραση αριθμού διαστάσεων

χρειάζεται περισσότερος χρόνος για τη μεταφορά τους από την HBase. Στο σχήμα 5.6β′ παρουσιάζουμε το χρόνο που απαιτείται για την εκτέλεση ενός ερωτήματος σε

δεδομένα μίας εβδομάδας. Παρατηρούμε ότι στις 3 διαστάσεις, όπου συμπεριλαμβάνεται η ημερομηνία σαν μία διάσταση, το ίδιο ερώτημα εκτελείται σε σημαντικά λιγότερο χρόνο, αφού δεν απαιτείται η προσπέλαση όλων των αρχείων, παρά μόνο αυτών που γνωρίζουμε ότι πιθανώς περιέχουν τις ζητούμενες τιμές. Αντίθετα, στις 2 διαστάσεις, όπου έχουμε μόνο την πληροφορία για τις διευθύνσεις IP που περιέχει κάθε αρχείο, ο χρόνος δεν διαφέρει σημαντικά από αυτόν που χρειάζεται για το πλήρες ερώτημα. Ακόμα και στην περίπτωση που ψάχνουμε τα στατιστικά για μία εβδομάδα, πρέπει να προσπελάσουμε όλα τα αρχεία για να αποφανθούμε ποιες ημερομηνίες μας ενδιαφέρουν και ποιες όχι.



Σχήμα 5.7: Επίδραση διαστάσεων κατά τη φόρτωση των δεδομένων

Το σχήμα 5.7 παρουσιάζει την επίδραση του αριθμού των διαστάσεων κατά την προεπεξεργασία και φόρτωση των δεδομένων. Όπως είναι λογικό ο χρόνος για τη δειγματοληψία είναι σχεδόν ο ίδιος και στις δύο περιπτώσεις. Αντιθέτως, ο χρόνος για το partitioning είναι ελαφρώς μεγαλύτερος στην περίπτωση των 3 διαστάσεων, γεγονός που οφείλεται λογικά στο λίγο μεγαλύτερο χρόνο αναζήτησης για το πού ανήκει μία εγγραφή όταν έχουμε περισσότερες διαστάσεις. Γενικά, λοιπόν, προκύπτει ως συμπέρασμα ότι όταν χρησιμοποιούμε πιο πολλές διαστάσεις, εμφανίζεται ένα επιπλέον overhead αλλά κερδίζουμε τη δυνατότητα να εκτελούμε πιο ευέλικτα ερωτήματα εφαρμόζοντας φίλτρα σε συγκεκριμένα εύρη τιμών από τις υπάρχουσες διαστάσεις.

Κεφάλαιο 6

Επίλογος

6.1 Σύνοψη - Συμπεράσματα

Στην εργασία αυτή ασχοληθήκαμε με τη σχεδίαση και υλοποίηση ενός συστήματος για την αποδοτική επεξεργασία μεγάλου όγκου δεδομένων που προέρχονται από log files με τη χρήση τεχνικών κατανεμημένης επεξεργασίας, όπως το μοντέλο Map Reduce. Τα δεδομένα που χρησιμοποιήθηκαν ως ένα use case προέρχονται από τον κόμβο GR-IX, τον ελληνικό κόμβο ουδέτερης διασύνδεσης. Ο σκοπός της διπλωματικής ήταν ο συνδυασμός αυτής της βασικής πληροφορίας με δευτερεύουσες πληροφορίες που αφορούν το αυτόνομο σύστημα που ανήκουν οι διευθύνσεις IP, τη χώρα προέλευσης και το όνομα του υπολογιστή που ανήκουν.

Αρχικά, εξετάσαμε την περίπτωση όπου η δευτερεύουσα πληροφορία περιέχεται σε ένα αρχείο που χωράει στην κύρια μνήμη της διεργασίας map που αναλαμβάνει την επεξεργασία του κάθε κομματιού του αρχείου εισόδου. Προτείναμε τη χρήση της τεχνικής του map join σε συνδυασμό με τον ορισμό μίας κατάλληλης συνάρτησης UDF που θα αναλάβει τη μετατροπή της διεύθυνσης IP στο αυτόνομο σύστημα που ανήκει. Ο τρόπος αυτός αποδείχτηκε σημαντικά καλύτερος από τη βασική υλοποίηση του join σε συστήματα όπως το Hive τόσο από πλευράς του συνολικού χρόνου εκτέλεσης όσο και από το συνολικό όγκο δεδομένων που πρέπει να ανταλλαχθούν μεταξύ των διεργασιών. Συγκεκριμένα, επέφερε βελτίωση κατά 69.4% σε σχέση με την απλή προσέγγιση του join που πραγματοποιεί το Hive.

Στη συνέχεια, επεκτείναμε την ανάλυση αυτή και σε περιπτώσεις όπου η δευτερεύουσα πληροφορία δεν μπορεί να χωρέσει στην κύρια μνήμη λόγω μεγέθους. Στην περίπτωση αυτή, προτείναμε δύο τρόπους για να γίνει μια προεπεξεργασία των δεδομένων, ώστε να γνωρίζουμε σε κάθε αρχείο το πιθανό εύρος των τιμών που βρίσκονται σε αυτό και να είμαστε σε θέση να γνωρίζουμε εκ των προτέρων ποιο τμήμα των metaδεδομένων θα χρειαστούμε. Ο δυναμικός τρόπος διαχωρισμού με τη βοήθεια ενός K-d tree αποδείχτηκε πιο αποτελεσματικός σε σχέση με το στατικό τόσο στο διαχωρισμό

των δεδομένων σε ισοκατανεμημένα αρχεία, όσο και στην εκτέλεση ερωτημάτων στα οποία θέλουμε να κάνουμε filtering ως προς μία συγκεκριμένη διάσταση. Συγκριτικά με την απλή προσέγγιση του join στο Hive πετύχαμε μία βελτίωση περίπου 70%.

Τέλος, παρουσιάσαμε τη συμπεριφορά και τις δυνατότητες του συστήματος σε σχέση με διάφορες παραμέτρους που αφορούν το σύστημα εκτέλεσης του Map Reduce που θα χρησιμοποιηθεί, τον αριθμό των κόμβων του συστήματος, το μέγεθος του dataset και τον αριθμό των διαστάσεων που χρησιμοποιούμε στο K-d tree. Στα πειράματα αυτά διαπιστώσαμε ότι το σύστημα κλιμακώνει καλά όσο προστίθενται κόμβοι στο cluster αλλά και όσο αυξάνεται το μέγεθος των δεδομένων. Επίσης, ο αριθμός των διαστάσεων που χρησιμοποιούμε στο K-d tree επηρεάζουν αρνητικά το χρόνο εκτέλεσης, όμως προσφέρουν μεγαλύτερη ευελιξία στα ερωτήματα που μπορούμε να εκτελέσουμε, ειδικά σε ερωτήματα filtering ως προς μία ή περισσότερες διαστάσεις. Μία τελευταία παρατήρηση είναι η υπεροχή του Shark σε σχέση με παραδοσιακά συστήματα, όπως το Hive σε ορισμένες περιπτώσεις ανάλογα με την εφαρμογή προς εκτέλεση.

6.2 Μελλοντικές Επεκτάσεις

Ως επεκτάσεις της παρούσας διπλωματικής εργασίας, προτείνουμε την περαιτέρω μελέτη του τρόπου που γίνεται το partitioning των δεδομένων. Μπορούν να εξεταστούν διαφορετικοί αλγόριθμοι κατανομής των δεδομένων ή και εναλλακτικές δομές δεδομένων αντί του K-d tree. Μπορεί επίσης να εξεταστεί περαιτέρω η χρήση πιο πολλών διαστάσεων στην εκτέλεση ερωτημάτων, πράγμα που θα επιτρέψει μεγαλύτερη ευελιξία στα ερωτήματα που εκτελούνται και να μελετηθεί η επιρροή περισσότερων διαστάσεων στο χρόνο εκτέλεσης. Ακόμα, μπορεί να γίνει χρήση διαφορετικών γλωσσών επεξεργασίας των ερωτημάτων αντί της HiveQL, όπως η πιο πρόσφατη Spark SQL. Τέλος, μπορεί να μελετηθεί η χρήση συστημάτων που επιτρέπουν την εκτέλεση ερωτημάτων κατά προσέγγιση κάνοντας δειγματοληψία στα δεδομένα εισόδου και παρουσιάζοντας τα αποτελέσματα με ένα ποσοστό λάθους, όπως για παράδειγμα το BlinkDB [7].

Βιβλιογραφία

[1] Data, data everywhere. http://www.economist.com/node/15557443.

[2] GeoLite Database. http://dev.maxmind.com/geoip/legacy/geolite/.

[3] HBase Bulk Loading. http://hbase.apache.org/book/arch.bulk.load.html.

[4] Internet Scans. http://scans.io/.

[5] SFlow Tool. http://www.sflow.org/.

[6] TreeMap. http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html.

[7] Agarwal Sameer, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[8] Rudolf Bayer. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta informatica*, 1(4):290–306, 1972.

[9] Bentley Jon Louis. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 975–986. ACM, 2010.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[12] Nikolaos Chatzis, Georgios Smaragdakis, Jan Böttger, Thomas Krenc, and Anja Feldmann. On the Benefits of Using a Large IXP as an Internet Vantage Point. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 333–346. ACM, 2013.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] Vivekanand Gopalkrishnan, Qing Li, and Kamalakar Karlapalem. Star/Snow-Flake Schema Driven Object-Relational Data Warehouse Design and Query Processing Strategies. In *DataWarehousing and Knowledge Discovery*, pages 11–22. Springer, 1999.

[15] Liyin Tang and Namit Jain. Join Strategies in Hive. *Hive Summit*, 2011.

[16] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[17] V. Koukis, C. Venetsanopoulos, and N. Koziris. ∼okeanos: Building a Cloud, Cluster by Cluster. *IEEE Internet Computing*, 17(3):67–71, 2013.

[18] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the 2013 international conference on Management of data*, pages 13–24. ACM, 2013.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.